ZiLOG

ZiLOG Developer Studio II—eZ80Acclaim!®

User Manual

UM014423-0607

Warning: DO NOT USE IN LIFE SUPPORT

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

Date	Revision Level	Section	Description
April 2006	21	All	Updated for ZDS II 4.10.0 release.
September 2006	22	“Anonymous Labels” on page 222	Added new section for CR 6971.
		“Warning and Error Messages” on page 168	Added a note for CR 5661.
		“Warning and Error Messages” on page 286	Added a note for CR 5661.
		Chapters 2 and 5	Changed Select Active Configuration to Select Build Configuration.
		Appendix D, “Using the Command Processor”	Updated.
		“Setup” on page 96	Updated step 6.
		“Firmware Upgrade” on page 120	Added path for Ethernet Smart Cable upgrade instructions.
		“Project Settings—Debugger Page” on page 95 and “New Project” on page 39	Added description of the Use Page Erase Before Flashing check box.
		“Flash Loader” on page 115	Added description of the Use Page Erase check box.
June 2007	23	All	Updated for ZDS II 4.11.0 release.

Table of Contents

Introduction

This section covers the following topics:

- "ZDS II System Requirements" on page xvii
- "ZiLOG Technical Support" on page xviii

ZDS II SYSTEM REQUIREMENTS

To effectively use this software and documentation, you need a basic understanding of the C and assembly languages, the eZ80Acclaim! architecture, and Microsoft Windows.

NOTE: The memory requirements might vary from system to system depending on the size of the assembly or C source files. If your system has only 8 MB of RAM, C source files with large functions and very large assembly files might cause an out-of-memory message on your system.

Supported Operating Systems

- Windows Vista**
- Windows XP Professional
- Windows 2000 SP4
- MS Windows 98 SE

NOTE: **The USB Smart Cable is not supported on 64-bit Windows Vista. The Ethernet Smart Cable, available separately in the Ethernet Smart Cable Accessory Kit, is supported.

Recommended Host System Configuration

- Windows XP Professional
- Pentium III 500-MHz processor or higher
- 128-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)
- Ethernet port (when using the Ethernet Smart Cable)
- Internet browser (Internet Explorer or Netscape)

Minimum Host System Configuration

- Windows 98 SE
- Pentium II 233-MHz processor
- 96-MB RAM
- 25-MB hard disk space (application only)
- Super VGA video adapter
- CD-ROM drive for installation
- USB high-speed port (when using the USB Smart Cable)
- Ethernet port (when using the Ethernet Smart Cable)
- Internet browser (Internet Explorer or Netscape)

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.

When Using the Ethernet Smart Cable

- Ethernet 10Base-T compatible connection

When Using the Serial Smart Cable

- RS232 communication port with hardware flow and modem control signals

NOTE: Some USB to RS232 devices are not compatible because they lack the necessary hardware signals and/or they use proprietary auto-sensing mechanisms that prevent the Smart Cable from connecting.

When Using ZPAK II

- Ethernet 10Base-T compatible connection
- RS232 communication port

ZILOG TECHNICAL SUPPORT

For technical questions about our products and tools or for design assistance, please use our web page:

`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 e-mail

To receive ZilOG Developer Studio (ZDS) II product updates and notifications, register at the Technical Support web page.

Before Contacting Technical Support

Before you use technical support, consult the following documentation:

- Readme.txt File

  Refer to the `Readme.txt` file in the following directory for last minute tips and information about problems that might 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!`, `eZ80Acclaim!`, `Crimzon`, or `Z8GP`.

  – *version* is the ZDS II version number. For example, *version* might be `4.9.0` or `5.0.0`.

- FAQ.html file

  The `FAQ.html` file contains answers to frequently asked questions and other information about good practices for 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. For example, *product* can be `ZNEO`, `Z8Encore!`, `eZ80Acclaim!`, `Crimzon`, or `Z8GP`.

  – *version* is the ZDS II version number. For example, *version* might be `4.11.0` or `5.0.0`.

- Troubleshooting sections
  - “Troubleshooting the Assembler” on page 229
  - “Troubleshooting the Linker” on page 283

Getting Started

This chapter describes the tools that make up the eZ80Acclaim! developer's environment and provides a tutorial of the eZ80Acclaim! developer's environment, so you can be working with our graphical user interface in a short time. The following topics are covered:

- "Installing ZDS II" on page 1
- "Developer's Environment Tutorial" on page 1
- "Using Non-Simulator Debug Tools" on page 14

INSTALLING ZDS II

Perform the following procedure to install the eZ80Acclaim! developer's environment:

1. Insert the CD in your CD-ROM drive.
2. Follow the setup instructions on your screen.

The installer displays a default location for ZDS II. You can change the location if you want to.

DEVELOPER'S ENVIRONMENT TUTORIAL

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

In this tour, you do the following:

- "Create a New Project" on page 2
- "Add a File to the Project" on page 6
- "Set Up the Project" on page 8
- "Save the Project" on page 13

When you complete this tour, you have a `sample.lod` file that is used in debugging.

NOTE: Be sure to read "Using the Integrated Development Environment" on page 15 to learn more about all the dialog boxes and their options discussed in this tour.

For the purpose of this tutorial, your eZ80Acclaim! developer's environment directory will be referred to as <*ZDS Installation Directory*>, which equates to the following:

<*ZILOGINSTALL*>`\ZDSII_eZ80Acclaim!_`<*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

1. To create a new project, select **New Project** from the File menu.

   From the New Project dialog box, click on the Browse button (...) to navigate to the directory where you want to save your project.

   The Select Project Name dialog box is displayed as shown in the following figure.

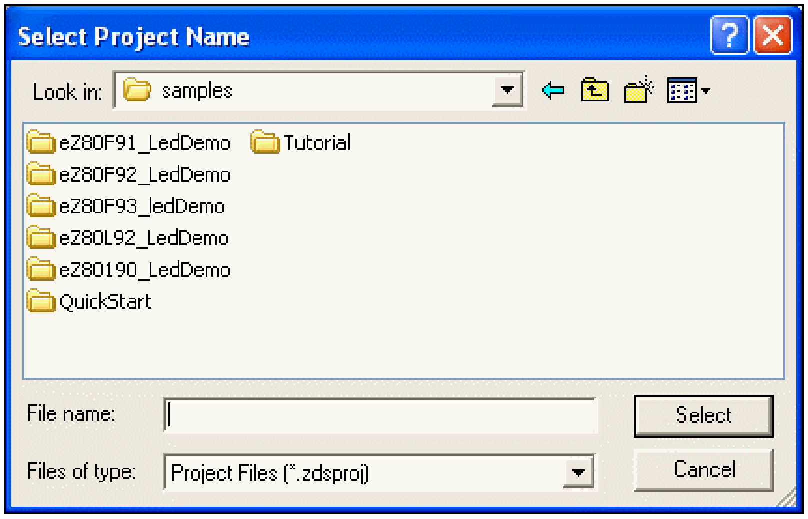


Figure 1. Select Project Name Dialog Box

2. Use the Look In drop-down list box to navigate to the directory where you want to save your project. For this tutorial, place your project in the following directory:

   <ZDS Installation Directory>`\samples\Tutorial`

   If ZiLOG Developer Studio was installed in the default directory, the actual path would be

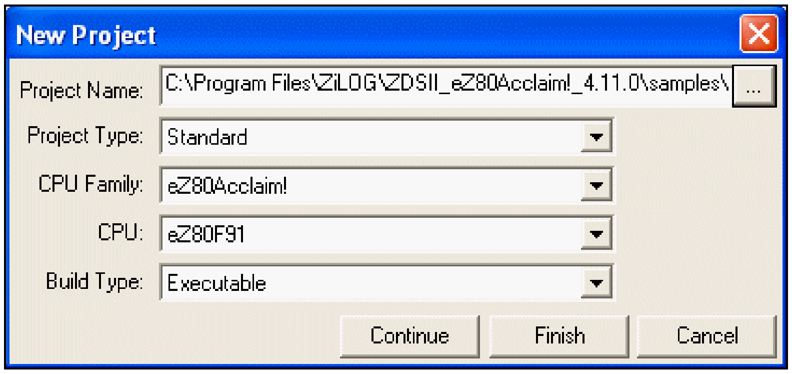
   `C:\Program Files\ZiLOG\ZDSII_eZ80Acclaim!_4.11.0\samples\Tutorial`

3. In the File Name field, type `sample` for the name of your project.

   The eZ80Acclaim! 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 type the extension `.zdsproj`. It is added automatically.

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

5. In the Project Type field, select **Standard** because the `sample` project uses `.c` files.
6. In the CPU Family drop-down list box, select **eZ80Acclaim!**.
7. In the CPU drop-down list box, select **eZ80F91**.
8. In the Build Type drop-down list box, select **Executable** to build an application.

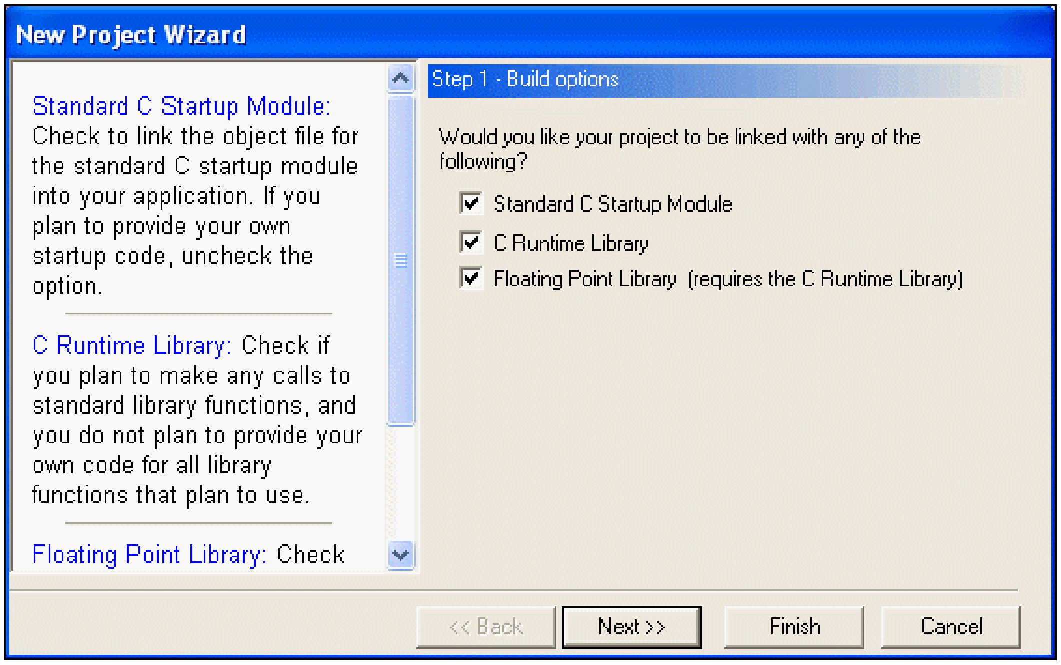

New Project

Project Name: C:\Program Files\ZiLOG\ZDSII_eZ80Acclaim!_4.11.0\samples\
Project Type: Standard
CPU Family: eZ80Acclaim!
CPU: eZ80F91
Build Type: Executable

Figure 2. New Project Dialog Box

9. Click **Continue**.

   The New Project Wizard dialog box is displayed. 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?

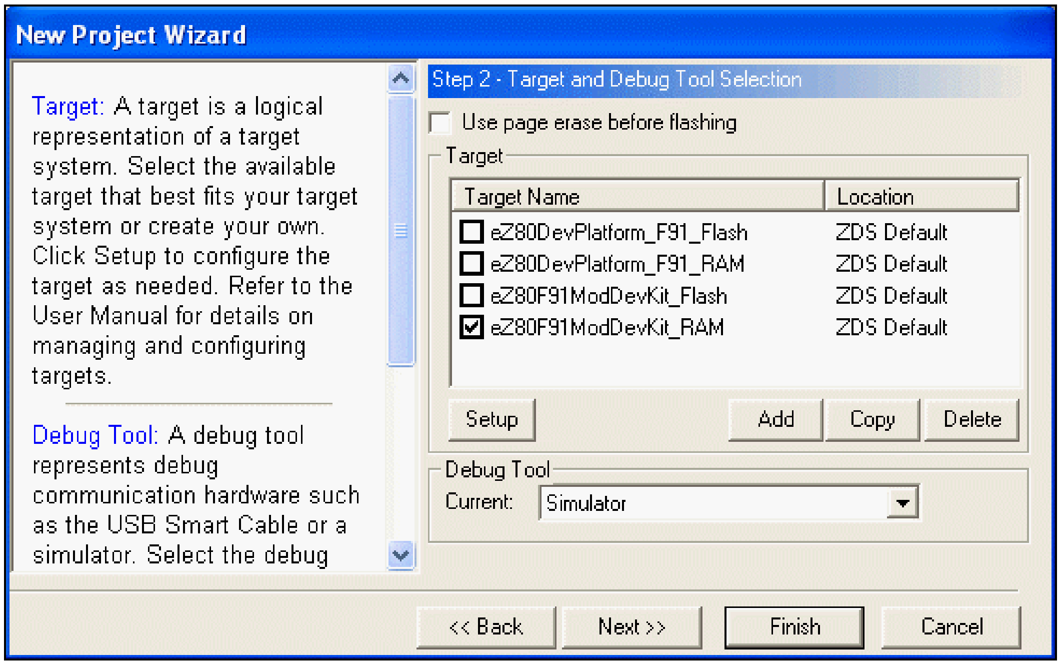
- ✓ Standard C Startup Module
- ✓ C Runtime Library
- ✓ Floating Point Library (requires the C Runtime Library)

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

10. Accept the defaults by clicking **Next**.

    The Target and Debug Tool Selection step of the New Project Wizard dialog box is displayed.

11. Select the eZ80F91ModDevKit_RAM target.


New Project Wizard

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 managing and configuring targets.

Debug Tool: A debug tool represents debug communication hardware such as the USB Smart Cable or a simulator. Select the debug

Step 2 - Target and Debug Tool Selection

☐ Use page erase before flashing

Target

Target Name	Location
☐ eZ80DevPlatform_F91_Flash	ZDS Default
☐ eZ80DevPlatform_F91_RAM	ZDS Default
☐ eZ80F91ModDevKit_Flash	ZDS Default
☑ eZ80F91ModDevKit_RAM	ZDS Default

Debug Tool

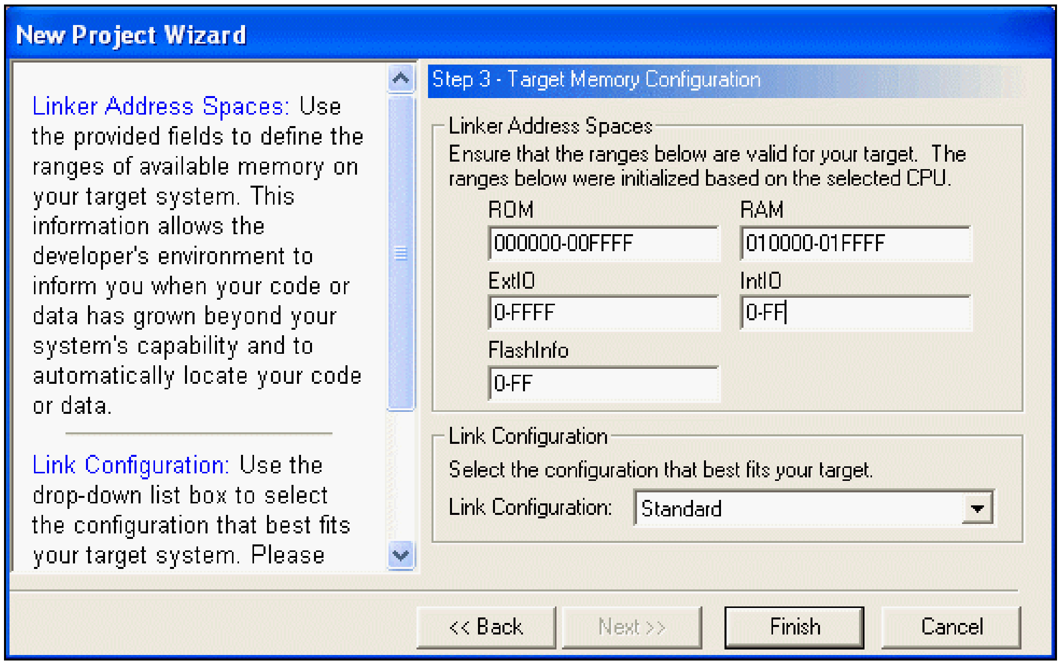
Current: Simulator

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

12. Click **Next**.

    The Target Memory Configuration step of the New Project Wizard dialog box is displayed.

13. Change the Linker Address Spaces fields as follows:
    - ROM: `000000-00FFFF`
    - ExtIO: `0-FFFF`
    - FlashInfo: `0-FF`
    - RAM: `010000-01FFFF`
    - IntIO: `0-FF`



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.

Link Configuration: Use the drop-down list box to select the configuration that best fits your target system. Please

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.

ROM: 000000-00FFFF

RAM: 010000-01FFFF

ExtIO: 0-FFFF

IntIO: 0-FF

FlashInfo: 0-FF

Link Configuration

Select the configuration that best fits your target.

Link Configuration: Standard

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

14. Click **Finish**.

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

Add a File to the Project

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

1. From the Project menu, select **Add Files**.

   The Add Files to Project dialog box is displayed.

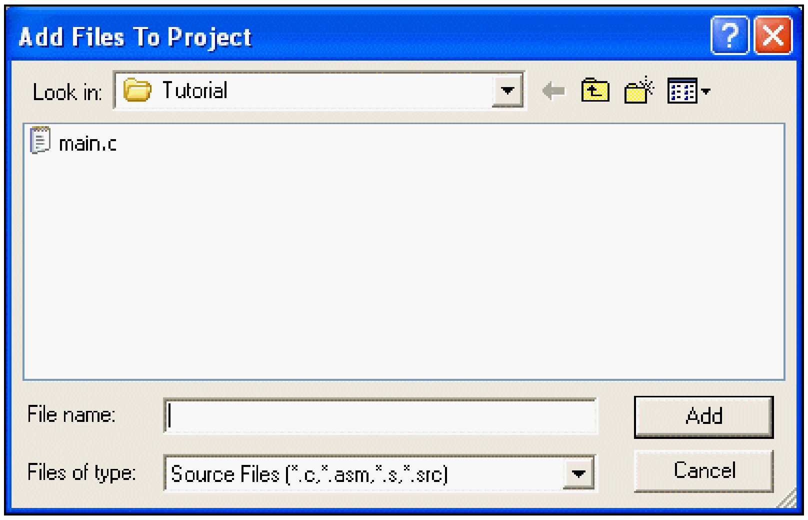


Figure 6. Add Files to Project Dialog Box

2. From the Add Files to Project dialog box, use the Look In drop-down list box to navigate to the following directory:

   <ZDS Installation Directory>`\samples\Tutorial`

3. Select the `main.c` file and click **Add**.

   The `main.c` file is then displayed under the Standard Project Files folder in the Project Workspace window on the left side of the IDE.

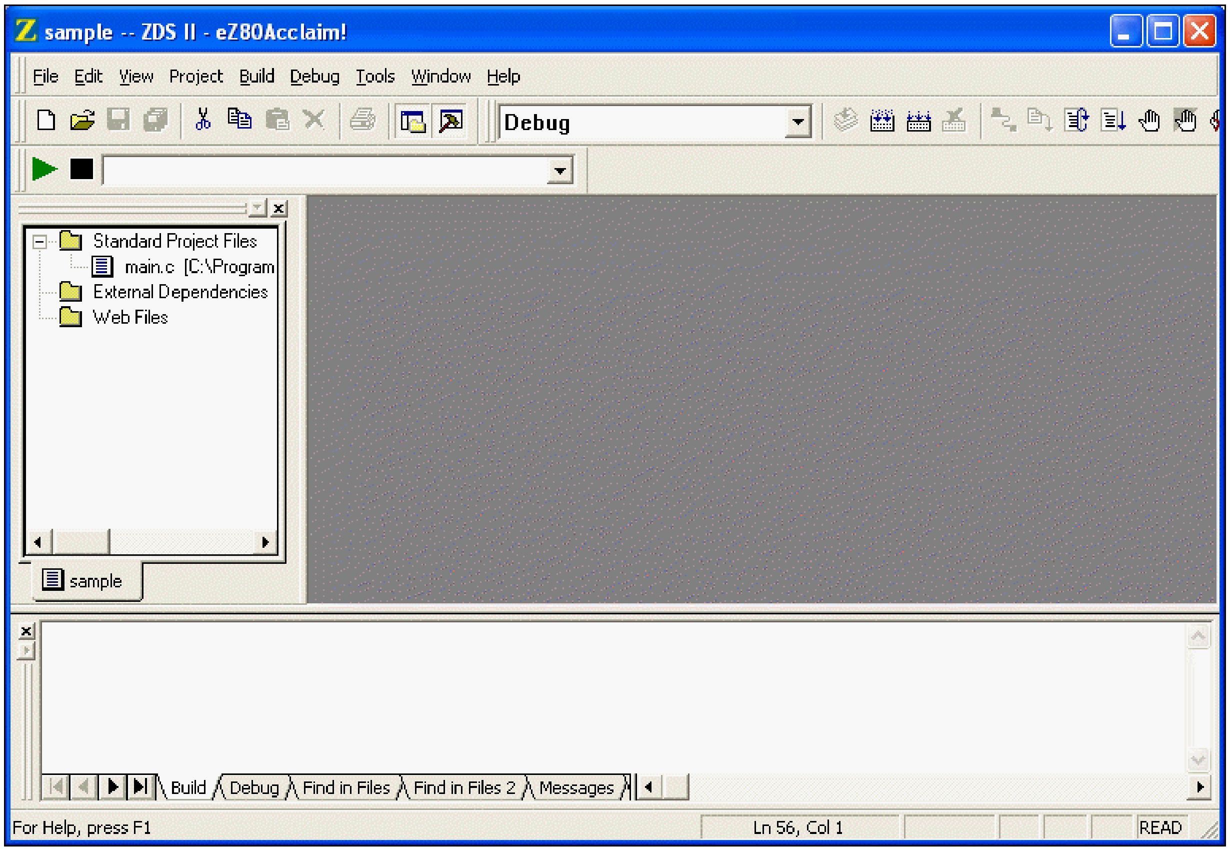


Figure 7. Sample Project

NOTE: To view any of the files in the Edit window during the tutorial, double-click on the file in the Project Workspace window.

Set Up the Project

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

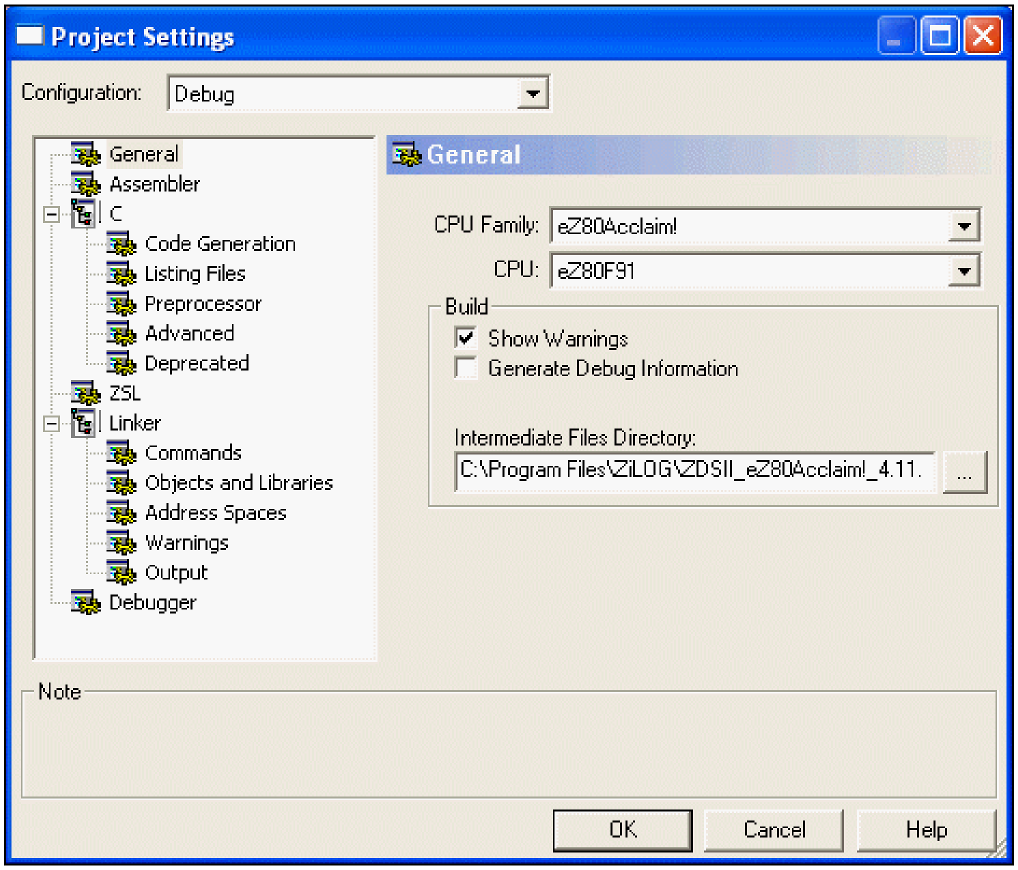
1. From the Project menu, select **Settings**.

   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, see “Settings” on page 55.

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

For your convenience, the Debug configuration is a predefined configuration of defaults set to enable the debugging of program code. For more information on project configurations such as adding your own configuration, see "Set Active Configuration" on page 108.

3. Select the General page.
4. Deselect the Generate Debug Information check box.



Configuration: Debug

General

CPU Family: eZ80Acclaim!

CPU: eZ80F91

Build

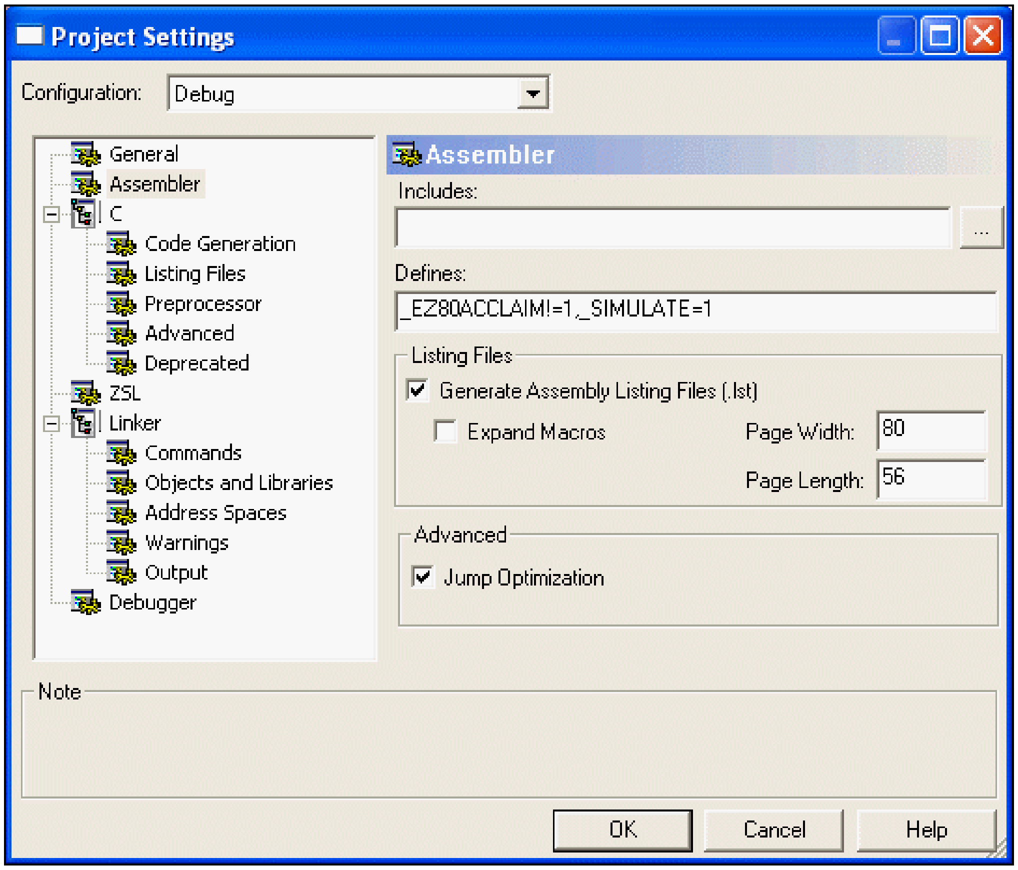
[✓] Show Warnings

[ ] Generate Debug Information

Intermediate Files Directory: C:\Program Files\ZiLOG\ZDSII_eZ80Acclaim!_4.11.

Figure 8. General Page of the Project Settings Dialog Box

5. Select the Assembler page.
6. Make sure that the Generate Assembly Listing Files (.lst) and Jump Optimization check boxes are selected.



Configuration: Debug

Includes:

Defines: _EZ80ACCLAIM!=1,_SIMULATE=1

Listing Files

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

Page Width: 80

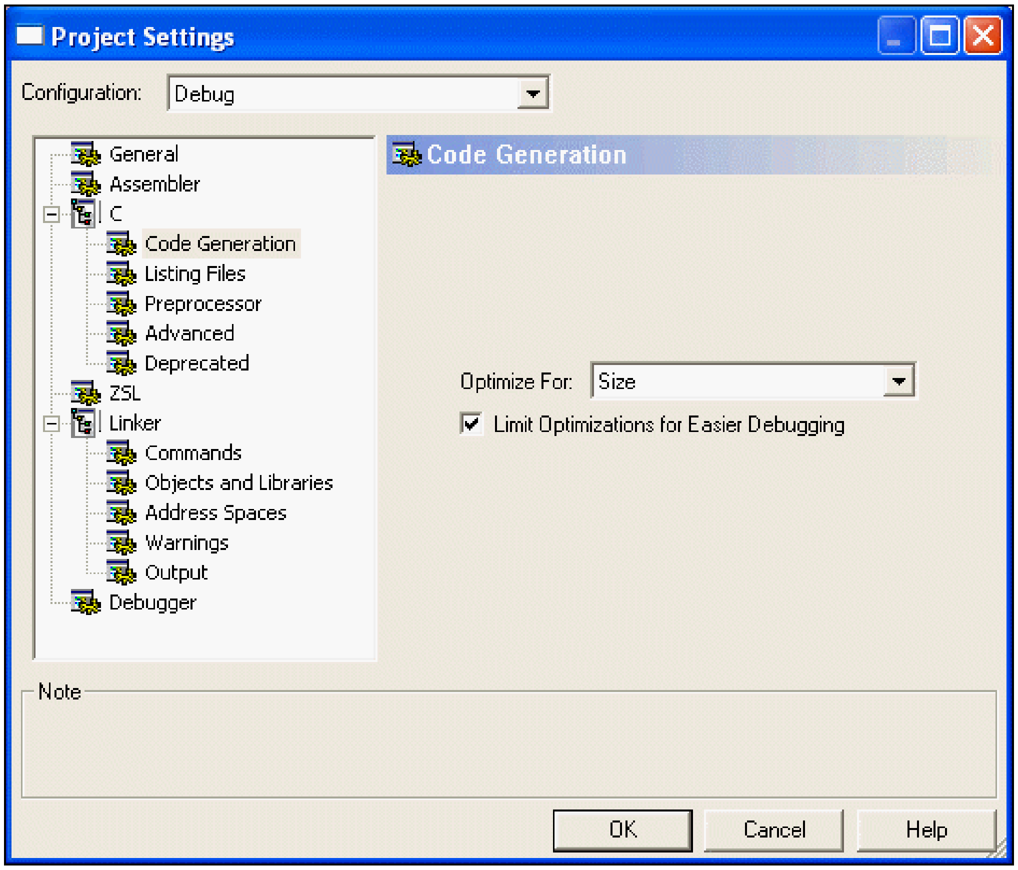
Page Length: 56

Advanced

- [✓] Jump Optimization

Figure 9. Assembler Page of the Project Settings Dialog Box

7. Select the Code Generation page.
8. Select the Limit Optimizations for Easier Debug check box.



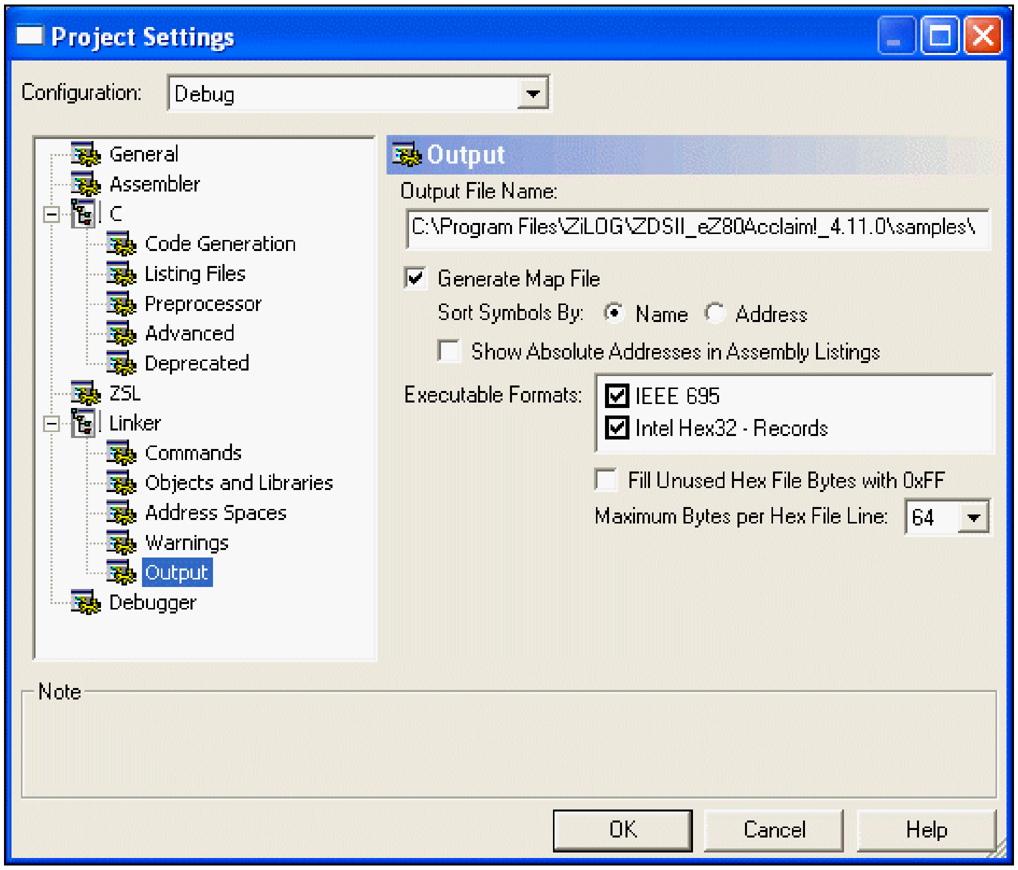
Configuration: Debug

Optimize For: Size

☑ Limit Optimizations for Easier Debugging

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

9. Select the Output page.
10. Make certain that both the IEEE 695 and Intel Hex32 - Records check boxes are selected.



Configuration: Debug

Output

Output File Name: C:\Program Files\ZiLOG\ZDSII_eZ80Acclaim!_4.11.0\samples\

[✓] 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 11. Output Page of the Project Settings Dialog Box

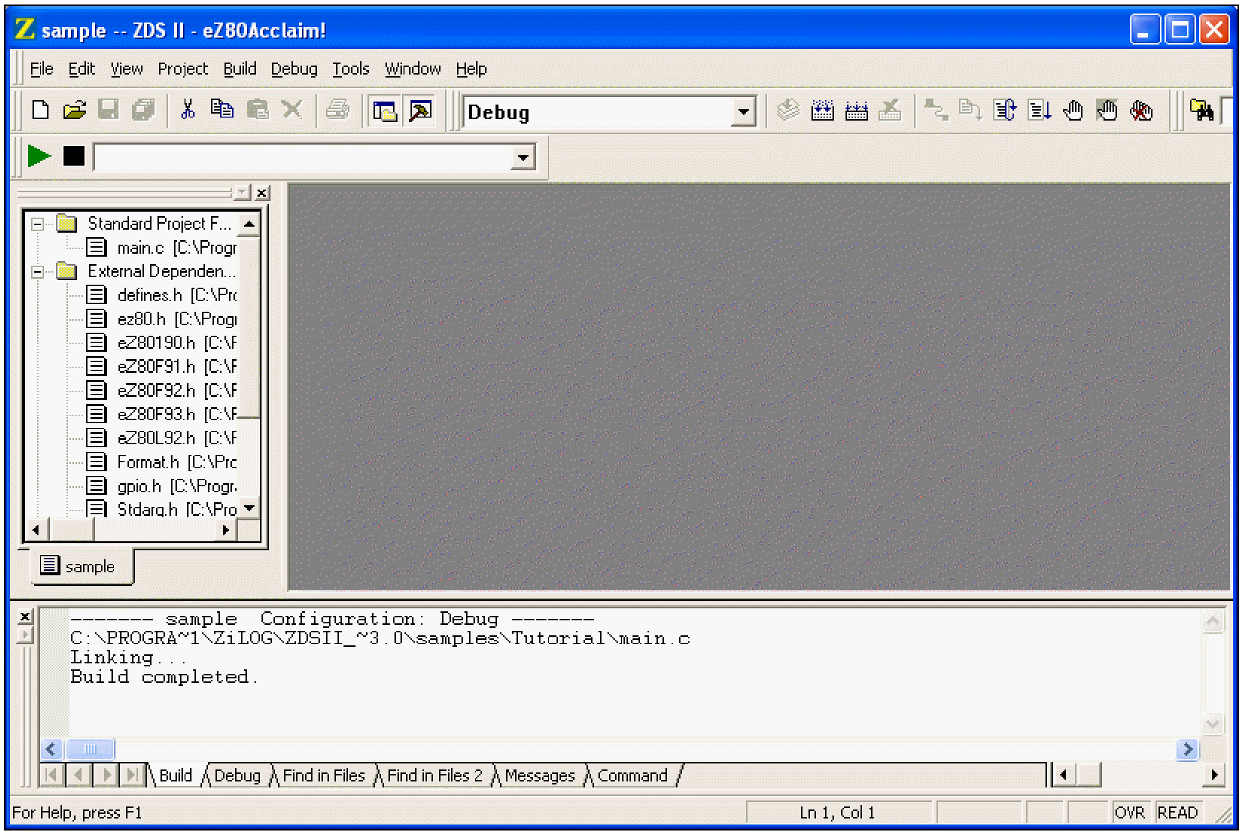
11. Click **OK** to save all the settings on the Project Settings dialog box.

    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 is as follows: "`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. Watch the compilation process in the Build Output window.



```
-------- sample  Configuration: Debug --------
C:\PROGRA~1\ZiLOG\ZDSII_~3.0\samples\Tutorial\main.c
Linking...
Build completed.
```

Figure 12. Build Output Window

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.

Save the Project

You need to save your project. From the File menu, select **Save Project**.

USING NON-SIMULATOR DEBUG TOOLS

ZDS supports the use of a number of target communication debug tools as well as an instruction set simulator. At a given time, one of the available debug tools can be configured to be used with a project.

ZDS for the eZ80Acclaim! supports the following non-Simulator debug tools:

- USB Smart Cable
- Ethernet Smart Cable
- Serial Smart Cable
- ZPAK II

Use the following procedure to configure the project to use various supported debug tools:

1. Create a new project or open an existing project.
2. From the Project menu, select **Settings**.

   The Project Settings dialog box is displayed.
3. Select the Debugger page.
4. In the Debug Tool area, select the desired debug tool from the Current drop-down list box.
5. Click **Setup** in the Debug Tool area.

   Refer to “Debug Tool” on page 102 for details about configuring the debug tool.
6. Click **OK** to accept any changes to debug tool settings.
7. Click **OK** to close and save the settings for the Project Settings dialog box.

Using the Integrated Development Environment

The following sections discuss how to use the integrated development environment (IDE):

- "Toolbars" on page 16
- "Windows" on page 29
- "Menu Bar" on page 37
- "Shortcut Keys" on page 131

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

After the discussion of the toolbars and windows, this chapter discusses the menu bar 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.

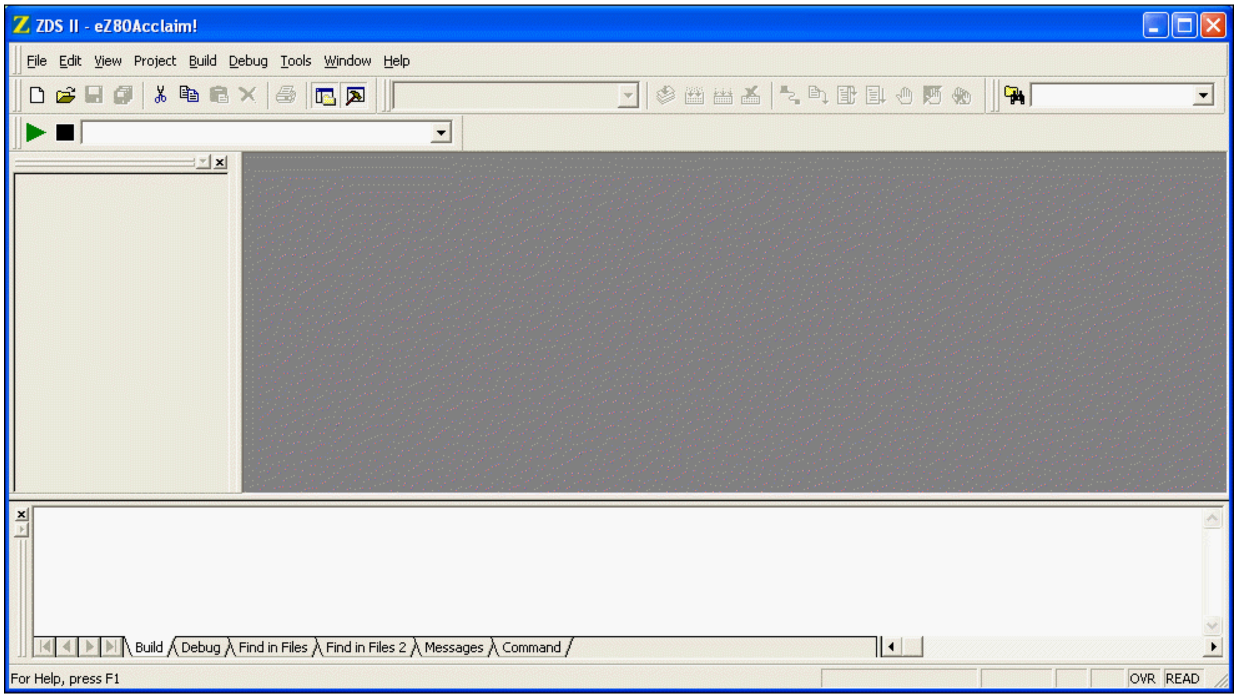

ZDS II - eZ80Acclaim!
File Edit View Project Build Debug Tools Window Help
Build Debug Find in Files Find in Files 2 Messages Command
For Help, press F1
OVR READ

Figure 13. eZ80Acclaim! Integrated Development Environment (IDE) Window

For a table of all the shortcuts used in the eZ80Acclaim! developer's environment, see "Shortcut Keys" on page 131.

TOOLBARS

The toolbars give you quick access to most features of the eZ80Acclaim! developer's environment. You can use these buttons to perform any task.

NOTE: There are cue cards for the toolbars. As you move the mouse pointer across the toolbars, the main function of each button is displayed. Also, you can drag and move the toolbars to different areas on the screen.

The following toolbars are available:

- "File Toolbar" on page 16
- "Build Toolbar" on page 18
- "Find Toolbar" on page 21
- "Command Processor Toolbar" on page 22
- "Bookmarks Toolbar" on page 22
- "Debug Toolbar" on page 23
- "Debug Windows Toolbar" on page 27

NOTE: For more information on debugging, see "Using the Debugger" on page 290.

File Toolbar

The File toolbar allows you to perform basic functions with your files using the following buttons:

- "New Button" on page 17
- "Open Button" on page 17
- "Save Button" on page 17
- "Save All Button" on page 17
- "Cut Button" on page 17
- "Copy Button" on page 17
- "Paste Button" on page 17
- "Delete Button" on page 17
- "Print Button" on page 17
- "Workspace Window Button" on page 17
- "Output Window Button" on page 17

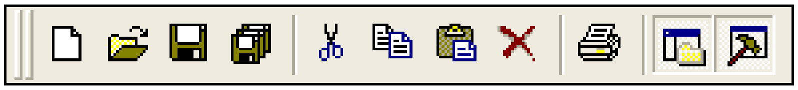


Figure 14. File Toolbar

New Button

The New button creates a new file.

Open Button

The Open button allows you to open an existing file.

Save Button

The Save button saves the active file.

Save All Button

The Save All button saves all open files and the currently loaded project.

Cut Button

The Cut button deletes selected text from the active file and puts it on the Windows clipboard.

Copy Button

The Copy button copies selected text from the active file and puts it on the Windows clipboard.

Paste Button

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

Delete Button

The Delete button deletes selected text from the active file.

Print Button

The Print button prints the active file.

Workspace Window Button

The Workspace Window button shows or hides the Project Workspace window.

Output Window Button

The Output Window button shows or hides the Output window.

Build Toolbar

The Build toolbar allows you to build your project, set breakpoints, and select a project configuration with the following controls and buttons:

- "Select Build Configuration List Box" on page 18
- "Compile/Assemble File Button" on page 18
- "Build Button" on page 18
- "Rebuild All Button" on page 18
- "Stop Build Button" on page 18
- "Connect to Target Button" on page 19
- "Download Code Button" on page 19
- "Reset Button" on page 20
- "Go Button" on page 20
- "Insert/Remove Breakpoint Button" on page 21
- "Enable/Disable Breakpoint Button" on page 21
- "Remove All Breakpoints Button" on page 21

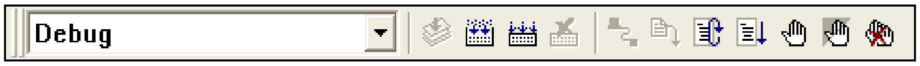


Figure 15. Build Toolbar

Select Build Configuration List Box

The Select Build Configuration drop-down list box lets you activate the build configuration for your project. See "Set Active Configuration" on page 108 for more information.

Compile/Assemble File Button

The Compile/Assemble File button compiles or assembles the active source file.

Build Button

The 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

The Rebuild All button rebuilds all files and links the project.

Stop Build Button

The Stop Build button stops a build in progress.

Connect to Target Button

The Connect to Target button starts a debug session using the following process:

1. Initializes the communication to the target hardware.
2. Resets the device.
3. Configures the device using the settings in the Configure Target dialog box.
4. Configures and executes the debugger options selected in the Debugger tab of the Options dialog box. The following options are ignored if selected:
   – Reset to Symbol 'main' (Where Applicable) check box
   – Verify File Downloads—Read After Write check box
   – Verify File Downloads—Upon Completion check box

This button does not download the software. 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. This button is available only when not in Debug mode.

For the Serial Smart Cable, ZDS II performs an external target reset and reconfigures PC and SPL as specified in the Configure Target dialog box.

Download Code Button

The 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. Starting in version 4.10.0, the Download Code button can also program Flash memory. A page erase is done instead of a mass erase for both internal and external Flash memory. Use this button anytime during a debug session. This button is not enabled when the target is the simulator.

NOTE: The current code on the target is overwritten.

If ZDS II is not in Debug mode when the Download Code button is clicked, the following process is executed:

1. Initializes the communication to the target hardware.
2. Resets the device with a hardware reset by driving ZDI pin #2 low.
3. Configures the device using the settings in the Configure Target dialog box.
4. Downloads the program.
5. Issues a software reset through the ZDI serial interface.
6. Configures and executes the debugger options selected in the Debugger tab of the Options dialog box. If it is a C project, ZDS II resets to the main function if it is found.

If ZDS II is already in Debug mode when the Download Code button is clicked, the following process is executed:

1. Resets the device using a software reset.
2. Downloads the program.

You might need to reset the device before execution because the program counter might have been changed after the download.

Reset Button

Click the Reset button in the Build or Debug toolbar to reset the program counter to the beginning of the program.

If ZDS II is not in Debug mode, the Reset button starts a debug session using the following process:

1. Initializes the communication to the target hardware.
2. Resets the device.
3. Configures the device using the settings in the Configure Target dialog box.
4. Downloads the program.
5. Configures and executes the debugger options selected in the Debugger tab of the Options dialog box. If it is a C project, ZDS II resets to the main function if it is found.

If ZDS II is already in Debug mode, the Reset button uses the following process:

1. ZDS II performs a soft reset in which just PC and SPL are reconfigured as specified in the Configure Target dialog box.
2. Configures the device using the settings in the Configure Target dialog box.
3. If files have been modified, ZDS II asks, “Would you like to rebuild the project?” before downloading the modified program. If there has been no file modification, the code is not reloaded.

The Serial Smart Cable performs an external target reset.

Go Button

Click the Go button to execute project code from the current program counter.

If not in Debug mode when the Go button is clicked, the following process is executed:

1. Initializes the communication to the target hardware.
2. Resets the device.
3. Configures the device using the settings in the Configure Target dialog box.
4. Downloads the program.

5. Configures and executes the debugger options selected in the Debugger tab of the Options dialog box. If it is a C project, ZDS II resets to the main function if it is found.
6. Executes the program from the reset location.

Insert/Remove Breakpoint Button

The 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 on breakpoints, see “Using Breakpoints” on page 306.

Enable/Disable Breakpoint Button

The 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 on breakpoints, see “Using Breakpoints” on page 306.

Remove All Breakpoints Button

The Remove All Breakpoints button deletes all breakpoints in the currently loaded project. To deactivate breakpoints in your program, use the Disable All Breakpoints button.

Find Toolbar

The Find toolbar provides access to text search functions with the following controls:

- “Find in Files Button” on page 21
- “Find Field” on page 21

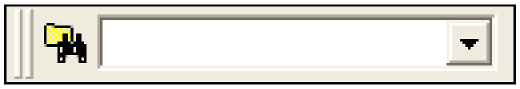


Figure 16. Find Toolbar

Find in Files Button

This 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, type 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 allows you to execute IDE and debugger commands with the following controls:

- "Run Command Button" on page 22
- "Stop Command Button" on page 22
- "Command Field" on page 22

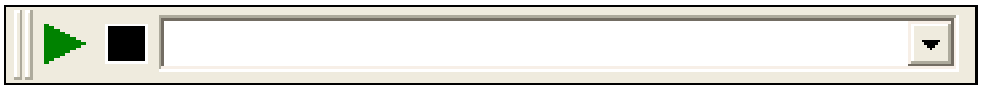


Figure 17. Command Processor Toolbar

See "Supported Script File Commands" on page 392 for a list of supported commands.

Run Command Button

The 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

The Stop Command button stops any currently running commands.

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, do the following:

1. Select Customize from the Tools menu.
2. Click 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 with the project settings.

Bookmarks Toolbar

The Bookmarks toolbar allows you to to set, remove, and find bookmarks with the following buttons:

- "Set Bookmark Button" on page 23
- "Next Bookmark Button" on page 23

- “Previous Bookmark Button” on page 23
- “Delete Bookmarks Button” on page 23

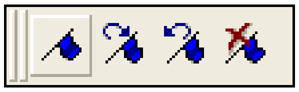


Figure 18. Bookmarks Toolbar

NOTE: This toolbar is not displayed in the default IDE window.

Set Bookmark Button

Click the Set Bookmark button to insert a bookmark in the active file for the line where your cursor is located.

Next Bookmark Button

Click the Next Bookmark button to position the cursor at the line where the next bookmark in the active file is located.

Previous Bookmark Button

Click the Previous Bookmark button to position the cursor at the line where the next bookmark in the active file is located.

Delete Bookmarks Button

Click the Delete Bookmarks button to remove all of the bookmarks in the currently loaded project.

Debug Toolbar

The Debug toolbar allows you to perform debugger functions with the following buttons:

- “Download Code Button” on page 24
- “Verify Download Button” on page 25
- “Reset Button” on page 25
- “Stop Debugging Button” on page 25
- “Go Button” on page 25
- “Run to Cursor Button” on page 26
- “Break Button” on page 26
- “Step Into Button” on page 26
- “Step Over Button” on page 26

- "Step Out Button" on page 26
- "Set Next Instruction" on page 114
- "Insert/Remove Breakpoint Button" on page 26
- "Enable/Disable Breakpoint Button" on page 27
- "Disable All Breakpoints Button" on page 27
- "Remove All Breakpoints Button" on page 27

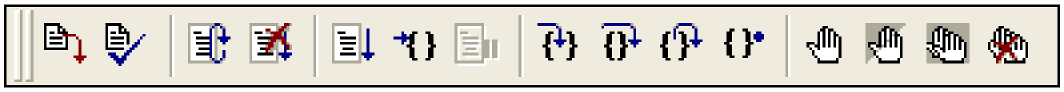


Figure 19. Debug Toolbar

Download Code Button

The 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. Starting in version 4.10.0, the Download Code button can also program Flash memory. A page erase is done instead of a mass erase for both internal and external Flash memory. Use this button anytime during a debug session. This button is not enabled when the target is the simulator.

NOTE: The current code on the target is overwritten.

If ZDS II is not in Debug mode when the Download Code button is clicked, the following process is executed:

1. Initializes the communication to the target hardware.
2. Resets the device with a hardware reset by driving ZDI pin #2 low.
3. Configures the device using the settings in the Configure Target dialog box.
4. Downloads the program.
5. Issues a software reset through the ZDI serial interface.
6. Configures and executes the debugger options selected in the Debugger tab of the Options dialog box. If it is a C project, ZDS II resets to the main function if it is found.

If ZDS II is already in Debug mode when the Download Code button is clicked, the following process is executed:

1. Resets the device using a software reset.
2. Downloads the program.

You might need to reset the device before execution because the program counter might have been changed after the download.

Verify Download Button

The Verify Download button determines download correctness by comparing the executable file contents to target memory.

Reset Button

Click the Reset button to reset the program counter to the beginning of the program.

If ZDS II is not in Debug mode, the Reset button starts a debug session using the following process:

1. Initializes the communication to the target hardware.
2. Resets the device.
3. Configures the device using the settings in the Configure Target dialog box.
4. Downloads the program.
5. Configures and executes the debugger options selected in the Debugger tab of the Options dialog box. If it is a C project, ZDS II resets to the main function if it is found.

If ZDS II is already in Debug mode, the Reset button uses the following process:

1. ZDS II performs a soft reset in which just PC and SPL are reconfigured as specified in the Configure Target dialog box.
2. Configures the device using the settings in the Configure Target dialog box.
3. If files have been modified, ZDS II asks, “Would you like to rebuild the project?” before downloading the modified program. If there has been no file modification, the code is not reloaded.

The Serial Smart Cable performs an external target reset.

Stop Debugging Button

The Stop Debugging button ends the current debug session.

To stop program execution, click the Break button.

Go Button

Click the Go button to execute project code from the current program counter.

If not in Debug mode when the Go button is clicked, the following process is executed:

1. Initializes the communication to the target hardware.
2. Resets the device.
3. Configures the device using the settings in the Configure Target dialog box.

4. Downloads the program.
5. Configures and executes the debugger options selected in the Debugger tab of the Options dialog box. If it is a C project, ZDS II resets to the main function if it is found.
6. Executes the program from the reset location.

Run to Cursor Button

The 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

The Break button stops program execution at the current program counter.

Step Into Button

The 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

The 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

The 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

The 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

The 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 on breakpoints, see “Using Breakpoints” on page 306.

Enable/Disable Breakpoint Button

The 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 on breakpoints, see “Using Breakpoints” on page 306.

Disable All Breakpoints Button

The Disable All Breakpoints button deactivates all breakpoints in the currently loaded project. To delete breakpoints from your program, use the Remove All Breakpoints button.

Remove All Breakpoints Button

The Remove All Breakpoints button deletes all breakpoints in the currently loaded project. To deactivate breakpoints in your program, use the Disable All Breakpoints button.

Debug Windows Toolbar

The Debug Windows toolbar allows you to display the Debug windows with the following buttons:

- “Registers Window Button” on page 27
- “Special Function Registers Window Button” on page 28
- “Clock Window Button” on page 28
- “Memory Window Button” on page 28
- “Watch Window Button” on page 28
- “Locals Window Button” on page 28
- “Call Stack Window Button” on page 28
- “Symbols Window Button” on page 28
- “Disassembly Window Button” on page 28
- “Simulated UART Output Window Button” on page 28

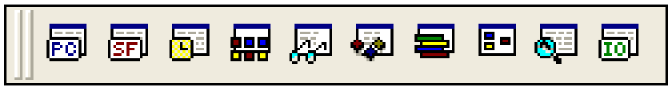


Figure 20. Debug Windows Toolbar

Registers Window Button

The Registers Window button displays or hides the Registers window. This window is described in “Registers Window” on page 292.

Special Function Registers Window Button

The Special Function Registers Window button opens one of ten Special Function Registers windows. This window is described in "Special Function Registers Window" on page 293.

Clock Window Button

The Clock Window button displays or hides the Clock window. This window is described in "Clock Window" on page 294.

Memory Window Button

The Memory Window button opens one of ten Memory windows. This window is described in "Memory Window" on page 295.

Watch Window Button

The Watch Window button displays or hides the Watch window. This window is described in "Watch Window" on page 300.

Locals Window Button

The Locals Window button displays or hides the Locals window. This window is described in "Locals Window" on page 302.

Call Stack Window Button

The Call Stack Window button displays or hides the Call Stack window. This window is described in "Call Stack Window" on page 303.

Symbols Window Button

The Symbols Window button displays or hides the Symbols window. This window is described in "Symbols Window" on page 304.

Disassembly Window Button

The Disassembly Window button displays or hides the Disassembly window. This window is described in "Disassembly Window" on page 305.

Simulated UART Output Window Button

The Simulated UART Output Window button displays or hides the Simulated UART Output window. This window is described in "Simulated UART Output Window" on page 306.

WINDOWS

The following ZDS II windows allow you to see various aspects of the tools while working with your project:

- "Project Workspace Window" on page 29
- "Edit Window" on page 30
- "Output Windows" on page 34
- "Debug Windows" on page 292

Project Workspace Window

The Project Workspace window on the left side of the developer's environment allows you to view your project files.

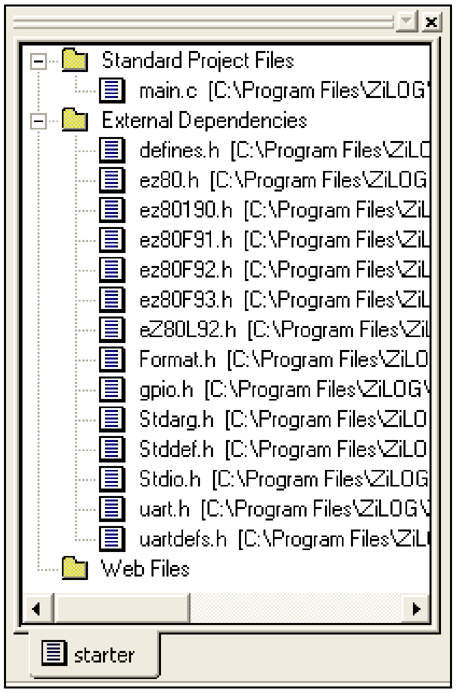

Standard Project Files
main.c [C:\Program Files\ZiLOG\
External Dependencies
defines.h [C:\Program Files\ZiLO
ez80.h [C:\Program Files\ZiLOG
ez80190.h [C:\Program Files\ZiL
ez80F91.h [C:\Program Files\ZiL
ez80F92.h [C:\Program Files\ZiL
ez80F93.h [C:\Program Files\ZiL
eZ80L92.h [C:\Program Files\Zil
Format.h [C:\Program Files\ZiLO
gpio.h [C:\Program Files\ZiLOG\
Stdarg.h [C:\Program Files\ZiLO
Stddef.h [C:\Program Files\ZiLO
Stdio.h [C:\Program Files\ZiLOG
uart.h [C:\Program Files\ZiLOG\
uartdefs.h [C:\Program Files\ZiL
Web Files
starter

Figure 21. Project Workspace Window for Standard Projects

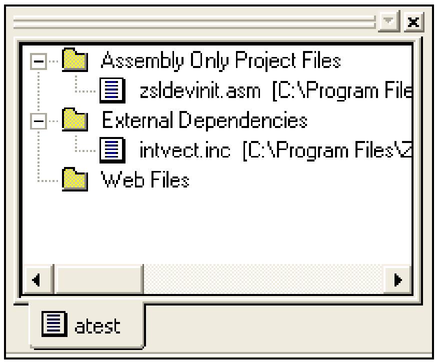

Assembly Only Project Files
zsldevinit.asm [C:\Program File
External Dependencies
intvect.inc [C:\Program Files\Z
Web Files
atest

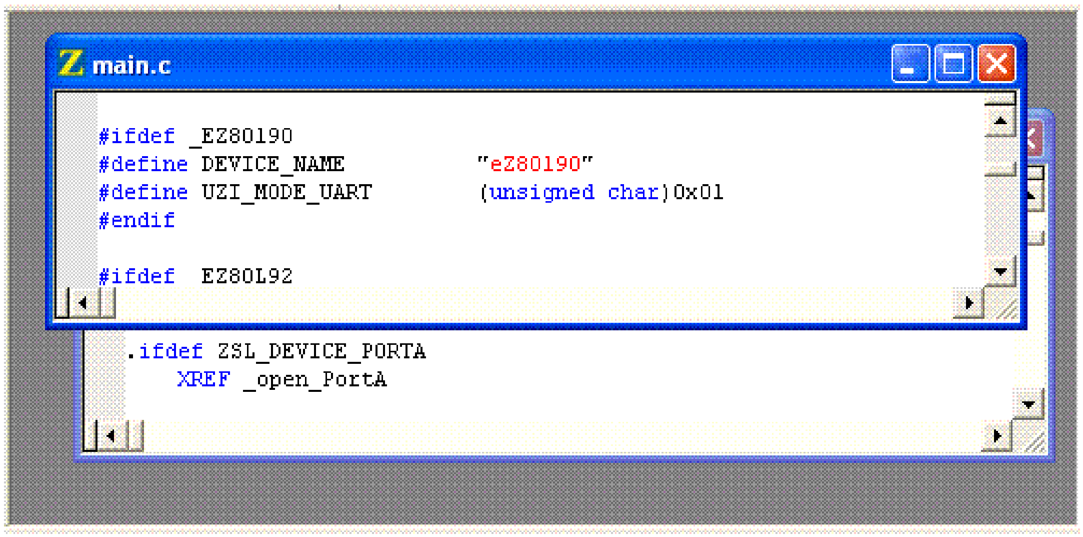
Figure 22. Project Workspace Window for Assembly Only Projects

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 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

The Edit window on the right side of the developer's environment allows you to edit the files in your project.



```
#ifdef _EZ80190
#define DEVICE_NAME          "eZ80190"
#define UZI_MODE_UART        (unsigned char)0x01
#endif

#ifdef  EZ80L92
```

```
.ifdef ZSL_DEVICE_PORTA
    XREF _open_PortA
```

Figure 23. Edit Window

The Edit window supports the following shortcuts:

Function	**Shortcuts**
Undo	Ctrl + Z
Redo	Ctrl + Y
Cut	Ctrl + X
Copy	Ctrl + C
Paste	Ctrl + V
Find	Ctrl + F
Repeat the previous search	F3
Go to	Ctrl + G
Go to matching { or }. 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.	Ctrl + E Ctrl +]

This section covers the following topics:

- "Using the Context Menus" on page 31
- "Using Bookmarks" on page 32

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 (see "Using Bookmarks" on page 32)

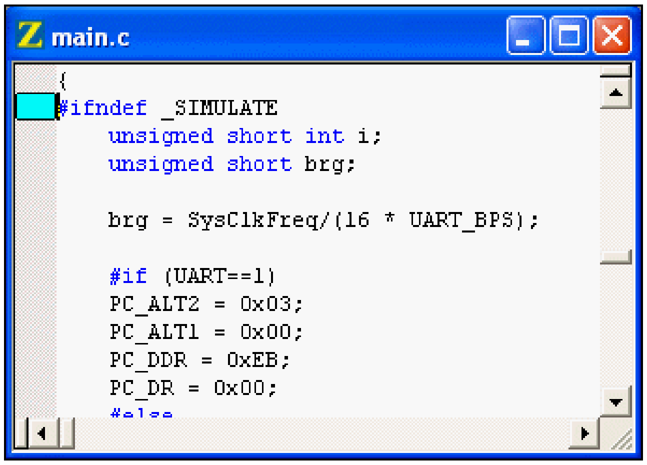
When you right-click outside of all files, the context menu allows you to do the following:

- Show or hide the Output windows, Project Workspace window, status bar, File toolbar, Build toolbar, Find toolbar, Command Processor toolbar, Debug toolbar, and Debug Windows toolbar
- Toggle Workbook Mode

  When in Workbook Mode, each open file has an associated tab along the bottom of the Edit windows area.
- 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. The cursor can be quickly positioned on a line containing bookmarks.



```
{
#ifndef _SIMULATE
    unsigned short int i;
    unsigned short brg;

    brg = SysClkFreq/(16 * UART_BPS);

    #if (UART==1)
    PC_ALT2 = 0x03;
    PC_ALT1 = 0x00;
    PC_DDR = 0xEB;
    PC_DR = 0x00;
```

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.
- Select **Toggle Bookmark** from the Edit menu.
- Type Ctrl+M.

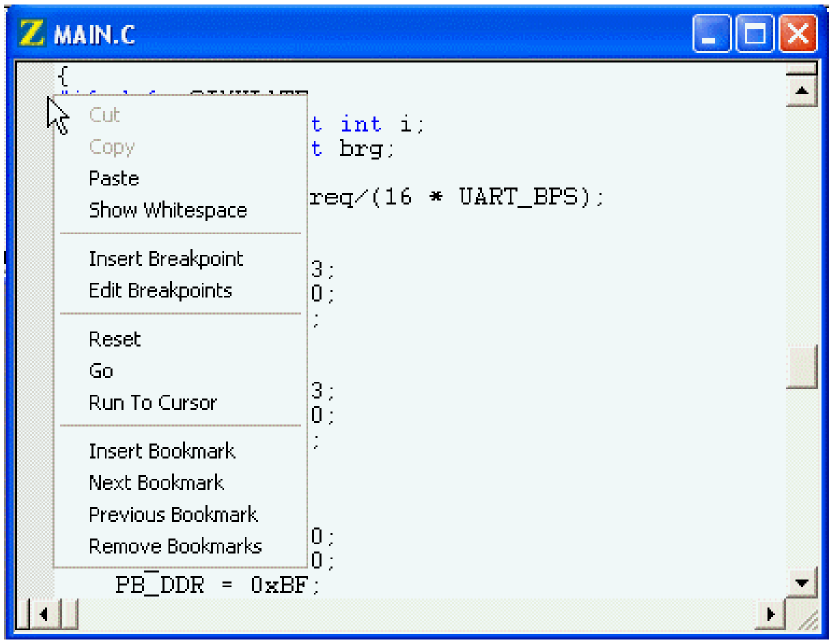


Figure 25. Inserting a Bookmark

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.
- Type 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.
- Type 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:

- “Build Output Window” on page 35

- “Debug Output Window” on page 35
- “Find in Files Output Windows” on page 35
- “Messages Output Window” on page 36
- “Command Output Window” on page 36

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

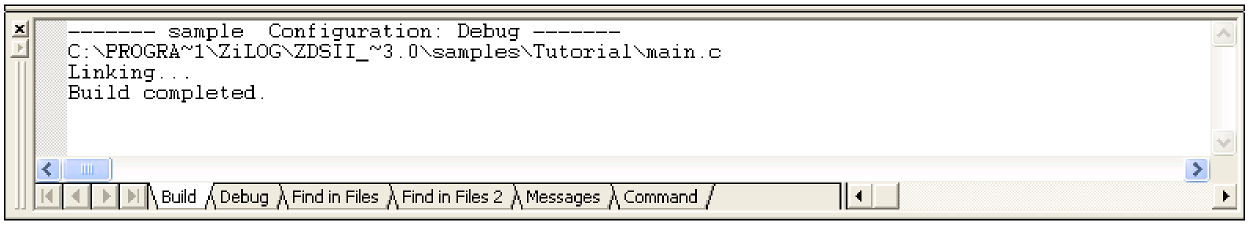
Double-click on the window's grip bar to cause it to become a floating window.

Double-click on 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

The Build Output window holds all text messages generated by the compiler, assembler, librarian, and linker, including error and warning messages.

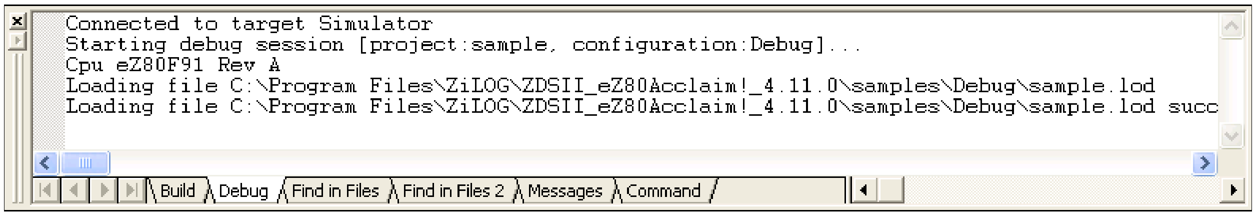


```
------- sample  Configuration: Debug -------
C:\PROGRA~1\ZiLOG\ZDSII_~3.0\samples\Tutorial\main.c
Linking...
Build completed.
```

Figure 26. Build Output Window

Debug Output Window

The Debug Output window holds all text messages generated by the debugger while you are in Debug mode.

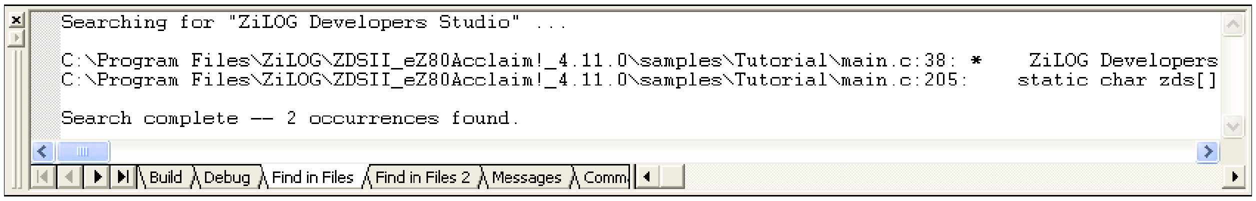


```
Connected to target Simulator
Starting debug session [project:sample, configuration:Debug]...
Cpu eZ80F91 Rev A
Loading file C:\Program Files\ZiLOG\ZDSII_eZ80Acclaim!_4.11.0\samples\Debug\sample.lod
Loading file C:\Program Files\ZiLOG\ZDSII_eZ80Acclaim!_4.11.0\samples\Debug\sample.lod succ
```

Figure 27. Debug Output Window

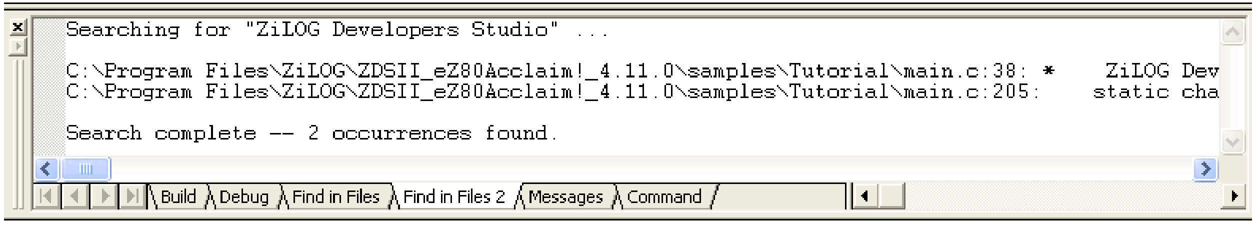
Find in Files Output Windows

The two Find in Files Output windows display the results of the Find in Files command (available from the Edit menu and the Edit toolbar). The File in Files 2 window is used when the Output to Pane 2 check box is selected in the Find in File dialog box (see “Find in Files” on page 49).



```
Searching for "ZiLOG Developers Studio" ...

C:\Program Files\ZiLOG\ZDSII_eZ80Acclaim!_4.11.0\samples\Tutorial\main.c:38: *    ZiLOG Developers
C:\Program Files\ZiLOG\ZDSII_eZ80Acclaim!_4.11.0\samples\Tutorial\main.c:205:    static char zds[]

Search complete -- 2 occurrences found.
```

Figure 28. Find in Files Output Window

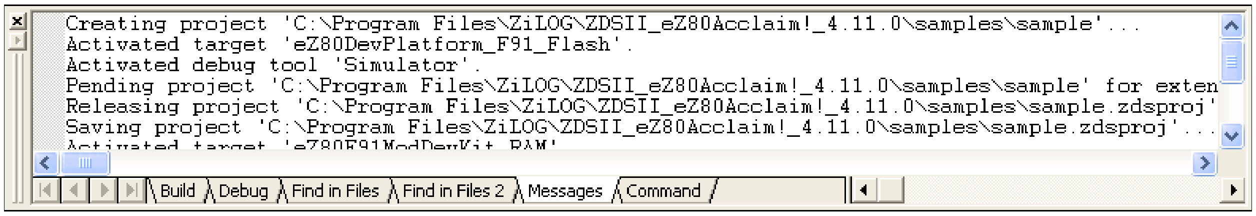


```
Searching for "ZiLOG Developers Studio" ...

C:\Program Files\ZiLOG\ZDSII_eZ80Acclaim!_4.11.0\samples\Tutorial\main.c:38: *    ZiLOG Dev
C:\Program Files\ZiLOG\ZDSII_eZ80Acclaim!_4.11.0\samples\Tutorial\main.c:205:    static cha

Search complete -- 2 occurrences found.
```

Figure 29. Find in Files 2 Output Window

Messages Output Window

The Messages Output window holds informational messages intended for the user. The Message Output window is activated (given focus) when error messages are added to the window's display. Warning and informational messages do not automatically activate the Message Output window. The Messages Output window also displays the chip revision identifier and the Smart Cable firmware version.

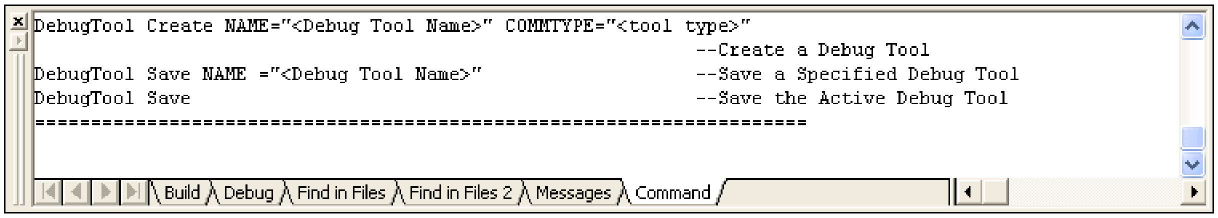


```
Creating project 'C:\Program Files\ZiLOG\ZDSII_eZ80Acclaim!_4.11.0\samples\sample'...
Activated target 'eZ80DevPlatform_F91_Flash'.
Activated debug tool 'Simulator'.
Pending project 'C:\Program Files\ZiLOG\ZDSII_eZ80Acclaim!_4.11.0\samples\sample' for exten
Releasing project 'C:\Program Files\ZiLOG\ZDSII_eZ80Acclaim!_4.11.0\samples\sample.zdsproj'
Saving project 'C:\Program Files\ZiLOG\ZDSII_eZ80Acclaim!_4.11.0\samples\sample.zdsproj'...
Activated target 'eZ80F91ModDevKit_RAM'
```

Figure 30. Messages Output Window

Command Output Window

The Command Output window holds output from the execution of commands.



```
DebugTool Create NAME="<Debug Tool Name>" COMMTYPE="<tool type>"
                                                     --Create a Debug Tool
DebugTool Save NAME ="<Debug Tool Name>"             --Save a Specified Debug Tool
DebugTool Save                                       --Save the Active Debug Tool
======================================================================
```

Figure 31. Command Output Window

MENU BAR

The menu bar lists menu items used in the eZ80Acclaim! developer's environment. Each 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. The following items are available from the menu bar:

- "File Menu" on page 37
- "Edit Menu" on page 47
- "View Menu" on page 53
- "Project Menu" on page 54
- "Build Menu" on page 107
- "Debug Menu" on page 111
- "Tools Menu" on page 114
- "Window Menu" on page 129
- "Help Menu" on page 130

File Menu

The File menu enables you to perform basic commands in the developer's environment:

- "New File" on page 38
- "Open File" on page 38
- "Close File" on page 38
- "New Project" on page 39
- "Open Project" on page 43
- "Save Project" on page 44
- "Close Project" on page 45
- "Save" on page 45

- "Save As" on page 45
- "Save All" on page 45
- "Print" on page 45
- "Print Preview" on page 46
- "Print Setup" on page 47
- "Recent Files" on page 47
- "Recent Projects" on page 47
- "Exit" on page 47

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, which allows you to open the files for your project.

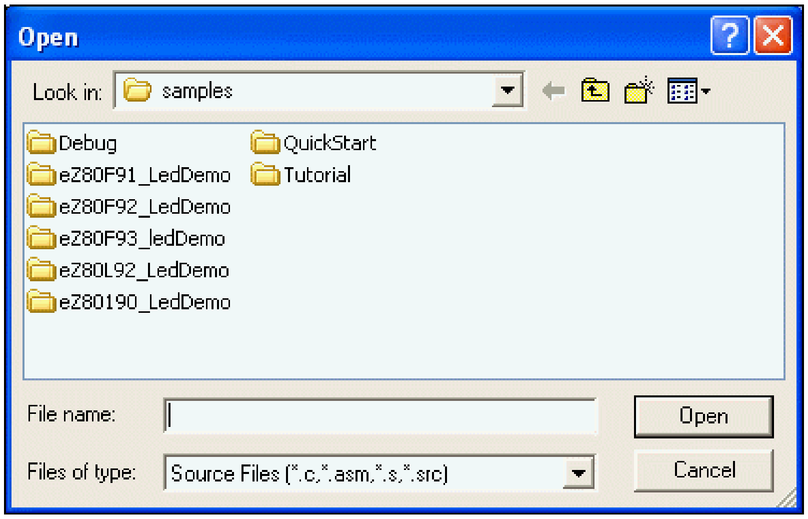


Figure 32. Open Dialog Box

NOTE: To delete a file from your project, use the Open Project dialog box. Highlight the file and press the Delete key. Answer the prompt accordingly.

Close File

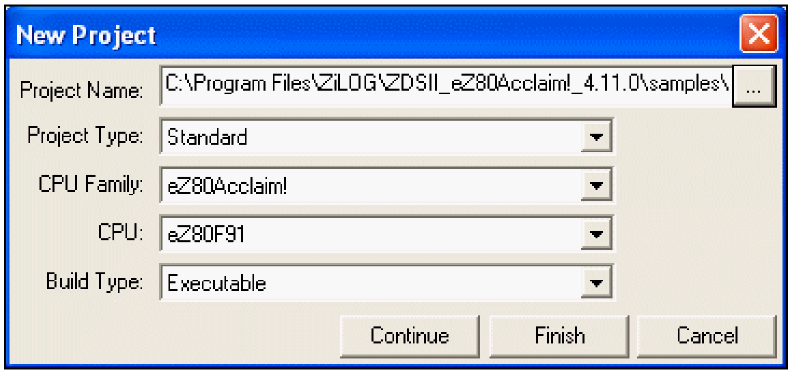
Select **Close File** from the File menu to close the selected file.

New Project

To create a new project, do the following:

1. Select **New Project** from the File menu.

   The New Project dialog box is displayed.



New Project

Project Name: C:\Program Files\ZiLOG\ZDSII_eZ80Acclaim!_4.11.0\samples\

Project Type: Standard

CPU Family: eZ80Acclaim!

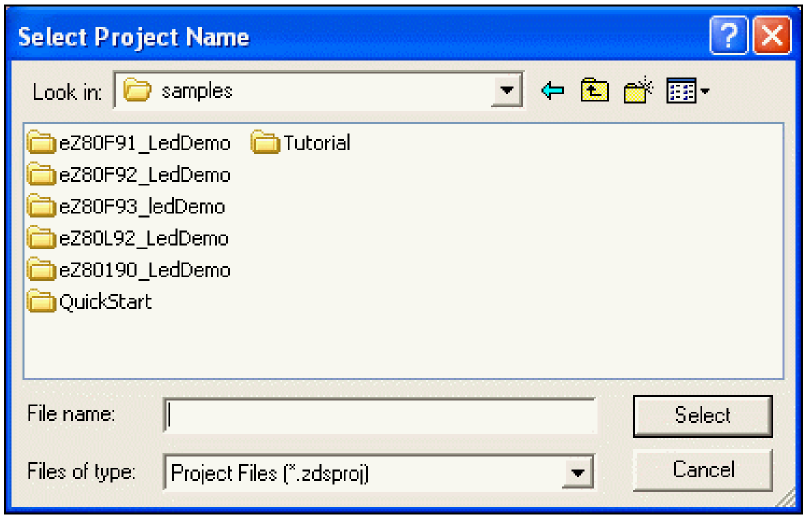
CPU: eZ80F91

Build Type: Executable

Figure 33. New Project Dialog Box

2. From the New Project dialog box, click on the Browse button (...) to navigate to the directory where you want to save your project.

   The Select Project Name dialog box is displayed.



Select Project Name

Look in: samples

eZ80F91_LedDemo
eZ80F92_LedDemo
eZ80F93_ledDemo
eZ80L92_LedDemo
eZ80190_LedDemo
QuickStart
Tutorial

File name:

Files of type: Project Files (*.zdsproj)

Figure 34. Select Project Name Dialog Box

3. Use the Look In drop-down list box to navigate to the directory where you want to save your project.
4. In the File Name field, type the name of your project.

You do not have to type the extension `.zdsproj`. The extension is added automatically.

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

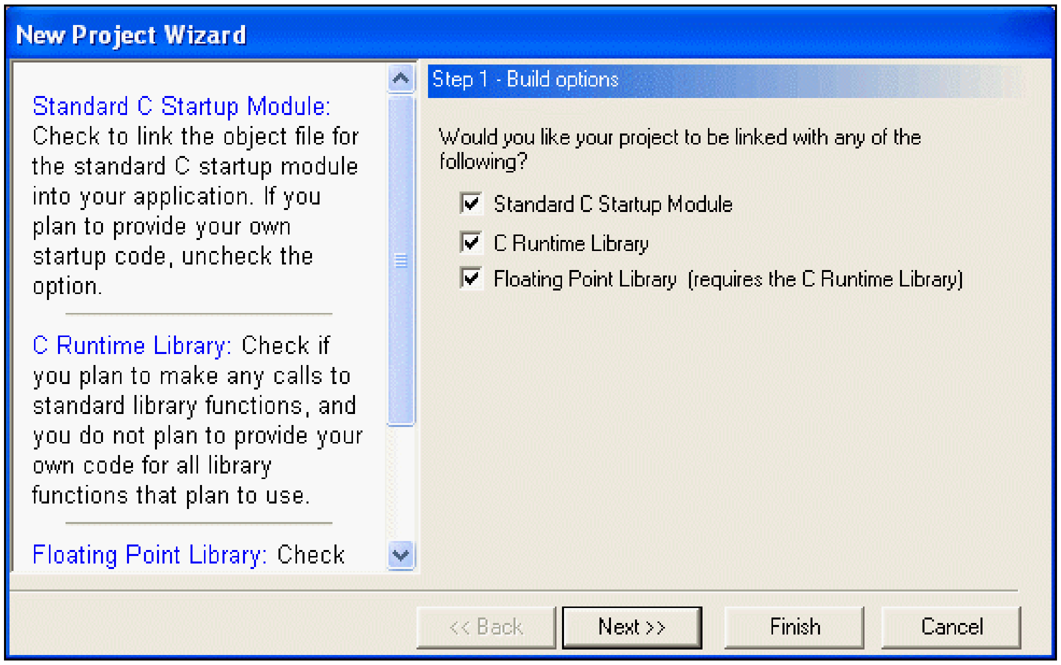
5. Click **Select** to return to the New Project dialog box.
6. In the Project Type field, select **Standard** for a project that will include C language source code. Select **Assembly Only** for a project that will include only assembly source code.
7. In the CPU Family drop-down list box, select **eZ80** or **eZ80Acclaim!**.
8. In the CPU drop-down list box, select a CPU.
9. In the Build Type drop-down list box, select **Executable** to build an application or select **Static Library** to build a static library.

   The default is **Executable**, which creates IEEE 695 (`.lod`) and Intel Hex32 (.hex) executables. For more information, see “Project Settings—Output Page” on page 92.
10. Click **Continue** to change the default project settings using the New Project Wizard.

    To accept all default settings, click **Finish**.

NOTE: For static libraries, click **Finish**.

For a Standard project, the New Project Wizard dialog box is displayed. For Assembly-Only executable projects, continue to Step 12.


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

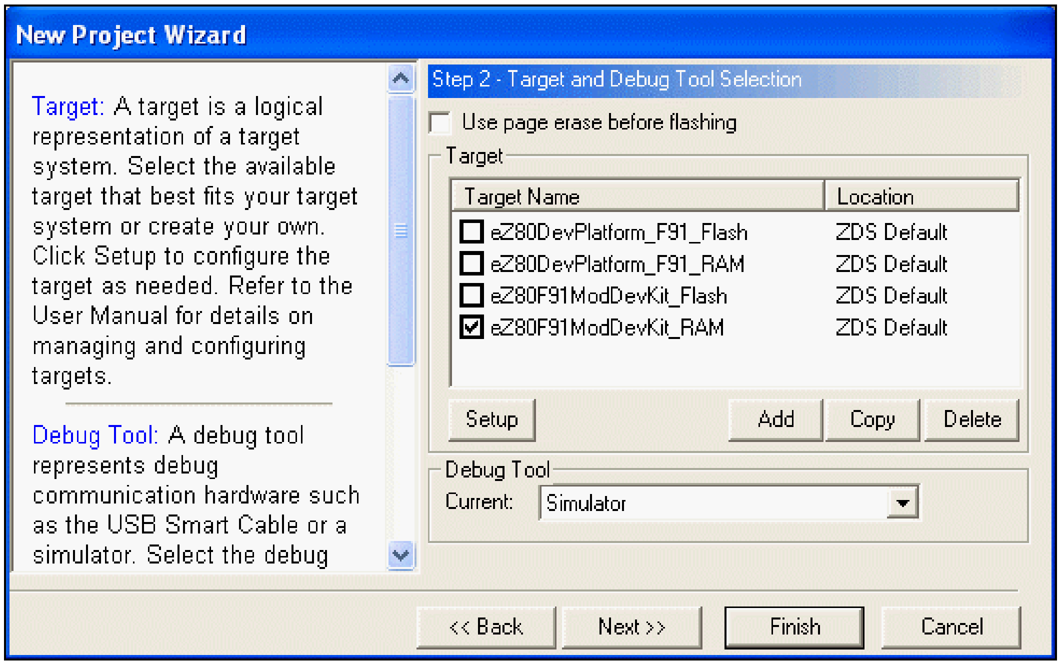
Would you like your project to be linked with any of the following?

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

Figure 35. New Project Wizard Dialog Box—Build Options Step

11. For standard projects only, select whether your project is linked with any or all of the following: standard C startup module, C run-time library, and floating-point library; then click **Next**.

    For executable projects, the Target and Debug Tool Selection step of the New Project Wizard dialog box is displayed.



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 managing and configuring targets.

Debug Tool: A debug tool represents debug communication hardware such as the USB Smart Cable or a simulator. Select the debug

☐ Use page erase before flashing

Target

Target Name	Location
☐ eZ80DevPlatform_F91_Flash	ZDS Default
☐ eZ80DevPlatform_F91_RAM	ZDS Default
☐ eZ80F91ModDevKit_Flash	ZDS Default
☑ eZ80F91ModDevKit_RAM	ZDS Default

Debug Tool
Current: Simulator

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

12. Select the Use Page Erase Before Flashing check box to configure the internal Flash memory of the target hardware to be page-erased. If this check box is not selected, the internal Flash is configured to be mass-erased.
13. Select the appropriate target from the Target list box.
14. Click **Setup** in the Target area.

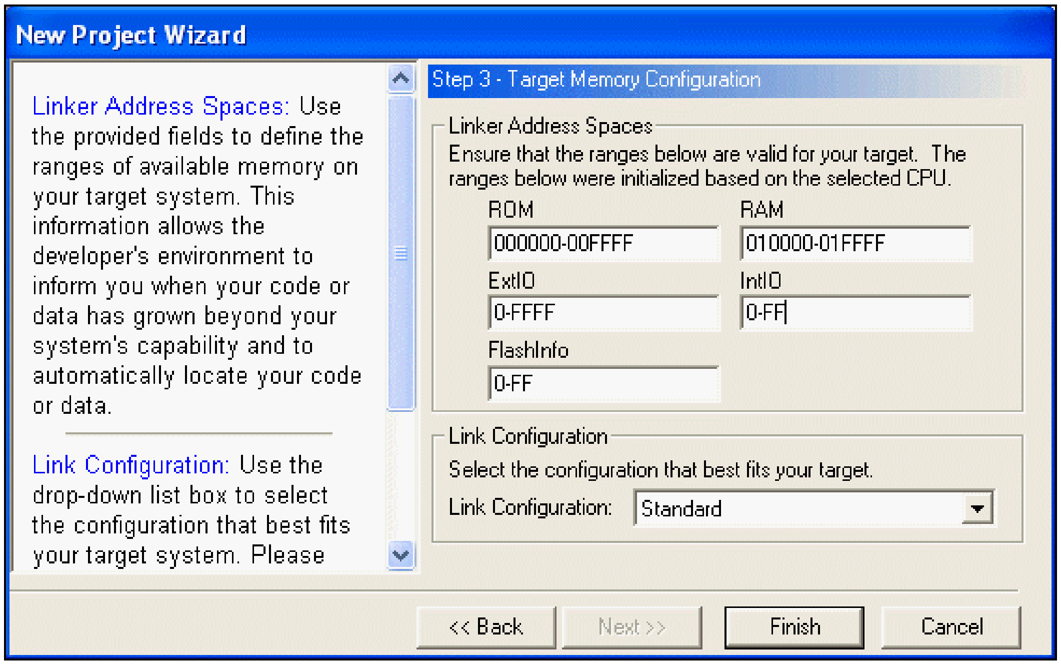
    Refer to “Setup” on page 96 for details on configuring a target.

NOTE: Click **Add** to create a new target (see “Add” on page 100) or click **Copy** to copy an existing target (see “Copy” on page 101).

15. Select the appropriate debug tool and (if you have not selected the Simulator) click **Setup** in the Debug Tool area.

    Refer to “Debug Tool” on page 102 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.



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.

Link Configuration: Use the drop-down list box to select the configuration that best fits your target system. Please

Linker Address Spaces

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

ROM: 000000-00FFFF
RAM: 010000-01FFFF
ExtIO: 0-FFFF
IntIO: 0-FF
FlashInfo: 0-FF

Link Configuration

Select the configuration that best fits your target.

Link Configuration: Standard

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

17. Enter the memory ranges appropriate for the target CPU and select the link configuration from the Link Configurations drop-down list box.
18. Select the configuration that best fits your target. The project settings are modified accordingly. ZDS II automatically generates a linker command file using the project settings. Refer to "Link Configuration" on page 77 for details about available link configurations.
19. Click **Finish**.

Open Project

To open an existing project, use the following procedure:

1. Select **Open Project** from the File menu.

   The Open Project dialog box is displayed.

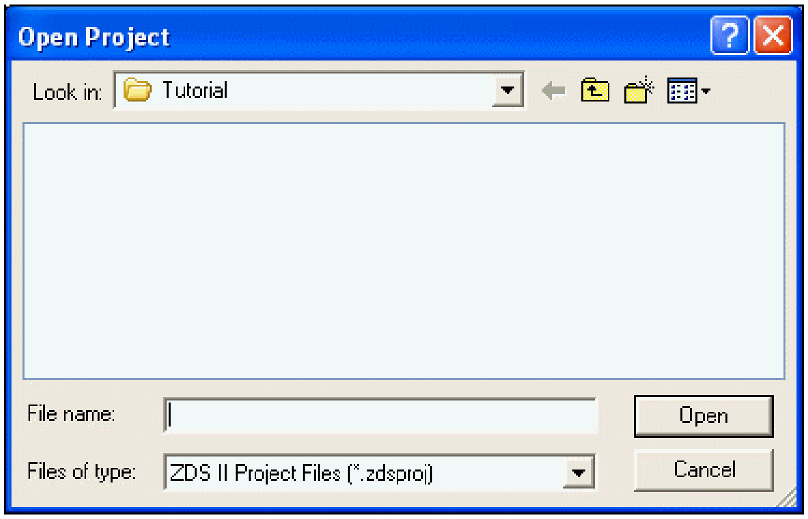


Figure 38. Open Project Dialog Box

2. Use the Look In drop-down list box to navigate to the appropriate directory where your project is located.
3. Select the project to be opened.
4. Click **Open** to open to open your project.

NOTE: To quickly open a project you were working in recently, see “Recent Projects” on page 47.

To delete a project file, use the Open Project dialog box. Highlight the file and press the Delete key. Answer the prompt accordingly.

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 check box is selected (see “Options—General Tab” on page 123), 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

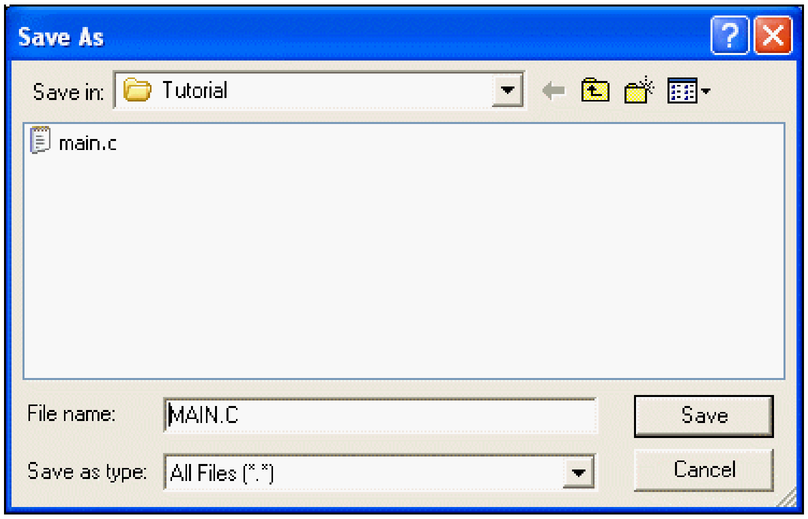
Select **Save** from the File menu to save the active file.

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.



File name: MAIN.C

Save as type: All Files (*.*)

Figure 39. Save As Dialog Box

2. Use the Save In drop-down list box 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 list box 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

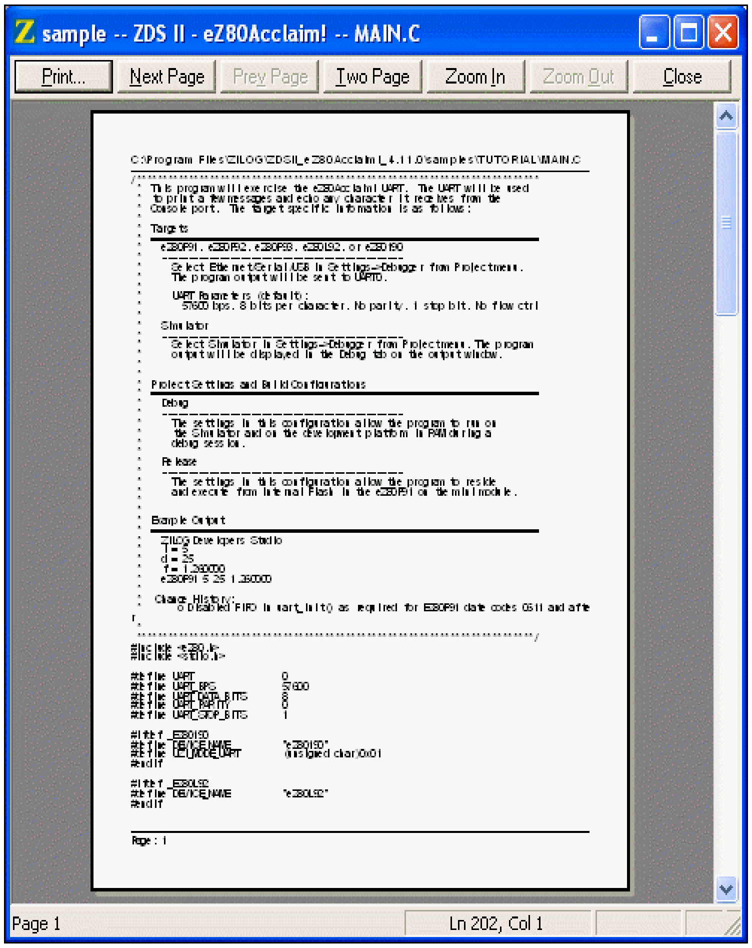
Select **Print** from the File menu to print the active file.

Print Preview

Select **Print Preview** from the File menu to display the file you want to print in Preview mode in a new window.

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 shown in Print Preview in a new window. As shown in the following figure, `main.c` is in Print Preview mode.



```
C:\Program Files\ZILOG\ZDSII_eZ80Acclaim!_4.11.0\samples\TUTORIAL\MAIN.C
/*************************************************************************
 *  This program will exercise the eZ80Acclaim! UART.  The UART will be used
 *   to print a few messages and echo any character it receives from the
 *  Console port.  The target specific information is as follows:
 *
 *  Targets
 *  __________________________________________________________________
 *    eZ80F91, eZ80F92, eZ80F93, eZ80L92, or eZ80190
 *    ------------------------------------
 *      Select Ethernet/Serial/USB in Settings->Debugger from Project menu.
 *      The program output will be sent to UART0.
 *
 *      UART Parameters (default):
 *        57600 bps, 8 bits per character, No parity, 1 stop bit, No flow ctrl
 *
 *    Simulator
 *    ------------------------------------
 *      Select Simulator in Settings->Debugger from Project menu. The program
 *      output will be displayed in the Debug tab on the output window.
 *
 *
 *  Project Settings and Build Configurations
 *  __________________________________________________________________
 *    Debug
 *    ------------------------------------
 *      The settings in this configuration allow the program to run on
 *       the Simulator and on the development platform in RAM during a
 *      debug session.
 *
 *    Release
 *    ------------------------------------
 *      The settings in this configuration allow the program to reside
 *      and execute from internal Flash in the eZ80F91 on the mini module.
 *
 *
 *  Example Output
 *  __________________________________________________________________
 *    ZiLOG Developers Studio
 *     i = 5
 *    d = 25
 *     f = 1.260000
 *    eZ80F91 5 25 1.260000
 *
 *   Change History:
 *       o Disabled FIFO in uart_init() as required for EZ80F91 date codes 0611 and afte
r.
 *
 *************************************************************************/
#include <ez80.h>
#include <stdio.h>

#define UART              0
#define UART_BPS          57600
#define UART_DATA_BITS    8
#define UART_PARITY       0
#define UART_STOP_BITS    1

#ifdef _EZ80190
#define DEVICE_NAME       "eZ80190"
#define UZI_MODE_UART     (unsigned char)0x01
#endif

#ifdef _EZ80L92
#define DEVICE_NAME       "eZ80L92"
#endif

Page : 1
```

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 determine the printer's setup 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. The following options are available:

- "Undo" on page 48
- "Redo" on page 48
- "Cut" on page 48
- "Copy" on page 48
- "Paste" on page 48
- "Delete" on page 48
- "Select All" on page 48
- "Show Whitespaces" on page 48
- "Find" on page 48
- "Find Again" on page 49
- "Find in Files" on page 49
- "Replace" on page 50
- "Go to Line" on page 51
- "Manage Breakpoints" on page 52
- "Toggle Bookmark" on page 53
- "Next Bookmark" on page 53

- “Previous Bookmark” on page 53
- “Remove All Bookmarks” on page 53

Undo

Select **Undo** from the Edit menu to undo the last edit made to the active file.

Redo

Select **Redo** from the Edit menu to redo the last edit made to the active file.

Cut

Select **Cut** from the Edit menu to delete selected text from the active file and put it on the Windows clipboard.

Copy

Select **Copy** from the Edit menu to copy selected text from the active file and put it on the Windows clipboard.

Paste

Select **Paste** from the Edit menu to paste the current contents of the clipboard into the active file at the current cursor position.

Delete

Select **Delete** from the Edit menu to delete selected text from the active file.

Select All

Select **Select All** from the Edit menu to highlight all text in the active file.

Show Whitespaces

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

Find

To find text in the active file, use the following procedure:

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

   The Find dialog box is displayed.

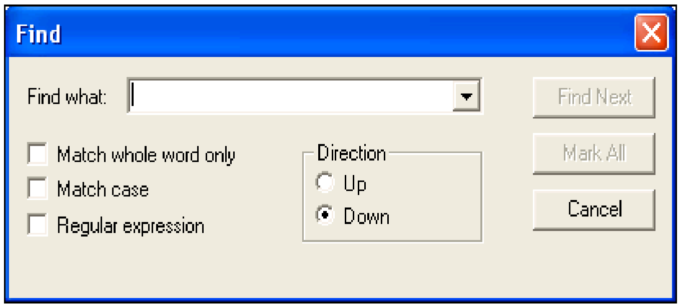


Figure 41. Find Dialog Box

2. Enter the text to search for in the Find What field or select a recent entry from the Find What drop-down list box. (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 check box if you want to ignore the search text when it occurs as part of longer words.
4. Select the Match Case check box if you want the search to be case sensitive.
5. Select the Regular Expression check box 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.

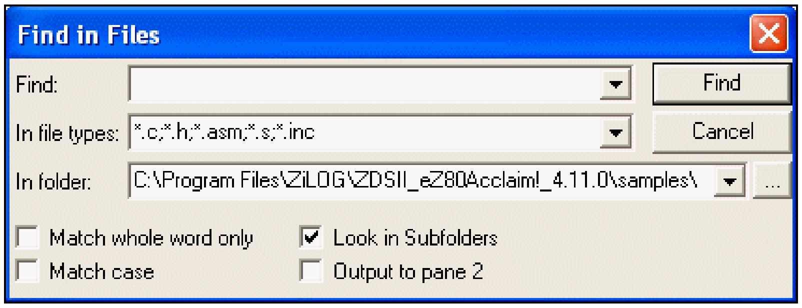
Find in Files

NOTE: This function searches the contents of the files on disk; therefore, unsaved data in open files are not searched.

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

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

   The Find in Files dialog box is displayed.



In file types: *.c;*.h;*.asm;*.s;*.inc
In folder: C:\Program Files\ZiLOG\ZDSII_eZ80Acclaim!_4.11.0\samples\
[✓ Look in Subfolders]

Figure 42. Find in Files Dialog Box

2. Enter the text to search for in the Find field or select a recent entry from the Find drop-down list box. (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 list box. Separate multiple file types with semicolons.
4. Use the Browse button (...) or the In Folder drop-down list box to select where the files are located that you want to search.
5. Select the Match Whole Word Only check box if you want to ignore the search text when it occurs as part of longer words.
6. Select the Match Case check box if you want the search to be case sensitive.
7. Select the Look in Subfolders check box if you want to search within subfolders.
8. Select the Output to Pane 2 check box 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, use the following procedure:

1. Select **Replace** from the Edit menu.

   The Replace dialog box is displayed.

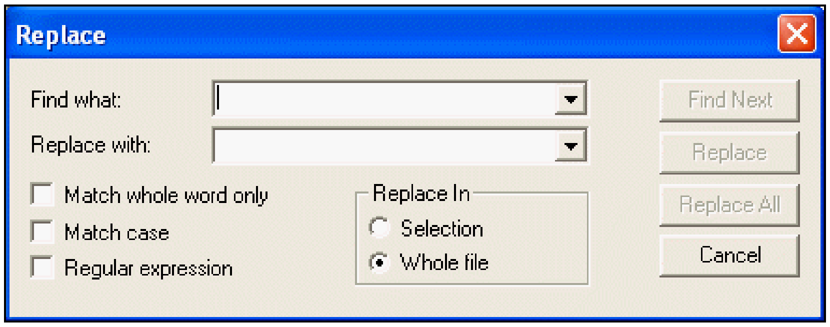


Figure 43. Replace Dialog Box

2. Enter the text to search for in the Find What field or select a recent entry from the Find What drop-down list box. (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 list box.
4. Select the Match Whole Word Only check box if you want to ignore the search text when it occurs as part of longer words.
5. Select the Match Case check box if you want the search to be case sensitive.
6. Select the Regular Expression check box if you want to use regular expressions.
7. Select whether you want the text to be replaced in text currently selected or in the whole file.
8. Click **Find Next** to jump to the next occurrence of the search text and then 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.

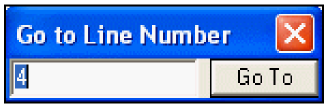
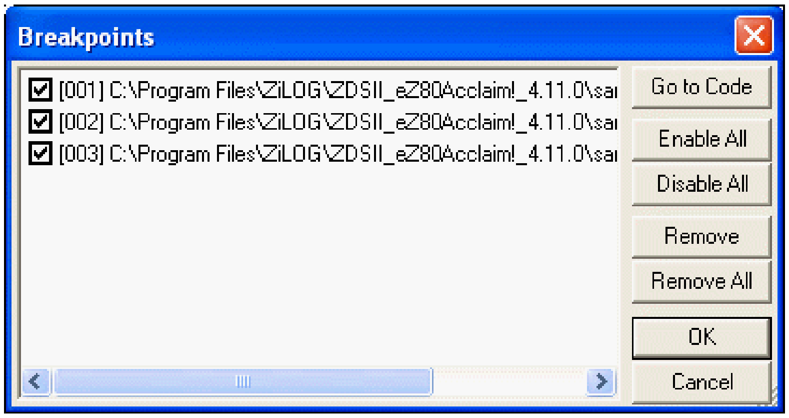


Figure 44. Go to Line Number Dialog Box

Enter the desired line number in the edit field and click **Go To**.

Manage Breakpoints

To view, go to, or remove breakpoints, select **Manage Breakpoints** from the Edit menu. You can access the dialog box during Debug mode and Edit mode.



☑ [001] C:\Program Files\ZiLOG\ZDSII_eZ80Acclaim!_4.11.0\sa
☑ [002] C:\Program Files\ZiLOG\ZDSII_eZ80Acclaim!_4.11.0\sa
☑ [003] C:\Program Files\ZiLOG\ZDSII_eZ80Acclaim!_4.11.0\sa

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.

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 on breakpoints, see “Using Breakpoints” on page 306.

Toggle Bookmark

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

Next Bookmark

Select **Next Bookmark** from the Edit menu to position the cursor at the line where 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 where 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 to display on the eZ80Acclaim! developer’s environment.

The View menu contains these options:

- “Debug Windows” on page 53
- “Workspace” on page 54
- “Output” on page 54
- “Status Bar” on page 54

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 on the Debug windows, see “Debug Windows” on page 292.

The Debug Windows submenu contains the following:

- “Registers Window” on page 292
- “Special Function Registers Window” on page 293
- “Clock Window” on page 294
- “Memory Window” on page 295
- “Watch Window” on page 300
- “Locals Window” on page 302
- “Call Stack Window” on page 303
- “Symbols Window” on page 304
- “Disassembly Window” on page 305
- “Simulated UART Output Window” on page 306

Workspace

Select **Workspace** from the View menu to display or hide the Project Workspace window.

Output

Select **Output** from the View menu to display or hide the Output windows.

Status Bar

Select **Status Bar** from the View menu to display or hide the status bar, which resides beneath the Output windows.

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:

- “Add Files” on page 54
- “Remove Selected File(s)” on page 55
- “Settings” on page 55
- “Export Makefile” on page 106

Add Files

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

1. From the Project menu, select **Add Files**.

The Add Files to Project dialog box is displayed.

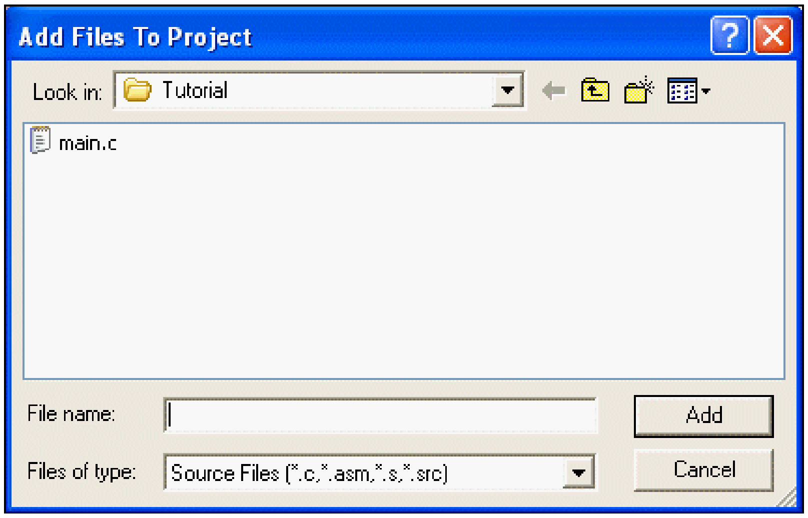


Figure 46. Add Files to Project Dialog Box

2. Use the Look In drop-down list box to navigate to the appropriate directory where the files you want to add are saved.
3. Click on the file you want to add or highlight multiple files by clicking on each file while holding down the Shift key.

   If you select files with .htm, .html, .class, .jar, .jpg, .jpeg, .wav, or .gif extensions, the files are converted to C files and saved in the Web Files folder in the Project Workspace window. These files are automatically converted to C source files during the build process.
4. Click **Add** to add these files to your project.

Remove Selected File(s)

Select **Remove Selected File(s)** 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 list box in the upper left corner of the Project Settings dialog box. For your convenience, the Debug and Release configurations are predefined. For more information on project configurations such as adding your own configuration, see “Set Active Configuration” on page 108.

The Project Settings dialog box has different pages you must use to set up the project:

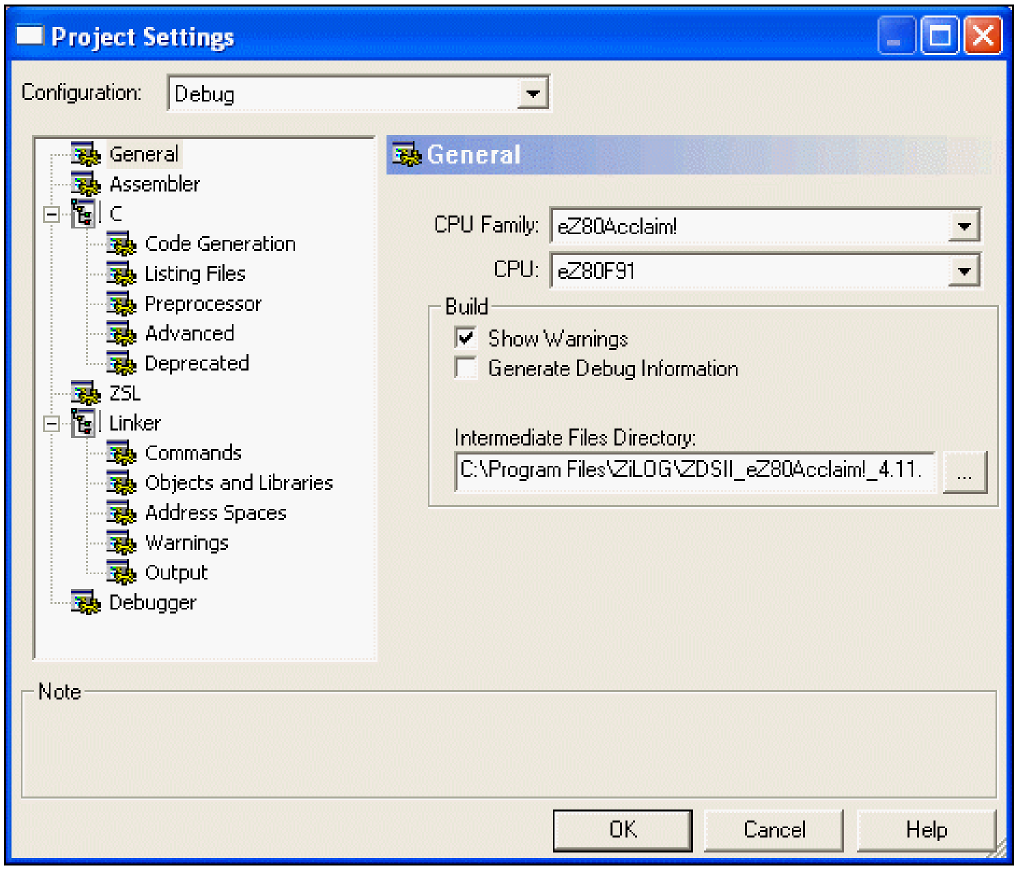
- "Project Settings—General Page" on page 57
- "Project Settings—Assembler Page" on page 59
- "Project Settings—Code Generation Page" on page 61 (not available for Assembly Only projects)
- "Project Settings—Listing Files Page" on page 63 (not available for Assembly Only projects)
- "Project Settings—Preprocessor Page" on page 65 (not available for Assembly Only projects)
- "Project Settings—Advanced Page" on page 66 (not available for Assembly Only projects)
- "Project Settings—Deprecated Page" on page 69 (not available for Assembly Only projects)
- "Project Settings—Librarian Page" on page 74 (available for Static Library projects only)
- "Project Settings—ZSL Page" on page 75
- "Project Settings—Commands Page" on page 77 (available for Executable projects only)
- "Project Settings—Objects and Libraries Page" on page 84 (available for Executable projects only)
- "Project Settings—Address Spaces Page" on page 89 (available for Executable projects only)
- "Project Settings—Warnings Page" on page 90 (available for Executable projects only)
- "Project Settings—Output Page" on page 92 (available for Executable projects only)
- "Project Settings—Debugger Page" on page 95 (available for Executable projects only)

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, Advanced, and Deprecated. 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.

Project Settings—General Page

From the Project Settings dialog box, select the General page. The options on the General page are described in this section.



CPU Family: eZ80Acclaim!
CPU: eZ80F91
☑ Show Warnings
☐ Generate Debug Information
Intermediate Files Directory: C:\Program Files\ZiLOG\ZDSII_eZ80Acclaim!_4.11.

Figure 47. General Page of the Project Settings Dialog Box

CPU Family

The CPU Family drop-down list box allows you to select the eZ80® or eZ80Acclaim! family.

CPU

The CPU drop-down list box defines which CPU you want to define for the target.

To change the CPU for your project, select the appropriate CPU in the CPU drop-down list box.

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

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

Generate Debug Information

The Generate Debug Information check box 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 check box 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” on page 62). 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 eZ80Acclaim! architecture, so the code for these functions is slightly smaller if this check box is disabled.

NOTE: This check box interacts with the Limit Optimizations for Easier Debugging check box on the Code Generation page (see “Limit Optimizations for Easier Debugging” on page 62). When the Limit Optimizations for Easier Debugging check box is selected, debug information is always generated so that debugging can be performed. The Generate Debug Information check box is grayed out (disabled) when the Limit Optimizations for Easier Debugging check box is selected. If the Limit Optimizations for Easier Debugging check box is later deselected (even in a later ZDS II session), the Generate Debug Information check box returns to the setting it had before the Limit Optimizations for Easier Debugging check box was selected.

Ignore Case of Symbols

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

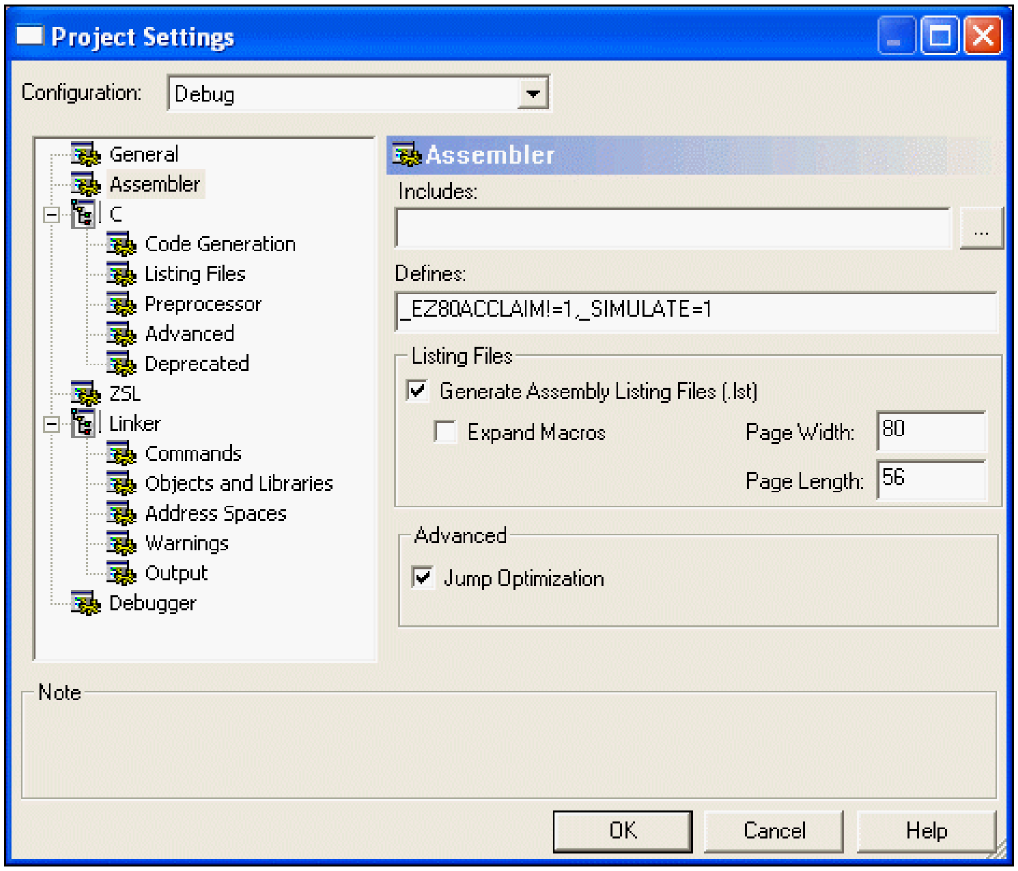
Intermediate Files Directory

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 "Project Settings—Output Page" on page 92).

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 are described in this section.



Configuration: Debug

Includes:

Defines: _EZ80ACCLAIM!=1,_SIMULATE=1

Listing Files

[✓] Generate Assembly Listing Files (.lst)

[ ] Expand Macros

Page Width: 80

Page Length: 56

Advanced

[✓] Jump Optimization

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 on the default ZDS II include directories.

The ZDS II default include directories 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_eZ80Acclaim!_<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) check box 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 in the General page (see “Intermediate Files Directory” on page 59). By default, this check box is selected.

Expand Macros

When selected, the Expand Macros check box tells the assembler to expand macros in the assembly listing file.

Page Length

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

Jump Optimization

When selected, the Jump Optimization check box allows the assembler to replace relative jump instructions (JR and DJNZ) with absolute jump instructions when the target label is either

- outside of the +127 to –128 range

  For example, when the target is out of range, the assembler changes

  ```
  DJNZ r0, lab
  ```

  to

  ```
  DJNZ r0, lab1
  JR lab2
  lab1:JP lab
  lab2:
  ```

- external to the assembly file

  When the target label is external to the assembly file, the assembler always assumes that the target address is out of range.

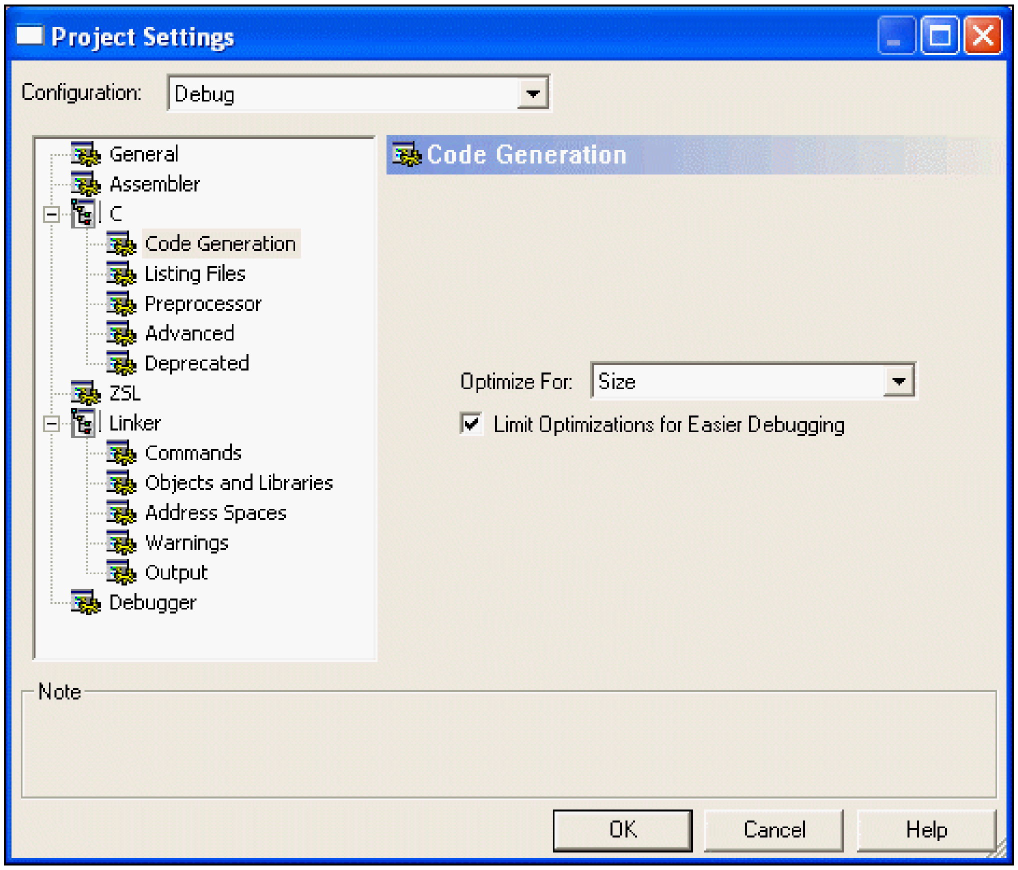
It is usually preferable to allow the assembler to make these replacements because if the target of the jump is out of range, the assembler would otherwise not be able to generate correct code for the jump. However, if you are very concerned about monitoring the code size of your assembled application, you can deselect the Jump Optimization check box. You will then get an error message (from the assembler if the target label is in the same assembly file or from the linker if it is not) every time the assembler is unable to reach the target label with a relative jump. This might give you an opportunity to try to tune your code for greater efficiency.

The default is checked.

Project Settings—Code Generation Page

NOTE: For Assembly Only projects, the Code Generation page is not available.

The following figure shows the Code Generation page.



Configuration: Debug

Optimize For: Size

✓ Limit Optimizations for Easier Debugging

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

Optimize For

Most code optimizations that can be applied by the compiler generate code that is both more compact and faster after the optimization. However, in a small percentage of cases, there is a trade-off between optimizing for these two goals. When you select **Size** in this drop-down list box, the compiler makes smaller code size its preferred criterion in such cases. Conversely, when you select **Speed**, the compiler values faster execution speed above size when a trade-off must be made. The differences in size and speed for your overall application between these two settings is real but, usually, fairly small due to the relative rarity of code that presents a size-speed optimization trade-off.

Limit Optimizations for Easier Debugging

Selecting this check box 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 check box, but it might difficult because of the changes that these optimizations make in the assembly code generated by the compiler.

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

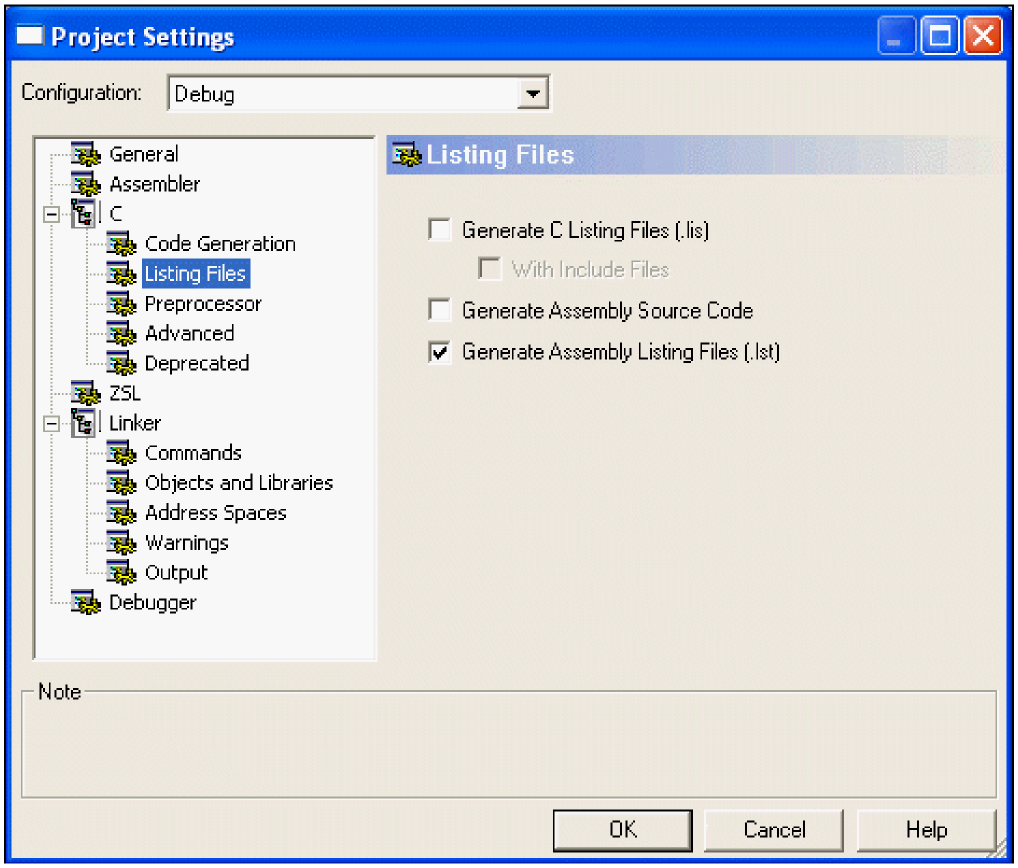
You *can* debug your application when this check box is deselected. The debugger continues to function normally, but debugging might be more confusing due to the factors described earlier.

NOTE: This check box interacts with the Generate Debug Information check box (see "Generate Debug Information" on page 58).

Project Settings—Listing Files Page

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

The following figure shows the Listing Files page.



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) check box 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 check box 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 check box 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 check box in the General page. See “Intermediate Files Directory” on page 59.

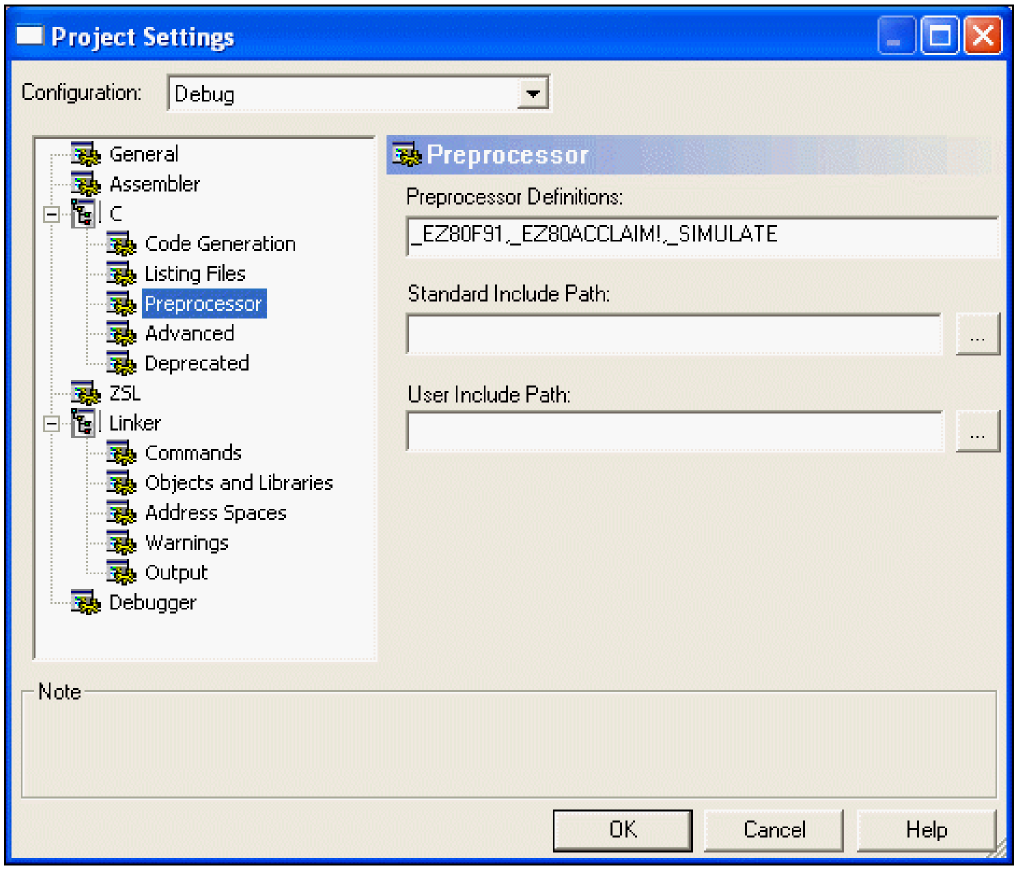
Generate Assembly Listing Files (.lst)

When this check box 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 59.

Project Settings—Preprocessor Page

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

The following figure shows the Preprocessor page.



Configuration: Debug

Preprocessor Definitions: _EZ80F91,_EZ80ACCLAIM!,_SIMULATE

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_eZ80Acclaim!_<version>`, where <*version*> might be `4.11.0` or `5.0.0.`

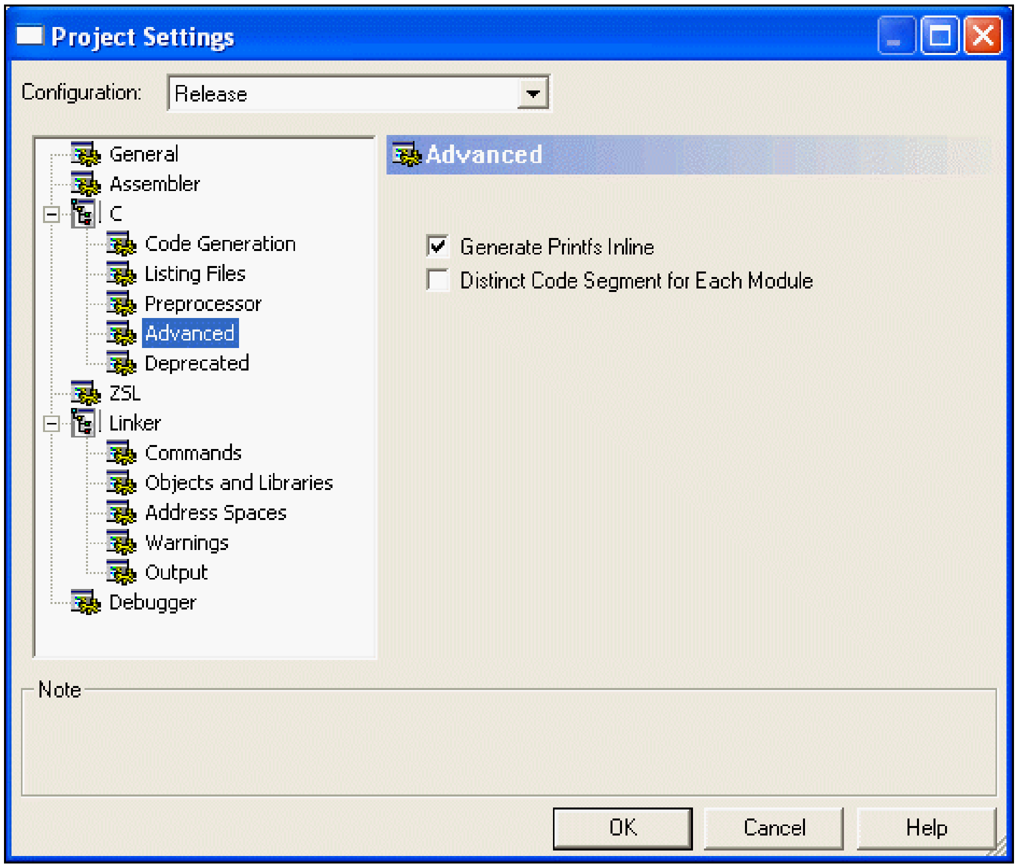
User Include Path

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

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

The following figure shows the Advanced page. This page is used for options that most users will rarely need to change from their default settings.



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

Figure 52. Advanced Page of the Project Settings Dialog Box

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 check box 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 check box 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 check box, 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:

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

  But code like 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 check box 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 check box selected because of the cumulative effect of all 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 check box is selected by default.

Distinct Code Segment for Each Module

For most applications, the code segment for each module compiled by the eZ80Acclaim! 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 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 check box to accomplish this purpose. When this check box 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 check box is deselected by default. Because you only need to select this check box if your project has an unusually complex link structure, this control has been placed on the Advanced page.

An example of the use of this feature would be to place most of the application's code in the ROM address space (see "Project Settings—Address Spaces Page" on page 89 for a discussion of the eZ80Acclaim! address spaces) except for a particular module that is to be run from the RAM space.

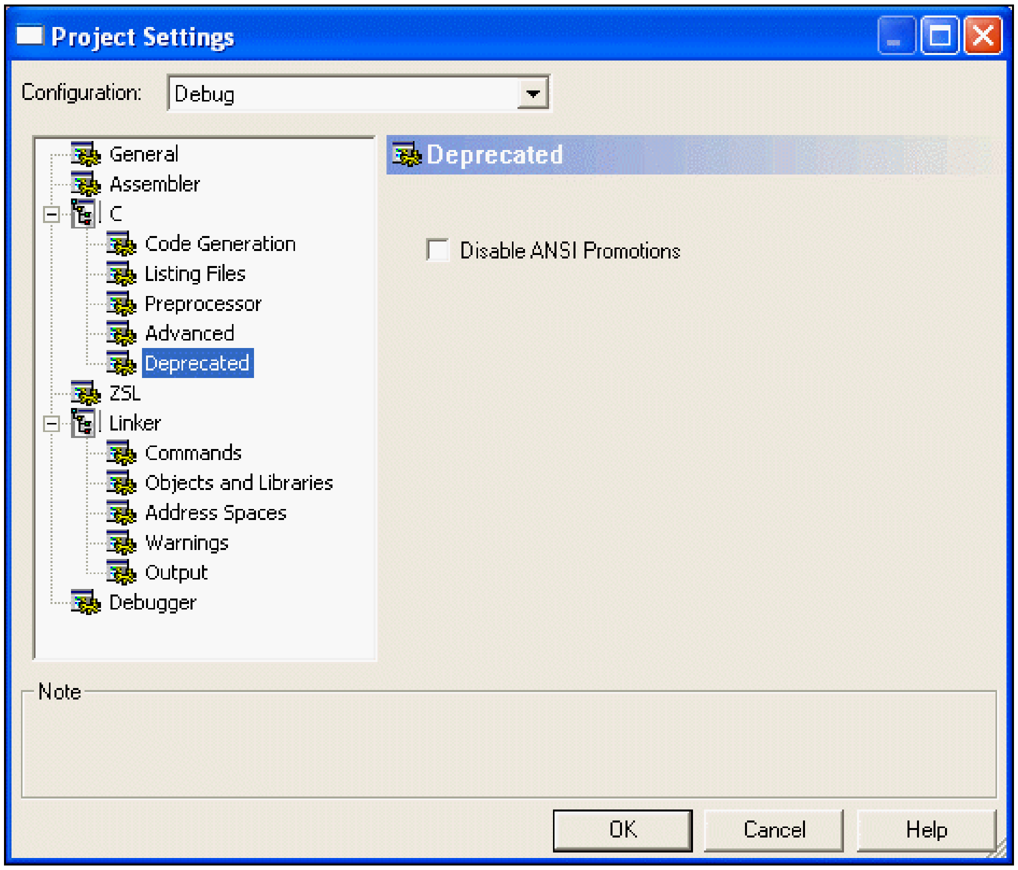
An important restriction on the use of this option for the eZ80Acclaim! is that it generally should not be used (check box deselected) when using the Copy to RAM link configuration (see "Copy to RAM" on page 80). Selecting the option prevents the linker from being able to bundle up the code segments from different modules into a single contiguous block to be copied to RAM.

NOTE: It is the user's responsibility to configure the linker command file properly when the Distinct Code Segment for Each Module check box is selected.

Project Settings—Deprecated Page

NOTE: For Assembly Only projects, the Deprecated page is not available.

The following figure shows the Deprecated page. This page contains options from older releases of ZDS II that, because of issues found in extended experience with those particular options across many applications, are no longer recommended for use. ZiLOG strongly recommends that you not use these features in new projects. If you have older projects that use these options, they will continue to be supported as in previous applications. However, ZiLOG recommends removing them from your projects over time to avoid the issues that have caused these features to be deprecated.



Configuration: Debug

Deprecated

Disable ANSI Promotions [ ]

Figure 53. Deprecated Page of the Project Settings Dialog Box

Disable ANSI Promotions

The option of enabling or disabling ANSI promotions refers to promoting char and short values to ints when doing computations, as described in more detail in this section. Disabling the promotions was made a user option in earlier releases of ZDS II with the goal of reducing code size because the promotions called for by the ANSI C standard are often unnecessary and can lead to considerable code bloat. However, over time, several problems were found in the compiler’s ability to apply this option consistently and correctly in all cases. Therefore, ZiLOG no longer recommends the use of this feature and, to address the original code size issue, has expended more effort to reduce code size and remove truly unnecessary promotions while observing the ANSI standard. For this reason, the Disable ANSI Promotions check box is now available only as a deprecated feature. It remains available because some users have carefully created working code that might depend on the old behavior and might have to expend additional effort now to keep their code working without the deprecated feature.

When the Disable ANSI Promotions check box is deselected, the compiler performs integer type promotions when necessary so that the program's observed behavior is as defined by the ANSI C Standard. Integer type promotions are conversions that occur automatically when a smaller (for example, 8 bits) variable is used in an expression involving larger (for example, 24 bits) variables. For example, when mixing chars and ints in an expression, the compiler casts the chars into ints. Conversions of this kind are always done, regardless of the setting of the Disable ANSI Promotions check box.

The ANSI Standard has special rules for the handling of chars (and shorts), and it is the application of these special rules that is disabled when the check box is selected. The special rules dictate that chars (both signed and unsigned) must always be promoted to ints before being used in arithmetic or relational (such as < and ==) operations. By selecting the ANSI Promotions check box, these rules are disregarded, and the compiler can operate on char entities without promoting them. This can make for smaller code because the compiler does not have to create extra code to do the promotions and then to operate on larger values. In making this a deprecated feature, ZiLOG has worked to make the compiler more efficient at avoiding truly needless promotions so that the code size penalty for observing the standard is negligible.

Disabling the promotions can often be a safe optimization to invoke, but this is subject to several exceptions. One exception is when an arithmetic overflow of the smaller variable is possible. (For example, the result of adding `(char)10` to `(char) 126` does not fit within an 8-bit `char` variable, so the result is `(char) -120`.) In such cases, you get different results depending on whether ANSI promotions are enabled or disabled. If you write

```
char a = 126;
char b = 10;
int i = a + b:
```

then with ANSI promotions enabled, you get the right answer: 136. With ANSI promotions disabled, you get the wrong answer: -120. The reason for the different result is that while in both cases there is a conversion from `char` to `int`, the conversion is applied earlier or later depending on this setting. With ANSI promotions enabled, the conversion is done as soon as possible, so it occurs before the addition, and the result is correct even though it is too large to fit into a `char`. With ANSI promotions disabled, the conversion is not done until a larger type is explicitly called for in the code. Therefore, the addition is done with `chars`, the overflow occurs, and only after that is the result converted to `int`.

By the ANSI Standard, these special promotions are only applied to `chars` and `shorts`. If you have the analogous code with the sum of two `ints` being assigned into a `long`, the compiler does not automatically promote the ints to longs before adding them, and if the sum overflows the `int` size, then the result is always wrong whether ANSI promotions are in effect or not. In this sense, the ANSI promotions make the handling of char types inconsistent compared to the treatment of other integer types.

It is better coding practice to show such promotions explicitly, as in the following:

```
int i = (int) a + (int) b;
```

Then, you get the same answer whether promotions are enabled or disabled. If instead, you write:

```
char c = a + b;
```

then even with ANSI promotions enabled, you do not get the right answer. You did not anticipate that the arithmetic operation can overflow an 8-bit value. With ANSI promotions disabled, the value of the expression (136) is truncated to fit into the 8-bit result, again yielding the value (char) -120. With ANSI promotions enabled, the expression evaluates directly to (char) -120. In this case, disabling ANSI promotions gives you the same wrong answer more efficiently!

There are two more types of code constructs that behave differently from the ANSI Standard when the ANSI promotions are disabled. These occur when an expression involving unsigned chars is then assigned to a signed int result and when relational operators are used to compare an unsigned char to a signed char. Both of these are generally poor programming practice due to the likelihood of operand signs not being handled consistently.

The following code illustrates the cases where the code behaves differently depending on the setting of the Disable ANSI Promotions check box. When ANSI promotions are on, the code prints the following:

```
START
EQUAL
EQUAL
EQUAL
SIGNED
DONE
```

When ANSI promotions are off, the code prints the following:

```
START
NOT EQUAL
NOT EQUAL
NOT EQUAL
UNSIGNED
DONE
```

In every case, the difference occurs because when promotions are on, the unsigned chars are first promoted to signed ints, and then the operation occurs; with promotions off, the operations occur first, and then the promotion happens afterward. In every case except the second test, the code with promotions off has to invoke the ANSI Standard's rules for how to convert a negative result into an unsigned type—another indication that it is generally poorly written code for which this setting makes a difference in program behavior.

```
#include <stdio.h>

unsigned char uch1 = 1;
unsigned char uch2 = 2;
unsigned char uch3 = 128;
int int1;
int int2;
char ch1 = -2;

char *neq_str = "NOT EQUAL";
char *eq_str = "EQUAL";

int main(void)
{
   puts("START");

   int1 = uch1 - uch2;
   if (int1 != -1)
     puts(neq_str);     //nopromote:00FFh != FFFFh
   else
     puts(eq_str);      //promote:  FFFFh == FFFFh

   int2 = ~uch3;
   if (int2 != ~128)
     puts(neq_str);     //nopromote:007Fh != FF7Fh
   else
     puts(eq_str);      //promote:  FF7Fh == FF7Fh

   int2 = -uch3;
   if (int2 != -128)
     puts(neq_str);     //nopromote:0080h != FF80h
   else
     puts(eq_str);      //promote:  FF80h == FF80h

   if (uch3 < ch1)
     puts("UNSIGNED");  //nopromote:(uchar)80h < (uchar)FEh
   else
     puts("SIGNED");    //promote:  (int)  128 > (int)  -2

   puts("DONE.");
}
```

The following recommended programming practices are good practice in any case for producing code that is both correct and efficient. These practices are especially important to avoid trouble if you are using the deprecated Disable ANSI Promotions option:

- Use variables of type char or unsigned char wherever the expected range of values for the variable is [-128..127] or [0..255], respectively.
- Use explicit casts (to int, unsigned int, long or unsigned long) where the result of an expression is expected to overflow the larger of the two operand types. (Even with ANSI promotions disabled, the compiler automatically promotes a smaller operand so that the types of the operands match.)

- It is good programming practice to use explicit casts, even where automatic promotions are expected.
- Explicitly cast constant expressions that you want to be evaluated as char (for example, `(char)0xFF`).

As a final note, due to the eZ80® processor architecture, it is important to deselect the Disable ANSI Promotions check box when you need a local stack frame that is longer than 127 bytes.

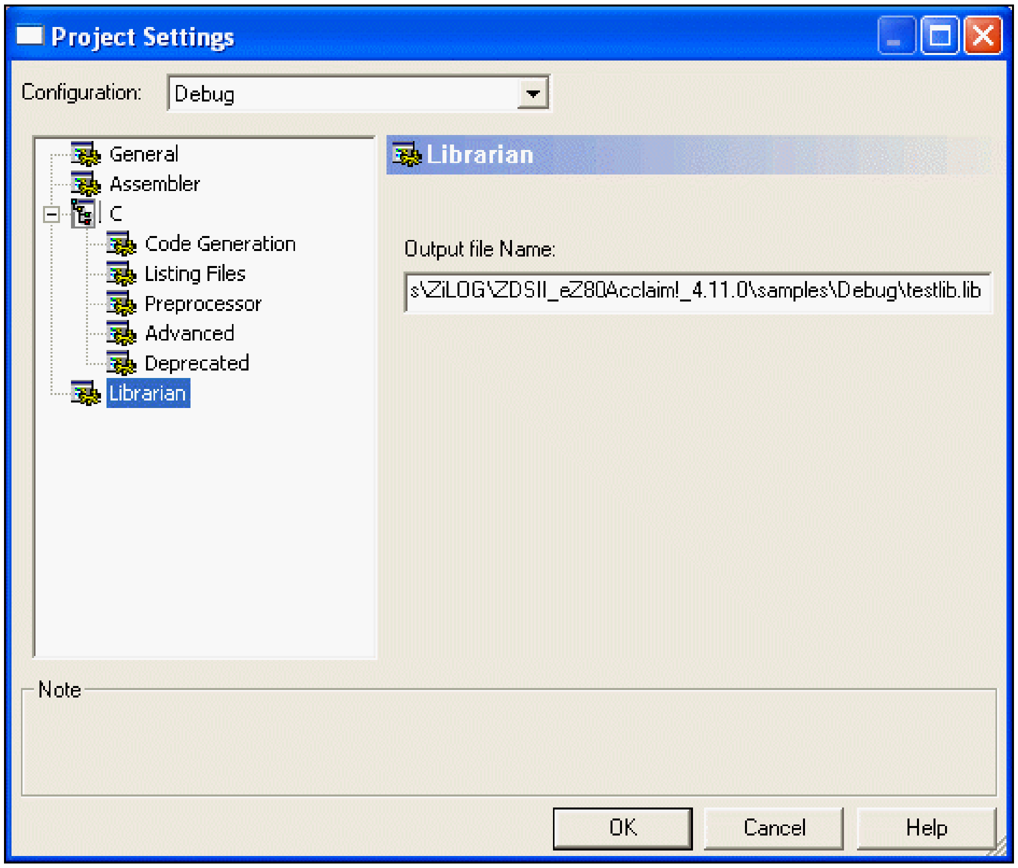
Project Settings—Librarian Page

NOTE: This page is available for Static Library projects only.

To configure the librarian, use the following procedure:

1. Select **Settings** from the Project menu.

   The Project Settings dialog box is displayed.
2. Click the Librarian page.



Configuration: Debug

Librarian

Output file Name:

s\ZiLOG\ZDSII_eZ80Acclaim!_4.11.0\samples\Debug\testlib.lib

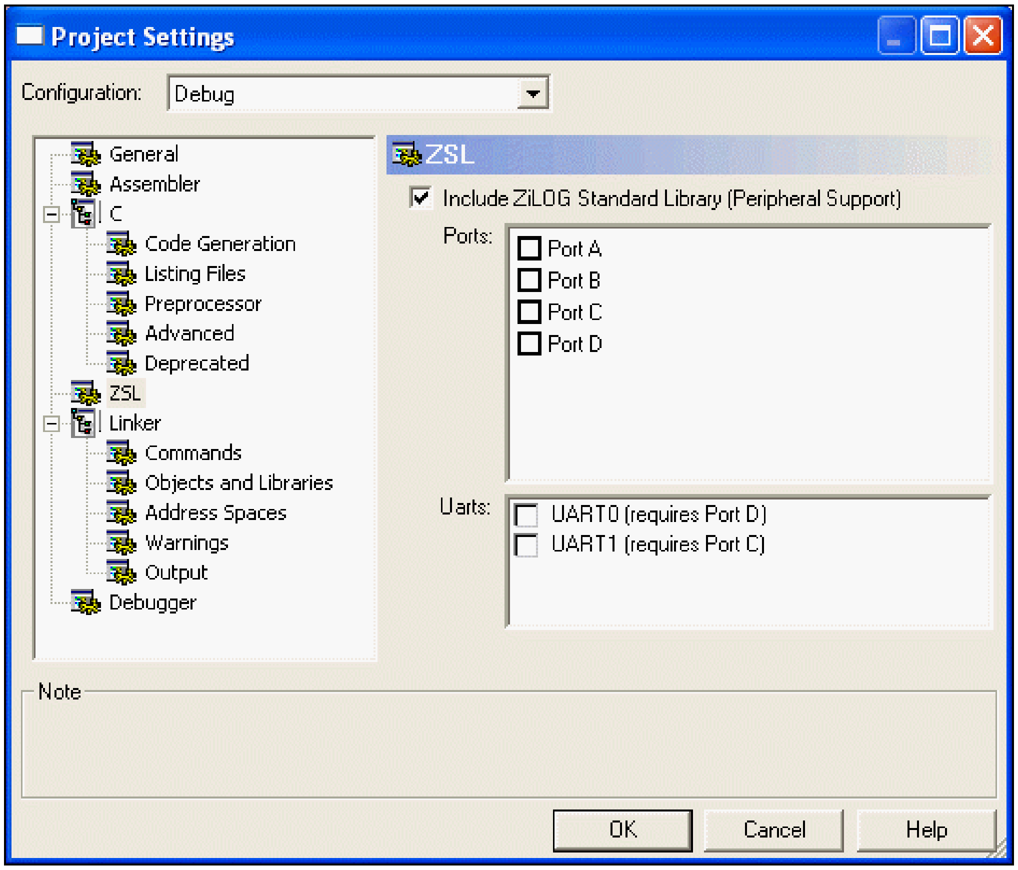
Figure 54. Librarian Page of the Project Settings Dialog Box

3. Use the Output File Name field to specify where your static library file is saved.

Project Settings—ZSL Page

In the Project Settings dialog box, select the ZSL page. The ZSL page allows you to use the ZiLOG Standard Library (ZSL) in addition to the run-time library (described in the "Using the ANSI C-Compiler" chapter). The ZSL contains functions for controlling the UART device driver and GPIO ports. These functions are described in detail in the *ZiLOG Standard Library API Reference Manual* (RM0037).

The options on the ZSL page are described in this section.



Configuration: Debug

[✓] Include ZiLOG Standard Library (Peripheral Support)

Ports:
- [ ] Port A
- [ ] Port B
- [ ] Port C
- [ ] Port D

Uarts:
- [ ] UART0 (requires Port D)
- [ ] UART1 (requires Port C)

Figure 55. ZSL Page of the Project Settings Dialog Box

Include ZiLOG Standard Library (Peripheral Support)

Select the Include ZiLOG Standard Library (Peripheral Support) check box to use the functions contained in the ZiLOG Standard Library. Some of the functions in the C Standard Library, especially I/O functions like `printf()`, rely on lower-level functions that they call to eventually interact with hardware devices such as UARTs. The ZiLOG Standard Library provides these lower-level support functions, specialized to the eZ80® as described in the *ZiLOG Standard Library API Reference Manual* (RM0037). Therefore, if you choose to deselect this check box and avoid using the functions of the ZSL, you must provide your own replacements for them or else rewrite the calling functions in the C runtime library so that the ZSL functions are not called.

Ports

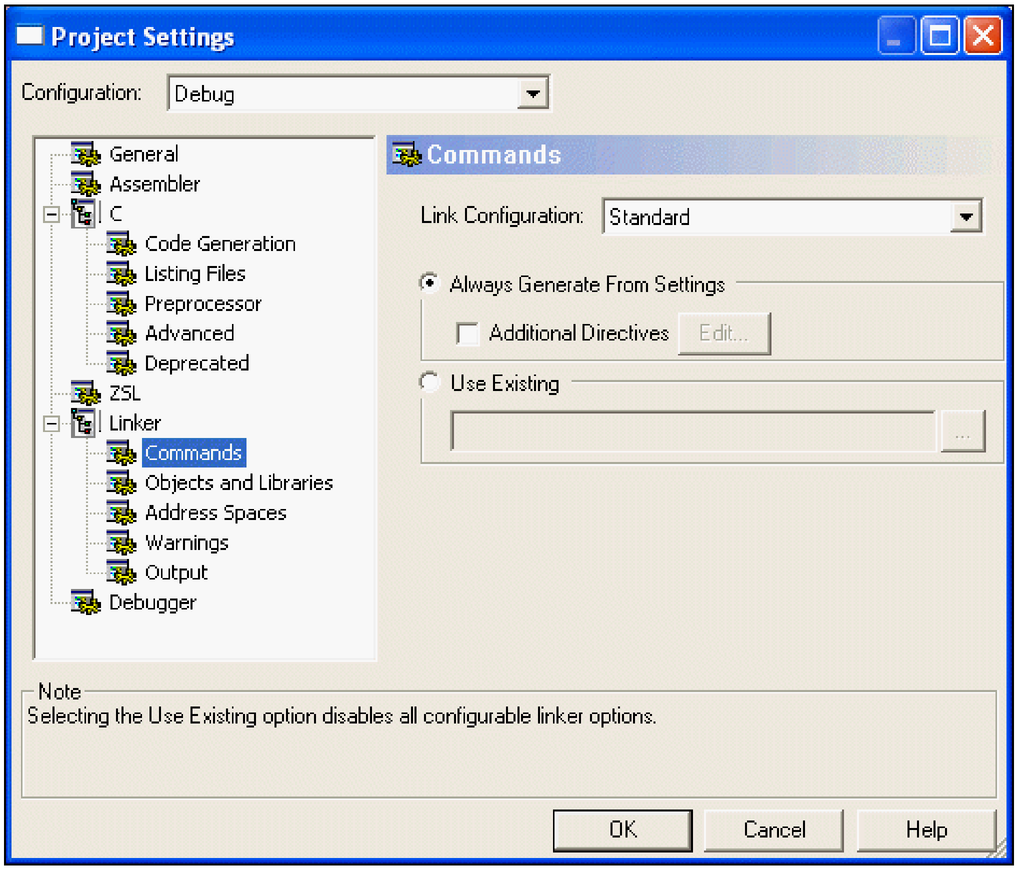
In the Ports area, select the check boxes for the ports that you are going to use.

Uarts

In the Uarts area, select the check boxes for the UARTS that you are going to use.

Project Settings—Commands Page

The following figure shows the Commands page.



Configuration: Debug

Link Configuration: Standard

(●) Always Generate From Settings

☐ Additional Directives

( ) Use Existing

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

Figure 56. Commands Page of the Project Settings Dialog Box

Link Configuration

Use the Link Configuration drop-down list box to select the configuration that best fits your target. The project settings are modified accordingly. ZDS II automatically generates a linker command file using the project settings. You can choose to include your own linker command file.

NOTE: ZDS II—eZ80Acclaim! allows you a great deal of flexibility over where physical blocks of RAM and ROM (both internal and external) are located in the system. However, any blocks of Flash memory that use different programming algorithms must be separated by at least 0x40 bytes. If this is not done, you cannot flash the project on the target. As an example, the internal and external Flash memory on the eZ80F91 development kit use different Flash algorithms so the Target memory range must contain at least a 0x40 byte gap between these two Flash regions.

- All RAM

  This configuration produces a single memory image that resides in, and executes from, RAM. This configuration is most useful when emulating code from RAM on the eZ80L92 and eZ80190 evaluation boards. Using the ZDS II standard startup module with this configuration is recommended as the surest and easiest way to produce an effective run-time environment.

  In this configuration, the linker maps all segments associated with the logical ROM address space to physical RAM. ZDS II therefore automatically generates two linker commands:

  ```
  GROUP MEMORY=ROM,RAM
  ```

  This command defines a new address space (MEMORY) that contains the existing logical address spaces RAM and ROM.

  ```
  RANGE MEMORY $0 : $FFFF
  ```

  This command defines the physical address range for the combined spaces.

  ZDS II creates the RANGE command starting with the lowest address specified in the Address Spaces page and ending with the highest address. The lowest address is `min(min(ROM), min(RAM))` and the highest address is `max(max(ROM),max(RAM))`. These two ZDS II-generated commands are critical for building an All RAM configuration.

  Both the `GROUP MEMORY` and `RANGE MEMORY` commands are required for the All RAM configuration. If the `RANGE MEMORY` command is omitted, the result can be wasted space. To illustrate the effects of such an error, suppose the Address Spaces page has the following values for a 64K memory machine:

  ```
  ROM 0-7FFF
  ```

  ```
  RAM 8000-FFFF
  ```

  If the All RAM configuration is not used to create the linker command file, ZDS II converts the values in the Address Spaces page to the following linker commands:

  ```
  RANGE ROM $0 : $7FFF
  ```

  ```
  RANGE RAM $8000 : $FFFF
  ```

  which, if followed by

  ```
  GROUP MEMORY=ROM,RAM
  ```

  would result in ROM segments starting at address 0h and RAM segments starting at `8000h`, potentially wasting space if the ROM segments do not fully occupy the range from `0h - 7FFFh`. The `RANGE MEMORY 0h : FFFFh` command overrides the

`RANGE` commands for the component address spaces and binds the RAM segments immediately after the ROM segments.

NOTE: The names following the = in the `GROUP` command define an ordering for the new GROUP. In the preceding example, all of the ROM segments are allocated memory at lower addresses than the RAM segments.

- Standard

  This configuration produces the most common embedded environment. Code resides in Flash/ROM, and data is placed in RAM. Using the ZDS II standard startup module with this configuration is recommended as the surest and easiest way to produce an effective run-time environment.

  In the Standard configuration, the hardware has both physical RAM and physical ROM. The Linker commands generated by ZDS II map logical RAM segments to physical RAM and logical ROM segments to physical ROM. Suppose the Address Spaces page contained the following values:

  ```
  ROM 0-7FFF
  RAM A000-FFFF
  ```

  ZDS II generates the following code for the linker command file:

  ```
  RANGE ROM $0 : $7FFF
  RANGE RAM $A000 : $FFFF
  ```

  The linker uses the `COPY` command on segments to better support standalone C programs. When running a C program under an operating system such as Windows or UNIX, all initialized variables are set to their starting values upon the start of program execution. In a standalone C implementation, however, there is no operating system to reload variables with their initial values. With no operating system, if an embedded application runs for a while and is then restarted at `main()`, the values of initialized variables will not be restored to their original value. The linker's `COPY` command, together with the startup module, provides a means to reinitialize variables.

  The linker normally loads the DATA segment (initialized data) into RAM. Once the initialized data has been loaded into RAM and modified by program execution, in the absence of the COPY mechanism the only way to reload the initial DATA segment would be to download the code to the target board again. This approach is not practical for most embedded systems. (The embedded application would have to save all the initialized data and reload the initial values upon RESET, for example.)

  The `COPY` command (for example, `COPY DATA ROM`) causes the linker to put a copy of the DATA segment in the ROM space at load time. The standard startup module will always copy the DATA segment to RAM before calling `main()`. The `COPY` command copies segments into spaces only. Any other copy combination generates an error. The startup module requires additional linker commands to perform the copy. "Linker Commands" on page 247 describes these commands.

The `ORDER` command allows the user to define the sequence of segments within a memory space. In the Standard configuration, ZDS II generates the following command to put the initialized data segment at a lower address than the uninitialized data segment:

```
ORDER DATA,BSS
```

- Copy to RAM

  This configuration produces a memory image that resides in Flash/ROM and is copied to RAM for execution. This configuration is typically used to take advantage of RAM's faster operation. Using the ZDS II standard startup module with this configuration is recommended as the surest and easiest way to produce an effective run-time environment.

  This configuration provides support for copying code as well as data segments from physical ROM to physical RAM. The idea is to compensate for the performance penalty often associated with running code from ROM. A ROM instruction fetch, for example, might require more wait states than a RAM instruction fetch. It is therefore more efficient to run code from RAM than to run it from ROM, provided the target system has enough RAM for the program code and data. As in the other configurations, ZDS II automatically generates linker commands in the linker command file to support this operating mode, when the Copy to RAM link configuration is selected.

  To run code from the RAM space, the Linker must do two things:

  – Reassign all of the CODE segment addresses to RAM instead of ROM.
  – Place a copy of the CODE segment in ROM for an application restart (the startup module re-copies it from ROM to RAM upon restart).

  The linker `CHANGE` command allows you either to rename a segment or reassign a segment to another space. For example

```
CHANGE TEXT is DATA
CHANGE CODE is RAM
CHANGE STRSECT is CODE
```

  causes the linker to

  1. Combine the `TEXT` segment into the `DATA` segment.

  2. Reassign the `CODE` segment to the RAM space.

  3. Combine the `STRSECT` segment into the `CODE` segment.

  These three `CHANGE` commands reassign all addresses in the `CODE`, `TEXT`, and `STRSECT` segments into RAM, for the fastest possible execution.

  The final step requires the RAM space to be copied in ROM space, so that it can be reloaded from ROM when the application starts. Adding the following code

```
COPY DATA ROM
COPY CODE ROM
```

causes both the `CODE` and `DATA` segments to be copied to ROM. These segments are then copied by the startup code to the appropriate RAM addresses, completing the actions associated with the Copy to RAM configuration.

NOTE: To use the Copy to RAM configuration (unless you have your own complex and carefully designed allocation of code segments into particular modules), the Distinct Code Segment for Each Module check box in the Advanced page (see "Distinct Code Segment for Each Module" on page 68) must be deselected. Selecting that check box prevents the linker from being able to group the CODE segments from all your code modules into a single contiguous block that can be copied to RAM automatically by the linker commands that are used in this configuration.

- Custom

  Choose this option if the other three options do not provide the right configuration. As a result, you are required to define all of the segment-to-space mappings, depending on the hardware configuration of the application board. You also need to include your own startup module for this selection.

- Deprecated CUSTOM Configuration

  This configuration was provided in earlier releases of ZDS II for eZ80Acclaim! before the 4.10.0 release. It is supported for backward compatibility. In those earlier releases, this configuration was called "Custom," but, in fact, it was nearly identical to the Standard configuration. The only difference was that the ordering of segments was left to the user and that the STRSECT segment was left in RAM instead of ROM. That segment contains string literals and should always be placed in ROM for production code but can be kept in RAM at times for debugging.

  In the 4.10.0 and subsequent releases of ZDS II, the Custom link configuration is used to support truly customized link command files. ZiLOG recommends that if you have an older project that used the Deprecated Custom configuration, you convert it either to the Standard configuration (by changing to that setting and verifying that the minor changes in link commands cause you no problems) or to the Custom configuration (by saving your existing link command file and selecting the Use Existing button (see "Use Existing" on page 83). At some future time, the Deprecated Custom configuration might no longer be supported in ZDS II.

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 “Use Existing” on page 83).

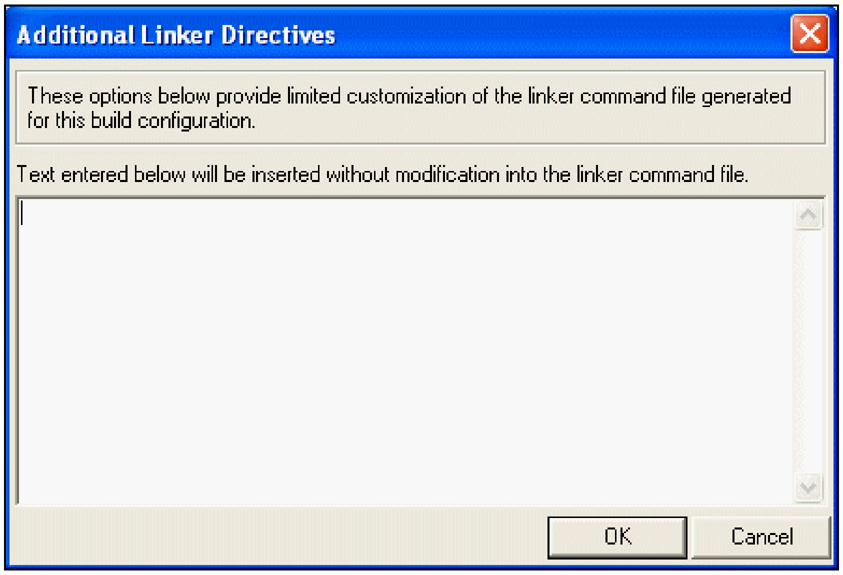
NOTE: Even though selecting Always Generate from Settings causes a new linker command file to be generated when you build your project, any directives that you have specified in the Additional Linker Directives dialog box are not erased or overridden.

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, do the following:

1. Select the Additional Directives check box.
2. Click **Edit**.

   The Additional Linker Directives dialog box is displayed.


Additional Linker Directives

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 57. Additional Linker Directives Dialog Box

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

You can use the Additional Directives check box if you need to 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 xviii.

If you have selected the Additional Directives check box, 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 Startup Module. This is no longer necessary. The directives needed to support a C startup 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 or if a custom linker command file is in use.

Use Existing

Use 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 on the Browse button (...).

   The Select Linker Command File dialog box is displayed.

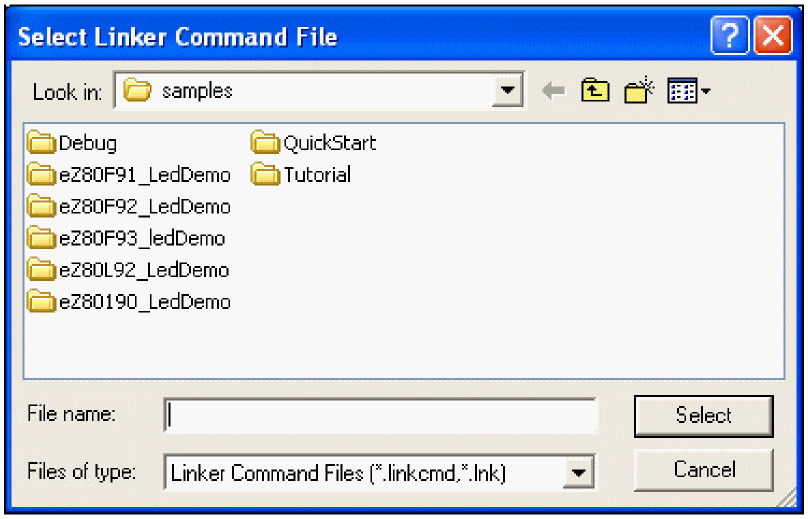


Figure 58. Select Linker Command File Dialog Box

3. Use the Look In drop-down list box to navigate to the linker command file that you want to use.
4. Click **Select**.

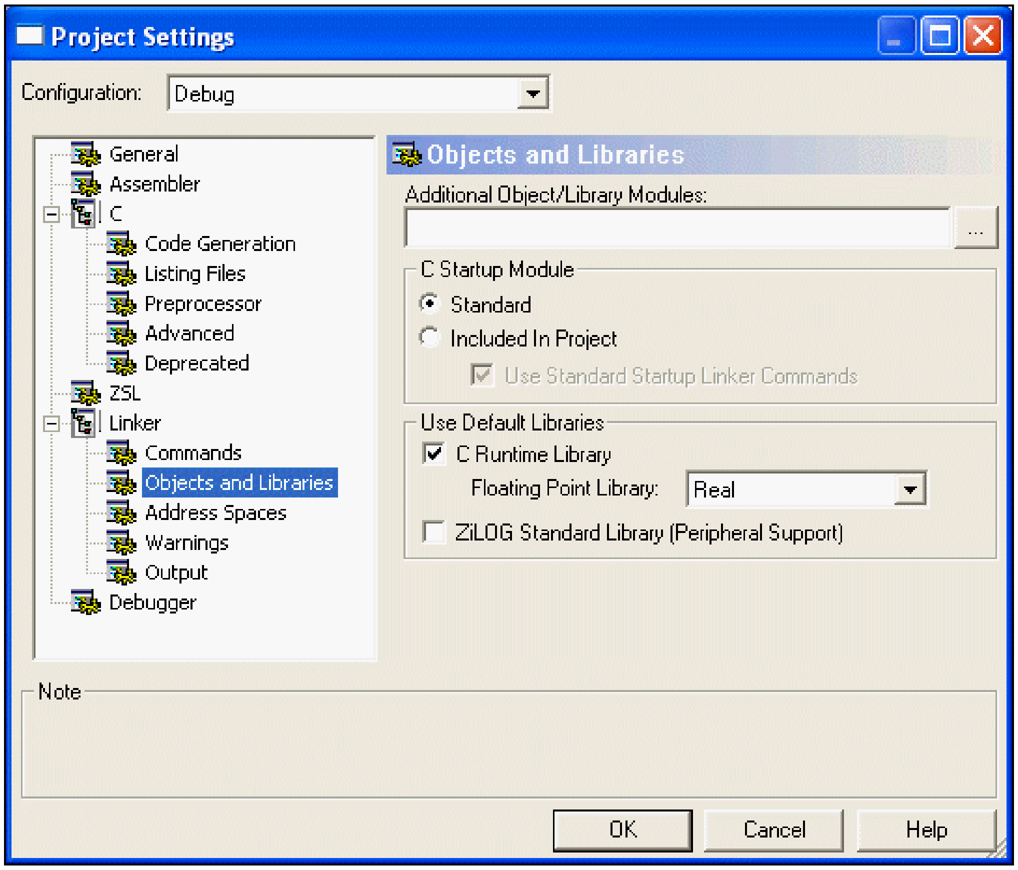
The Use Existing button is the alternative to the Always Generate from Settings button (see "Always Generate from Settings" on page 81). When this button is selected, a new linker command file is not generated when you build your project. Instead, the linker command file that you specify in this field is applied every time.

When the Use Existing button is selected, many project settings are grayed out, including all 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 "Project Workspace Window" on page 29), 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

The following figure shows the Objects and Libraries page.


Project Settings

Configuration: Debug

- General
- Assembler
- C
  - Code Generation
  - Listing Files
  - Preprocessor
  - Advanced
  - Deprecated
- ZSL
- Linker
  - Commands
  - Objects and Libraries
  - Address Spaces
  - Warnings
  - Output
- Debugger

Objects and Libraries

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
- ZiLOG Standard Library (Peripheral Support)

Note

OK | Cancel | Help

Figure 59. 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 startup 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 Startup Module

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

Standard

If the Standard button is selected, the precompiled startup module shipped with ZDS II is used. This standard startup module performs a minimum amount of initialization to prepare the run-time environment as required by the ANSI C Standard and also does some eZ80Acclaim!-specific configuration such as setting up the external memory interface (if selected) and interrupt vector table initialization. See "ANSI Standard Compliance" on page 164 for full details of the operations performed in the standard startup module.

Some of these steps carried out in the standard startup module might not be required for every application, so if code space is extremely tight, you might want to make some judicious modifications to the startup code. The source code for the startup module is located in several files in the following directories:

<ZDS Installation Directory>`\src\boot\common`

and

<ZDS Installation Directory>`\src\boot\`*<processor family>*

Here, *<ZDS Installation Directory>* is the directory in which ZiLOG Developer Studio was installed. By default, this is `C:\Program Files\ZiLOG\ZDSII_eZ80Acclaim!_`*<version>*, where *<version>* might be `4.11.0` or `5.0.0`. Similarly, *<processor family>* is, for example, `eZ80F91` or `eZ80F92`.

The common directory contains code to initialize the C run-time environment and interrupt vectors, while the processor-specific directories have the code necessary for setting up the memory device arrangement. The basic startup source code in the common directory is the `cstartup.asm` file; the `startup.asm` file in the same directory contains older, now outdated initialization code, which is retained for backward compatibility. Refer to the `FAQ.html` file for your release of ZDS II for more details.

Included in Project

If the Included in Project button is selected, then the standard startup module is not linked to your application. In this case, you are responsible for including suitable startup 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 startup module to tailor it to your project, you need to select the Included in Project button for your changes to take effect.

Use Standard Startup Linker Commands

If you select this check box, the same linker commands that support the standard startup module are inserted into your linker command file, even though you have chosen to include your own, nonstandard startup module in the project. This option is usually helpful in getting your project properly configured and initialized because all C startup 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 startup module for initializing the C run-time environment. You do not have to refer to those symbols in your own startup module, but many users will find it useful to do so, especially since user-customized startup modules are often derived from modifying the standard startup module. There are also a few linker commands (such as `CHANGE`, `COPY`, `ORDER`, and `GROUP`) that are used to implement your link configuration. See “Linker Commands” on page 247 for a description of the commands, and “Linker Configurations” on page 240 for an explanation of the link configurations.

If you are using the Standard, Copy to RAM, or All RAM link configuration, you must make sure that your startup module defines the `.RESET`, `.IVECTS`, and `.STARTUP` segments if you want to apply the Use Standard Startup Linker Commands option. These segments are referred to by the startup linker commands for those link configurations. The standard definitions of these segments can be found in the `cstartup.asm` (for `.STARTUP`) and `vectors24.asm` or `vectors16.asm` (for `.RESET` and `.IVECTS`) files. All of these files can be found in the following directory:

<ZDS Installation Directory>`\src\boot\common`

where *<ZDS Installation Directory>* is the directory in which ZiLOG Developer Studio was installed. By default, this is `C:\Program Files\ZiLOG\ZDSII_eZ80Acclaim!_`*<version>*, where *<version>* might be `4.10.0` or `5.0.0`.

This option is only available when the Included in Project button has been selected. The default for newly created projects is that this check box, 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 eZ80Acclaim!, there are two available libraries, the C run-time library and the ZiLOG Standard Library (ZSL). 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 “Floating Point Library” on page 88.

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

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 "Using the ANSI C-Compiler" on page 134. If your project makes any calls to standard library functions, you need to select the Use C Runtime Library check box unless you prefer to provide your own code for all library functions that you call. As noted in "Additional Object/Library Modules" on page 85, 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 check box. 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 check box; only functions you call in your application are linked into your application.

Floating Point Library

The Floating Point Library drop-down list box 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 check box (see "Generate Printfs Inline" on page 67 for a full discussion). You only need to select the dummy floating-point library if you have to disable the Generate Printfs Inline check box and your application uses no floating-point operations. In that case, selecting **Dummy** keeps your code size from bloating unnecessarily.

- None

  If you 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.

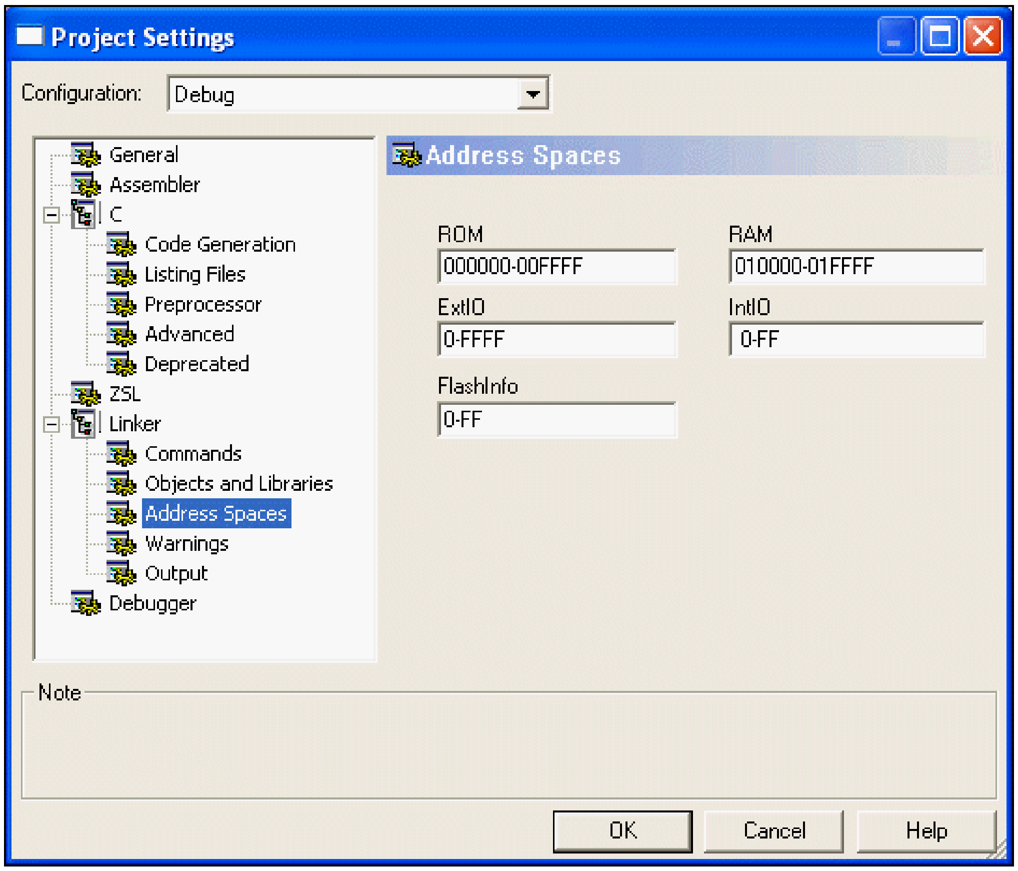
NOTE: None of the libraries mentioned here are available for Assembly Only projects.

ZiLOG Standard Library (Peripheral Support

Select this check box to use the ZiLOG Standard Library (ZSL) in addition to the run-time library (described in the “Using the ANSI C-Compiler” chapter). The ZSL contains functions for controlling the UART device driver and GPIO ports. These functions are described in detail in the *ZiLOG Standard Library API Reference Manual* (RM0037).

Project Settings—Address Spaces Page

The following figure shows the Address Spaces page.



Configuration: Debug

Address Spaces

ROM: 000000-00FFFF
RAM: 010000-01FFFF
ExtIO: 0-FFFF
IntIO: 0-FF
FlashInfo: 0-FF

Figure 60. Address Spaces Page of the Project Settings Dialog Box

Memory ranges are used to determine the amount of memory available on your target system. Using this information, eZ80Acclaim! developer's environment lets 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.

The Address Spaces fields define the memory layout of your target system. The Address Spaces page allows you to configure the ranges of memory available on your target eZ80Acclaim! processor. These ranges vary from processor to processor, as well as from target system to target system.

Depending on your CPU selection, the Address Spaces page can contain the following fields:

- ROM
- RAM
- ExtIO
- IntIO
- FlashInfo

Address ranges are set in the Address Spaces fields. The following is the syntax of a memory range:

<*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:

```
00-df
```

```
0000-ffff
```

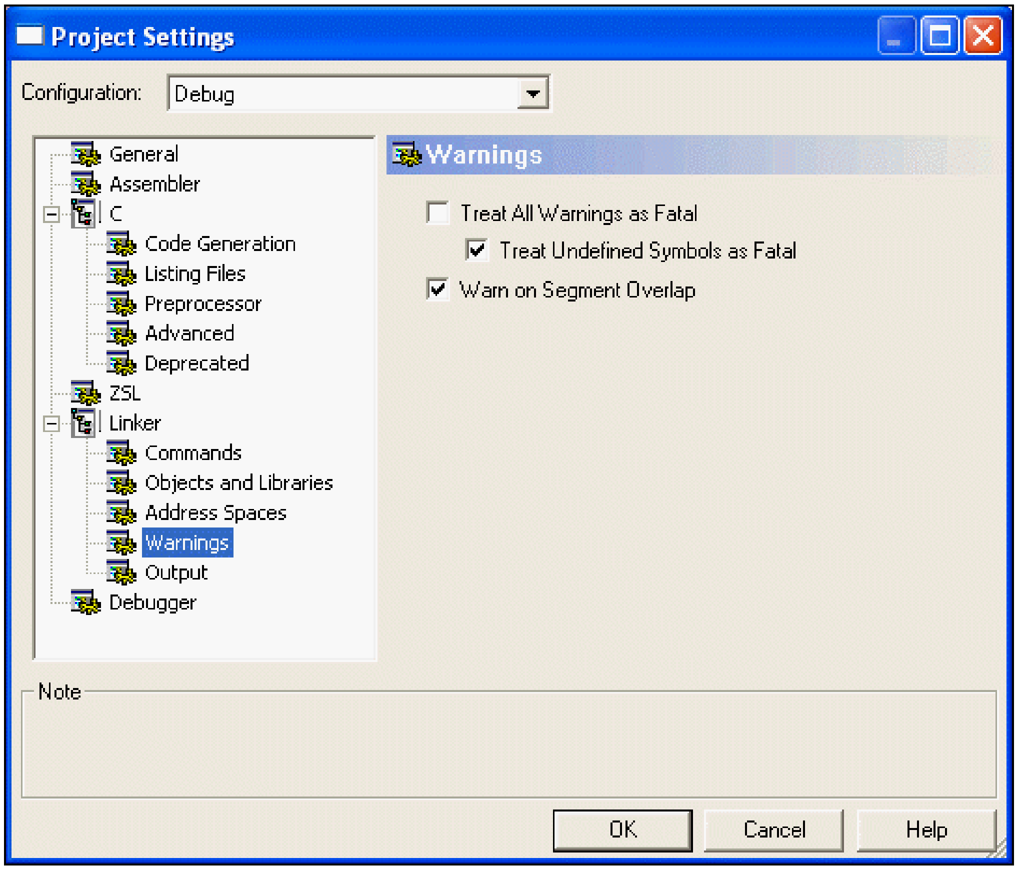
```
0000-1fff,4000-5fff
```

Holes in your memory can be defined for the linker using this mechanism. 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.

NOTE: ZDS II—eZ80Acclaim! allows you a great deal of flexibility over where physical blocks of RAM and ROM (both internal and external) are located in the system. However, any blocks of Flash memory that use different programming algorithms must be separated by at least 0x40 bytes. If this is not done, you cannot flash the project on the target. As an example, the internal and external Flash memory on the eZ80F91 development kit use different Flash algorithms so the target memory range must contain at least a 0x40 byte gap between these two Flash regions.

Project Settings—Warnings Page

The following figure shows the Warnings page.


Project Settings

Configuration: Debug

Warnings

☐ Treat All Warnings as Fatal
☑ Treat Undefined Symbols as Fatal
☑ Warn on Segment Overlap

Figure 61. Warnings Page of the Project Settings Dialog Box

Treat All Warnings as Fatal

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

NOTE: Selecting this check box displays any warning (as errors), regardless of the state of the Show Warnings check box in the General page (see “Show Warnings” on page 58).

Treat Undefined Symbols as Fatal

When selected, this check box causes the linker to treat “undefined external symbol” warnings as fatal errors. If this check box 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 check box is selected because a completely valid executable cannot be built when the program contains references to undefined external symbols. If this check box is deselected, the linker proceeds with generating output files even if there are undefined symbols.

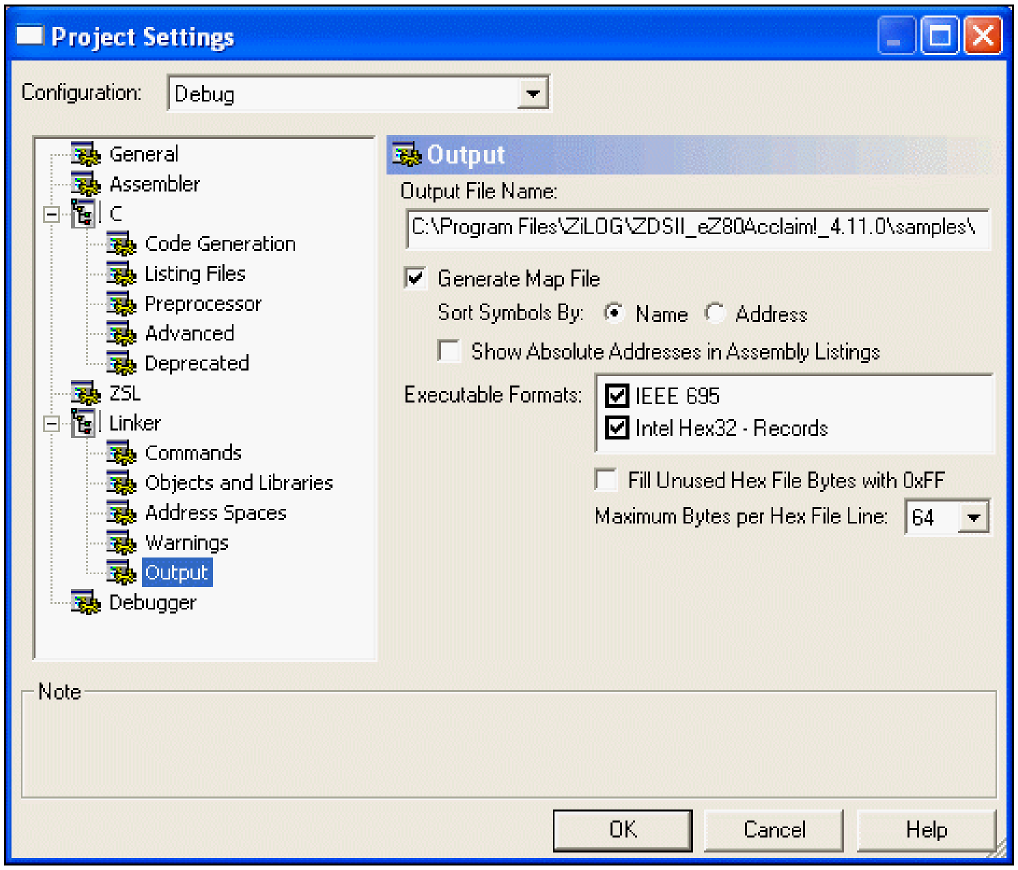
NOTE: Selecting this check box displays any “undefined external symbol” warning as an error, regardless of the state of the Show Warnings check box in the General page (see “Show Warnings” on page 58).

Warn on Segment Overlap

This check box enables or disables warnings when overlap occurs while binding segments. By default, the check box is selected, which is the recommended setting for eZ80Acclaim!. For some ZiLOG processors, benign segment overlaps can occur, but, for the eZ80Acclaim!, an overlap condition usually indicates an error in project configuration that must be corrected. These errors in eZ80Acclaim! 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

The following figure shows the Output page.



Output File Name: C:\Program Files\ZiLOG\ZDSII_eZ80Acclaim!_4.11.0\samples\

[✓] 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 62. Output Page of the Project Settings Dialog Box

Output File Name

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

Generate Map File

This check box 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 "MAP" on page 253 and "How much memory is my program using?" on page 286. Inside the map file, symbols are listed in the order specified by the Sort Symbols By area (see "Sort Symbols By" on page 94).

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 check box 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 check box 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) check box 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) check boxes in the Listing Files page of the Project Settings dialog box must be selected.

Executable Formats

These check boxes 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 (`.hex`), which is a backward-compatible superset of the Intel Hex16 format. IEEE 695 is the default format for debugging in ZDS II, and the Intel hex format is useful for compatibility with some third-party tools. You can also select both check boxes, which produces executable files in both formats.

Fill Unused Hex File Bytes with 0xFF

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

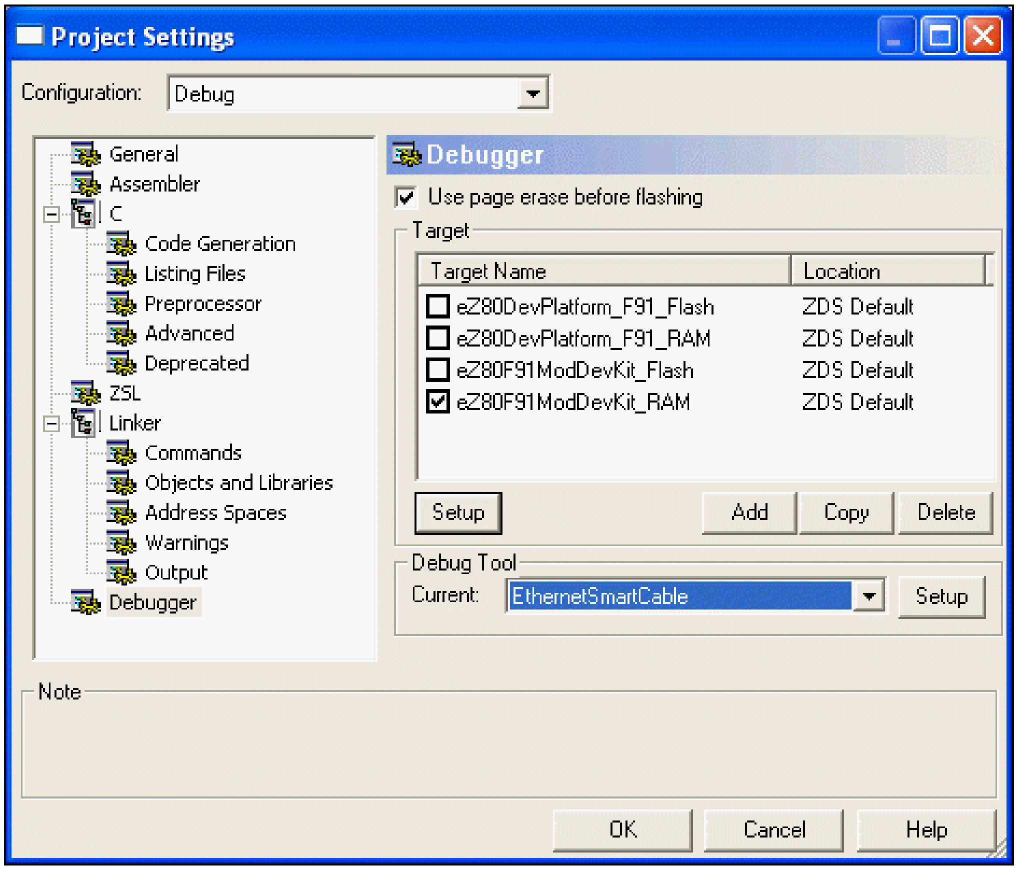
NOTE: Use caution when selecting this option. The resulting hex file begins at the first hex address (`0x0000`) and ends at the last page address that the program requires. This significantly increases the programming time when using the resulting output hex file. The hex file might try to fill nonexistent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

Maximum Bytes per Hex File Line

This drop-down list box 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.



Debugger

☑ Use page erase before flashing

Target

Target Name	Location
☐ eZ80DevPlatform_F91_Flash	ZDS Default
☐ eZ80DevPlatform_F91_RAM	ZDS Default
☐ eZ80F91ModDevKit_Flash	ZDS Default
☑ eZ80F91ModDevKit_RAM	ZDS Default

Debug Tool

Current: EthernetSmartCable

Figure 63. 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 eZ80Acclaim! simulator, the USB Smart Cable, the serial Smart Cable, the Ethernet Smart Cable, and ZPAK II.

Use Page Erase Before Flashing

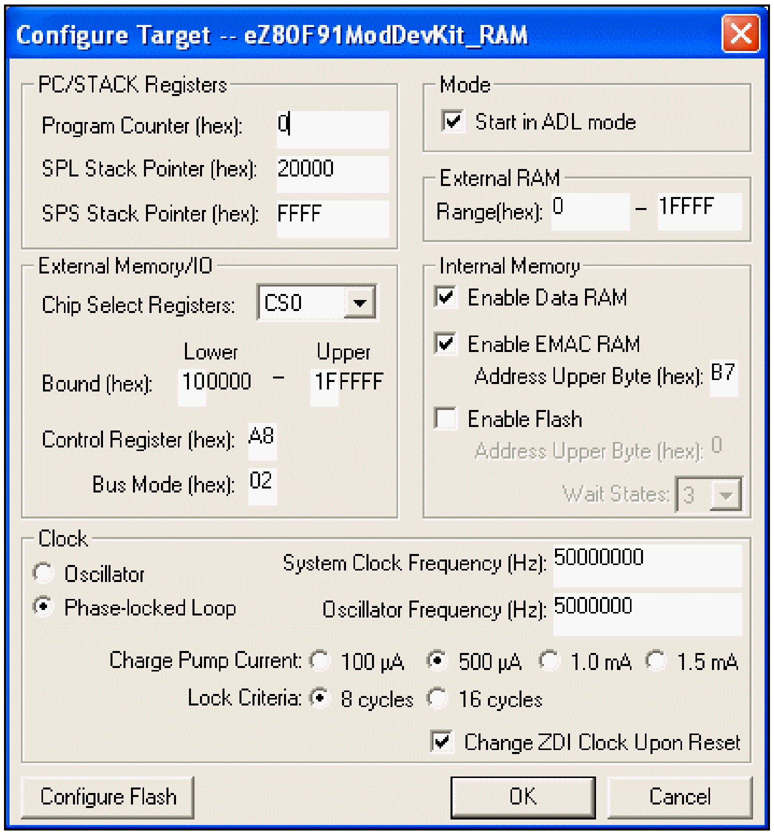
Select the Use Page Erase Before Flashing check box to configure the internal Flash memory of the target hardware to be page-erased. If this check box is not selected, the internal Flash is configured to be mass-erased.

Target

Select the appropriate target from the Target list box.

Setup

Click **Setup** in the Target area to display the Configure Target dialog box.



Configure Target -- eZ80F91ModDevKit_RAM

PC/STACK Registers
Program Counter (hex): 0
SPL Stack Pointer (hex): 20000
SPS Stack Pointer (hex): FFFF

Mode
[✓] Start in ADL mode

External RAM
Range(hex): 0 – 1FFFF

External Memory/IO
Chip Select Registers: CS0
Bound (hex): Lower 100000 – Upper 1FFFFF
Control Register (hex): A8
Bus Mode (hex): 02

Internal Memory
[✓] Enable Data RAM
[✓] Enable EMAC RAM
Address Upper Byte (hex): B7
[ ] Enable Flash
Address Upper Byte (hex): 0
Wait States: 3

Clock
Oscillator [✓ Phase-locked Loop]
System Clock Frequency (Hz): 50000000
Oscillator Frequency (Hz): 5000000
Charge Pump Current: 100 µA [✓ 500 µA] 1.0 mA 1.5 mA
Lock Criteria: [✓ 8 cycles] 16 cycles
[✓] Change ZDI Clock Upon Reset

Figure 64. 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 "New Project" on page 39) or the General page of the Project Settings dialog box (see "Project Settings—General Page" on page 57). Chip select and external bus interface settings are only available for CPUs that support an external bus.

1. Type the address of the first line of code to be executed in the Program Counter (hex) field.
2. Type the upper address of RAM (24-bit address boundary) in the SPL Stack Pointer (hex) field. This option is used while in ADL mode.
3. Type the upper address of RAM (16-bit address boundary) in the SPS Stack Pointer (hex) field. This option is used while in non-ADL mode.
4. For each chip select register (CS0–CS3), do the following:
   – Choose a chip select register from the Chip Select Registers drop-down list box.

     The chip select registers control the type of access, address bounds, and wait state assertion.
   – Enter the lower bound for the chip select register in the Lower Bound (hex) field.
   – Enter the upper bound for the chip select register in the Upper Bound (hex) field.
   – Enter the control register in the Control Register (hex) field.
   – Enter the bus mode in the Bus Mode (hex) field.
5. Select the Start in ADL Mode check box for 24-bit linear addressing. Deselect the check box for 16-bit addressing.

NOTE: This information is used only by the Debugger and does not affect your code initialization. The actual mode needs to be in your source files.

6. Enter the upper and lower addresses for the External RAM Range (hex) fields.

   The external RAM range specifies a range of RAM available to use as “scratch-pad” memory by ZDS for internal Flash memory page erase. The contents of the external RAM range are saved and then restored after a Flash erase operation.

NOTE: If external RAM is not available, the external RAM range *must* be set to the internal RAM address range.

   The eZ80F9x and eZ80L92 require the external RAM range to be in external RAM to support the page erase function.

   If external RAM is not available, the Do Not Erase Info Page check box in the Flash Loader Processor dialog box must be deselected so that ZDS can use mass erase rather than page erase.

   For the eZ80F91 only, you can program an internal-memory-only project if the range in the External RAM Range (hex) fields matches the range in the RAM field in the Address Spaces page (see “Project Settings—Address Spaces Page” on page 89).

7. Select the Enable Data RAM check box to enable the general-purpose internal RAM block. Enter the address in the Address Upper Byte (hex) field.
8. Select the Enable EMAC RAM check box to enable the Ethernet Media Access Controller’s internal RAM. Enter the address in the Address Upper Byte (hex) field.

9. Select the Enable Flash check box if you want to use internal Flash. Enter the address in the Address Upper Byte (hex) field. This shifts Flash and affects the pages displayed in the Flash Loader Processor dialog box. Select the number of wait states from the Wait States drop-down list box. The wait states value is based on the value of the system clock frequency according to the following table:

Wait States	System Clock (MHz)
0	<12
1	12–23.9
2	24–35.9
3	36–47.9
4	48–59.9
5	60–71.9
6	72–84
7	>84

   You can select any wait states value; however, 5, 6, and 7 are not recommended for performance reasons. Based on the currently configured system clock frequency, ZDS II suggests the appropriate wait states value by appending an asterisk to it in the Wait States drop-down list box. The asterisk moves to different values when the system clock frequency is changed in the same dialog box. When the target clock frequency is changed, you must update the wait states value if needed.

10. To use the oscillator, select the Oscillator button and enter the frequency in the System Clock Frequency (Hz) field.

11. To use the phase-locked loop, select the Phase-Locked Loop button, enter the clock frequency in the System Clock Frequency (Hz) field, enter the oscillator frequency in the Oscillator Frequency (Hz) field, select a charge pump current, and select a lock criteria.

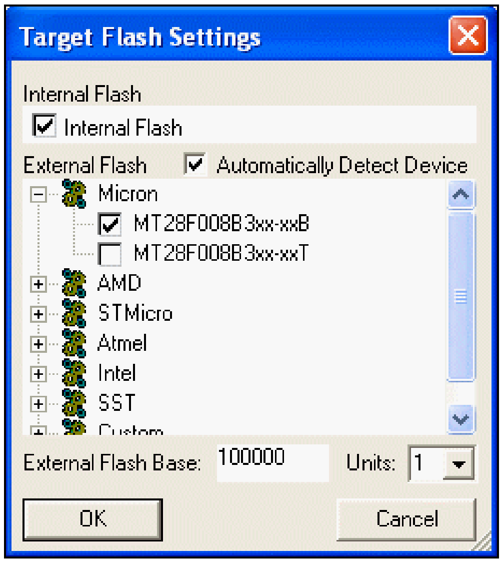
    The eZ80F91 device contains a Phase-Locked Loop (PLL) module, the output of which can be used as the system clock. This allows the application to run at 50 MHz with an oscillator frequency between 1 and 10 MHz. Since the system defaults to using the oscillator upon power-on or hardware reset, the application program must enable and select the PLL as the source of the system clock. This also requires the ZDI clock frequency to change if a debug session is started so that a reliable connection can be maintained. ZDS automatically changes the rate after the first Reset or Go command is invoked and the Change ZDI Clock Upon Reset check box has been selected.

ZDS assumes that the system clock change occurs somewhere after reset and before the `main()` routine. For information about how to set up the charge pump current and lock criteria, see the "Phase Locked Loop" chapter of the eZ80F91 MCU Product Specification (PS0192).

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 Clock window when running the simulator. See "Clock Window" on page 294.

12. Click **Configure Flash**.

    The Target Flash Settings dialog box is displayed.



External Flash Base: 100000
Units: 1

Figure 65. Target Flash Settings Dialog Box

– Select the Internal Flash check box 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.

– If you want to use external Flash, select the Automatically Detect Device check box or 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. 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.

If you select the Automatically Detect Device check box, ZDS II attempts to determine the external Flash device manufacturer and type when Flash is used when downloading code for a debug session. If this attempt is successful, the device type found is used for external Flash operations. If the attempt fails, the external Flash operations default to the manufacturer and device selected for the target. If these values are not supplied and automatic detection fails or is deselected, external Flash operations do not work.

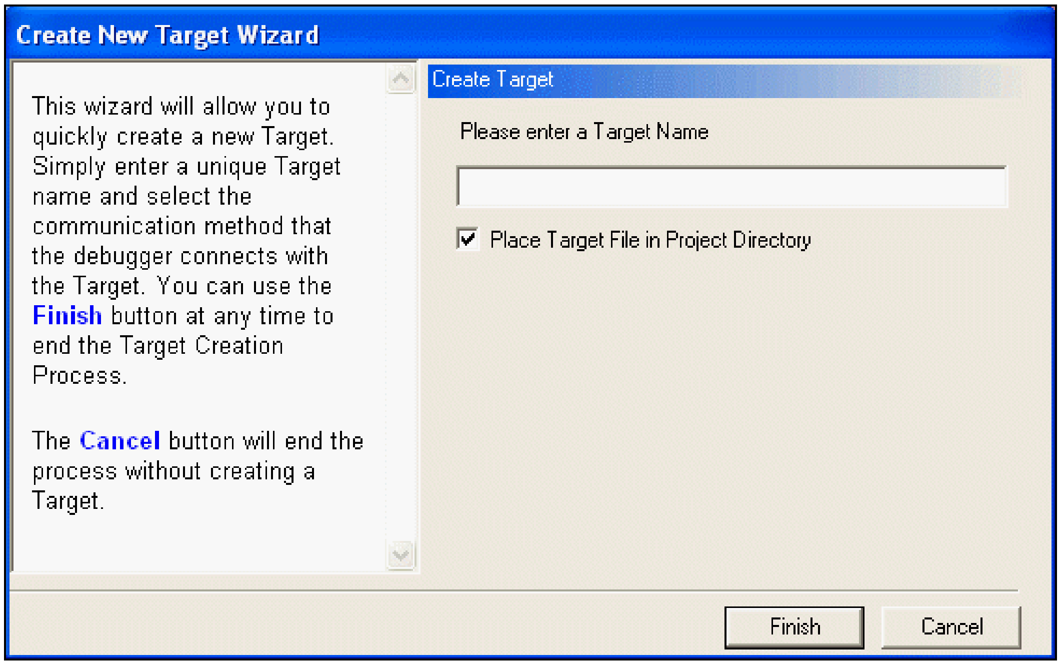
– In the External Flash Base field, type where you want the external Flash to start.
– In the Units drop-down list box, select the number of Flash devices present.

  For example, if you have two devices stacked on top of each other, select **2** in the Units list box.
– Click **OK** to return to the Configure Target dialog box.

13. Click **OK**.

Add

Click **Add** to display the Create New Target Wizard dialog box.


Create New Target Wizard

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.

Create Target

Please enter a Target Name

[✓] Place Target File in Project Directory

Figure 66. Create New Target Wizard Dialog Box

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

Copy

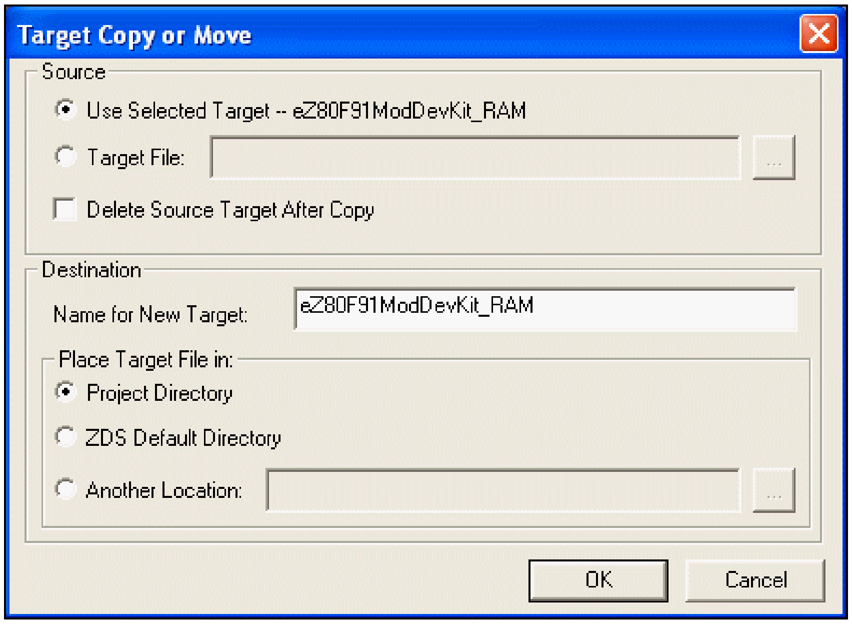


Figure 67. Target Copy or Move Dialog Box

1. Select a target in the Target area of the Debugger page.
2. Click **Copy**.
3. 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 for the name for the new target in the Name for New Target field.
4. Select the Delete Source Target After Copy check box if you do not want to keep the original target.
5. In the Place Target File In area, select the location where you want the new target file saved: in the project directory, ZDS default directory, or another location.
6. Click **OK**.

Delete

Click **Delete** to remove the currently highlighted target

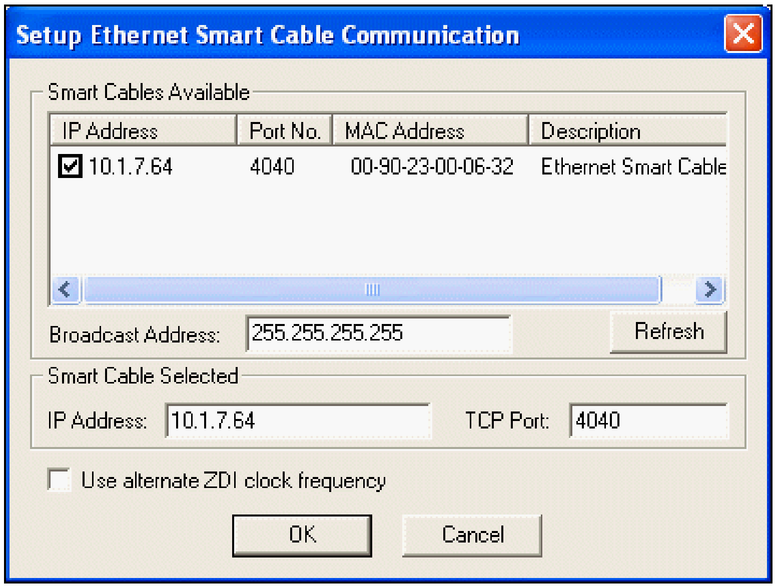
The following message is displayed: "`Delete` *target_name* `Target?`". Click **Yes** to delete the target or **No** to cancel the command.

Debug Tool

Select the appropriate debug tool from the Current drop-down list box.

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

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



IP Address	Port No.	MAC Address	Description
☑ 10.1.7.64	4040	00-90-23-00-06-32	Ethernet Smart Cable

Broadcast Address: 255.255.255.255

IP Address: 10.1.7.64 TCP Port: 4040

Figure 68. Setup Ethernet Smart Cable Communication Dialog Box

– 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 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 uses the default broadcast address if the Broadcast Address field is empty.

– 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.

– Type the port number in the TCP Port field.

– Select the Use Alternate ZDI Clock Frequency check box if you want to override the default ZDI clock frequency. The alternate ZDI clock frequency should only be used if a reliable connection cannot be established with the default ZDI clock frequency. The default ZDI clock frequency will be used if an alternate ZDI clock frequency is not available for the given system clock frequency.

- Click **OK**.

- If you select **SerialSmartCable** and click **Setup** in the Debug Tool area, the Setup Serial Communication dialog box is displayed.

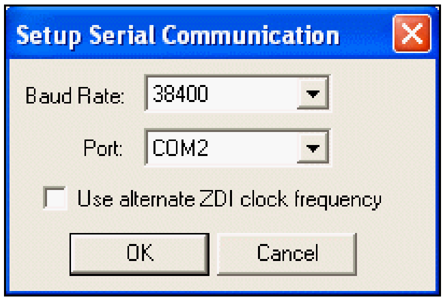


Figure 69. Setup Serial Communication Dialog Box

- Use the Baud Rate drop-down list box to select the appropriate baud rate: 19200, 38400, 57600, or 115200. The default is 57600.
- Select the host COM port connected to your target.

  ZDS II sets the COM port settings for data, parity, stop, and flow control. You do not need to set these.
- Select the Use Alternate ZDI Clock Frequency check box if you want to use the alternate ZDI clock frequency.

  In previous versions of the ZDS II, you were required to choose the appropriate ZDI clock frequency. In releases after 4.7.2, that option has been replaced with the system clock and oscillator frequency so that ZDS can choose the appropriate ZDI communication rate(s).

  ZDS uses the information in the following table to make the ZDI clock frequency selection:

System Clock Frequency (MHz)	Default ZDI Clock Frequency (MHz)	Alternate ZDI Clock Frequency (MHz)
2–3.9	1	1
4–6.9	2	1
7–7.9	2	2
8–12.9	4	2
13–15.9	4	4
16–24.9	8	4
25–50	8	8

The minimum that the system clock frequency can reliably support is 5 MHz. Running the system clock at a lower frequency might result in unreliable operation.

If a reliable connection cannot be established and the system clock frequency is within an overlapping area, you can override the default choice by selecting the Use Alternate ZDI Clock Frequency check box. Selecting the alternate system clock table changes the ZDI clock frequency used by ZDS. For instance, if the system clock is 20 MHz, the 4-MHz ZDI clock rate might prove to be more reliable then the 8-MHz rate.

ZDS assumes that the system clock change occurs somewhere after reset and before the `main()` routine.

– Click **OK**.

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

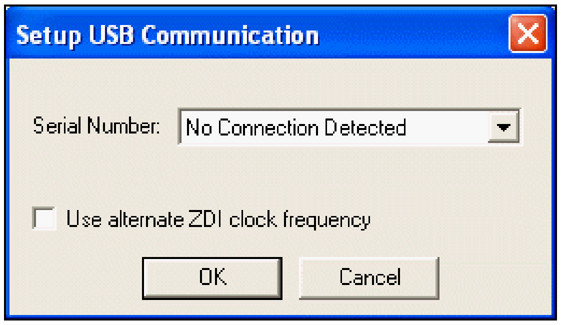


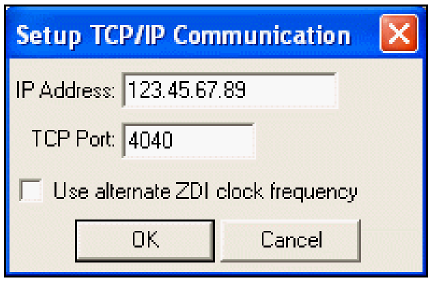
Figure 70. Setup USB Communication Dialog Box

– Use the Serial Number drop-down list box to select the appropriate serial number.

– Select the Use Alternate ZDI Clock Frequency check box if you want to override the default ZDI clock frequency. The alternate ZDI clock frequency should only be used if a reliable connection cannot be established with the default ZDI clock frequency. The default ZDI clock frequency will be used if an alternate ZDI clock frequency is not available for the given system clock frequency.

– Click **OK**.

- If you select **ZPAKII** and click **Setup** in the Debug Tool area, the Setup TCP/IP Communication dialog box is displayed.



IP Address: 123.45.67.89
TCP Port: 4040
☐ Use alternate ZDI clock frequency

Figure 71. Setup TCP/IP Communication Dialog Box

– Type the IP address in the IP Address field.
– Type the port number in the TCP Port field.
– Select the Use Alternate ZDI Clock Frequency check box if you want to use the alternate ZDI clock frequency.

  In previous versions of the ZDS II, you were required to choose the appropriate ZDI clock frequency. In releases after 4.7.2, that option has been replaced with the system clock and oscillator frequency so that ZDS can choose the appropriate ZDI communication rate(s).

  ZDS uses the information in the following table to make the ZDI clock frequency selection:

System Clock Frequency (MHz)	Default ZDI Clock Frequency (MHz)	Alternate ZDI Clock Frequency (MHz)
2–3.9	1	1
4–6.9	2	1
7–7.9	2	2
8–12.9	4	2
13–15.9	4	4
16–24.9	8	4
25–50	8	8

NOTE: The minimum that the system clock frequency can reliably support is 5 MHz. Running the system clock at a lower frequency might result in unreliable operation.

If a reliable connection cannot be established and the system clock frequency is within an overlapping area, you can override the default choice by selecting the Use Alternate ZDI Clock Frequency check box. Selecting the alternate system clock table changes the ZDI clock frequency used by ZDS. For instance, if the system clock is 20 MHz, the 4-MHz ZDI clock rate might prove to be more reliable then the 8-MHz rate.

ZDS assumes that the system clock change occurs somewhere after reset and before the `main()` routine.

– Click **OK**.

Export Makefile

The Export Makefile command exports a buildable project in external make file format. To do this, complete the following procedure:

1. From the Project menu, select **Export Makefile**.

   The Save As dialog box is displayed.

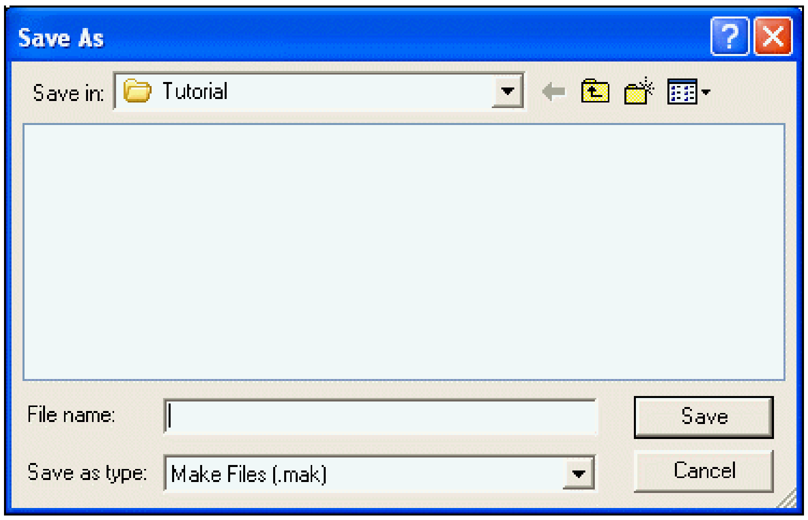


Figure 72. Save As Dialog Box

2. Use the Save In drop-down list box to navigate to the directory where you want to save your project.

   The default location is in your project directory.

3. Type the make file name in the File Name field.

   You do not have to type the extension `.mak`. The extension is added automatically.

4. Click **Save**

   The project is now available as an external make file.

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 the following commands:

- "Compile" on page 107
- "Build" on page 107
- "Rebuild All" on page 107
- "Stop Build" on page 107
- "Clean" on page 107
- "Update All Dependencies" on page 107
- "Set Active Configuration" on page 108
- "Manage Configurations" on page 109

Compile

Select **Compile** from the Build menu to compile or assemble the active file in the Edit window.

Build

Select **Build** from the Build menu to build your project. The build compiles and/or assembles any files that have changed since the last build and then links the project.

Rebuild All

Select **Rebuild All** from the Build menu to rebuild *all* the files in your project. This option also links the project.

Stop Build

Select **Stop Build** from the Build menu to stop a build in progress.

Clean

Select **Clean** from the Build menu to remove intermediate build files.

Update All Dependencies

Select **Update All Dependencies** from the Build menu to update your source file dependencies.

Set Active Configuration

You can use the Select Configurations 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.

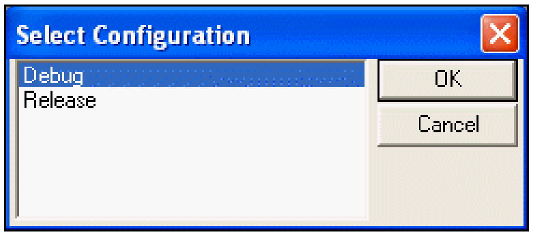


Figure 73. Select Configuration Dialog Box

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

There are two standard configuration settings:

- Debug

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

- Release

  This configuration contains all 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 the setting changes that need to 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 “Manage Configurations” on page 109.

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

- Use the Select Configuration dialog box.

  See “Set Active Configuration” on page 108

- Use the Build toolbar.

  See “Select Build Configuration List Box” on page 18.

Use the Project Settings dialog box to modify build configuration settings. See “Settings” on page 55.

Manage Configurations

For your specific needs, you can add or copy different configurations for your projects. To add a customized configuration, do the following:

1. From the Build menu, select **Manage Configurations**.

   The Manage Configurations dialog box is displayed.

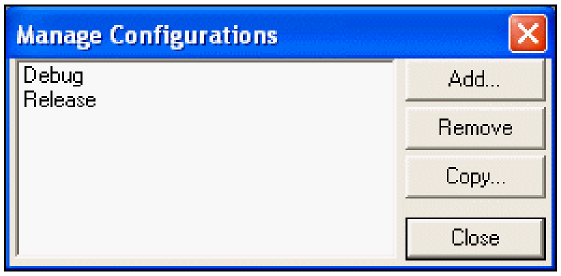


Figure 74. Manage Configurations Dialog Box

2. From the Manage Configurations dialog box, click **Add**.

   The Add Project Configuration dialog box is displayed.

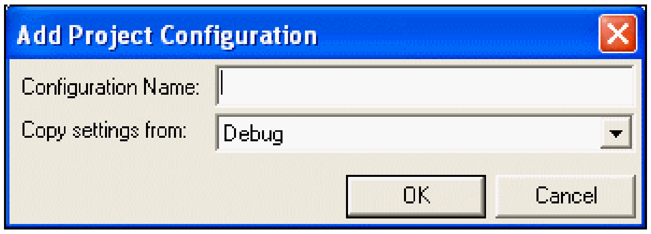


Figure 75. Add Project Configuration Dialog Box

3. Type the name of the new configuration in the Configuration Name field.
4. Select a similar configuration from the Copy Settings From drop-down list box.
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 list box on the Build toolbar.

   Now that you have created a blank template, you are ready to select the settings for this new configuration.

7. 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**.

To copy the settings from an existing configuration to an existing configuration, do the following:

1. From the Build menu, select **Manage Configurations**.

   The Manage Configurations dialog box is displayed.

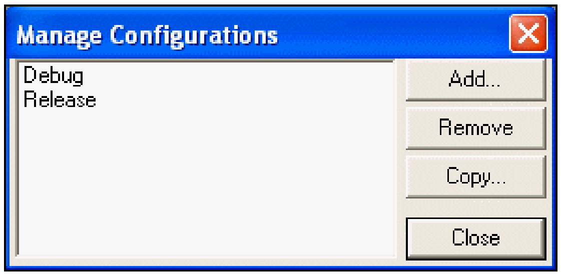
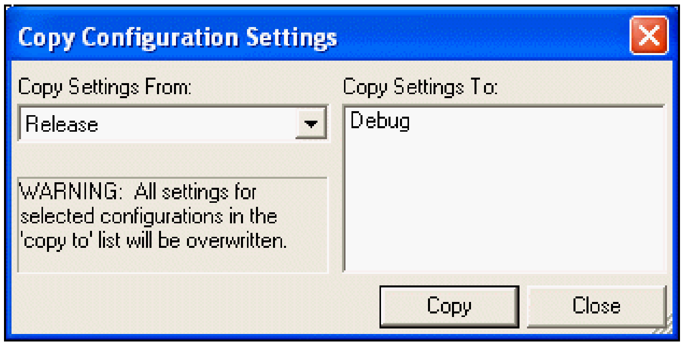


Figure 76. Manage Configurations Dialog Box

2. From the Manage Configurations dialog box, click **Copy**.

   The Copy Configuration Settings dialog box is displayed.



WARNING: All settings for selected configurations in the 'copy to' list will be overwritten.

Figure 77. Copy Configuration Settings Dialog Box

3. Select the configuration with the desired settings from the Copy Settings From drop-down list box.
4. Highlight the configuration(s) in the Copy Settings To field that you want to change.
5. Click **Copy**.

Debug Menu

From the Debug menu, you can access the following functions for the debugger:

- "Connect to Target" on page 111
- "Download Code" on page 112
- "Verify Download" on page 112
- "Stop Debugging" on page 112
- "Reset" on page 112
- "Go" on page 113
- "Run to Cursor" on page 113
- "Break" on page 114
- "Step Into" on page 114
- "Step Over" on page 114
- "Step Out" on page 114
- "Set Next Instruction" on page 114

NOTE: For more information on the debugger, see "Using the Debugger" on page 290.

Connect to Target

The Connect to Target command starts a debug session using the following process:

1. Initializes the communication to the target hardware.
2. Resets the device.
3. Configures the device using the settings in the Configure Target dialog box.
4. Configures and executes the debugger options selected in the Debugger tab of the Options dialog box. The following options are ignored if selected:
   – Reset to Symbol 'main' (Where Applicable) check box
   – Verify File Downloads—Read After Write check box
   – Verify File Downloads—Upon Completion check box

This command does not download the software. Use this command 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. This command is available only when not in Debug mode.

For the Serial Smart Cable, ZDS II performs an external target reset and reconfigures PC and SPL as specified in the Configure Target dialog box.

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. Starting in version 4.10.0, the Download Code command can also program Flash memory. A page erase is done instead of a mass erase for both internal and external Flash memory. Use this command anytime during a debug session. This command is not enabled when the debug tool is the simulator.

NOTE: The current code on the target is overwritten.

If ZDS II is not in Debug mode when the Download Code command is selected, the following process is executed:

1. Initializes the communication to the target hardware.
2. Resets the device with a hardware reset by driving ZDI pin #2 low.
3. Configures the device using the settings in the Configure Target dialog box.
4. Downloads the program.
5. Issues a software reset through the ZDI serial interface.
6. Configures and executes the debugger options selected in the Debugger tab of the Options dialog box. If it is a C project, ZDS II resets to the main function if it is found.

If ZDS II is already in Debug mode when the Download Code command is selected, the following process is executed:

1. Resets the device using a software reset.
2. Downloads the program.

You might need to reset the device before execution because the program counter might have been changed after the download.

Verify Download

Select **Verify Download** from the Debug menu to determine download correctness by comparing the 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 ZDS II is not in Debug mode, the Reset command starts a debug session using the following process:

1. Initializes the communication to the target hardware.
2. Resets the device.
3. Configures the device using the settings in the Configure Target dialog box.
4. Downloads the program.
5. Configures and executes the debugger options selected in the Debugger tab of the Options dialog box. If it is a C project, ZDS II resets to the main function if it is found.

If ZDS II is already in Debug mode, the Reset command uses the following process:

1. ZDS II performs a soft reset in which just PC and SPL are reconfigured as specified in the Configure Target dialog box.
2. Configures the device using the settings in the Configure Target dialog box.
3. If files have been modified, ZDS II asks, "Would you like to rebuild the project?" before downloading the modified program. If there has been no file modification, the code is not reloaded.

The Serial Smart Cable performs an external target reset.

Go

Select **Go** from the Debug menu to execute project code from the current program counter.

If not in Debug mode when the Go command is selected, the following process is executed:

1. Initializes the communication to the target hardware.
2. Resets the device.
3. Configures the device using the settings in the Configure Target dialog box.
4. Downloads the program.
5. Configures and executes the debugger options selected in the Debugger tab of the Options dialog box. If it is a C project, ZDS II resets to the main function if it is found.
6. Executes the program from the reset location.

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, calculate a file checksum, update the firmware, and customize the appearance of the eZ80Acclaim! developer's environment.

The Tools menu has the following options:

- "Flash Loader" on page 115
- "Calculate File Checksum" on page 120
- "Firmware Upgrade" on page 120
- "Customize" on page 121
- "Options" on page 122

Flash Loader

Use the following procedure to program internal and external Flash for the eZ80® and eZ80Acclaim!:

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 "Setup" on page 96), do the following:
   – Ensure that the ROM/RAM Chip Select Registers fields define the target RAM properly.
   – Make certain the Enable Flash check box is selected. Enter the address in the Address Upper Byte (hex) field. This shifts Flash and affects the pages displayed in the Flash Loader Processor dialog box. Select the number of wait states from the Wait States drop-down list box. The wait states value is based on the value of the system clock frequency according to the following table:

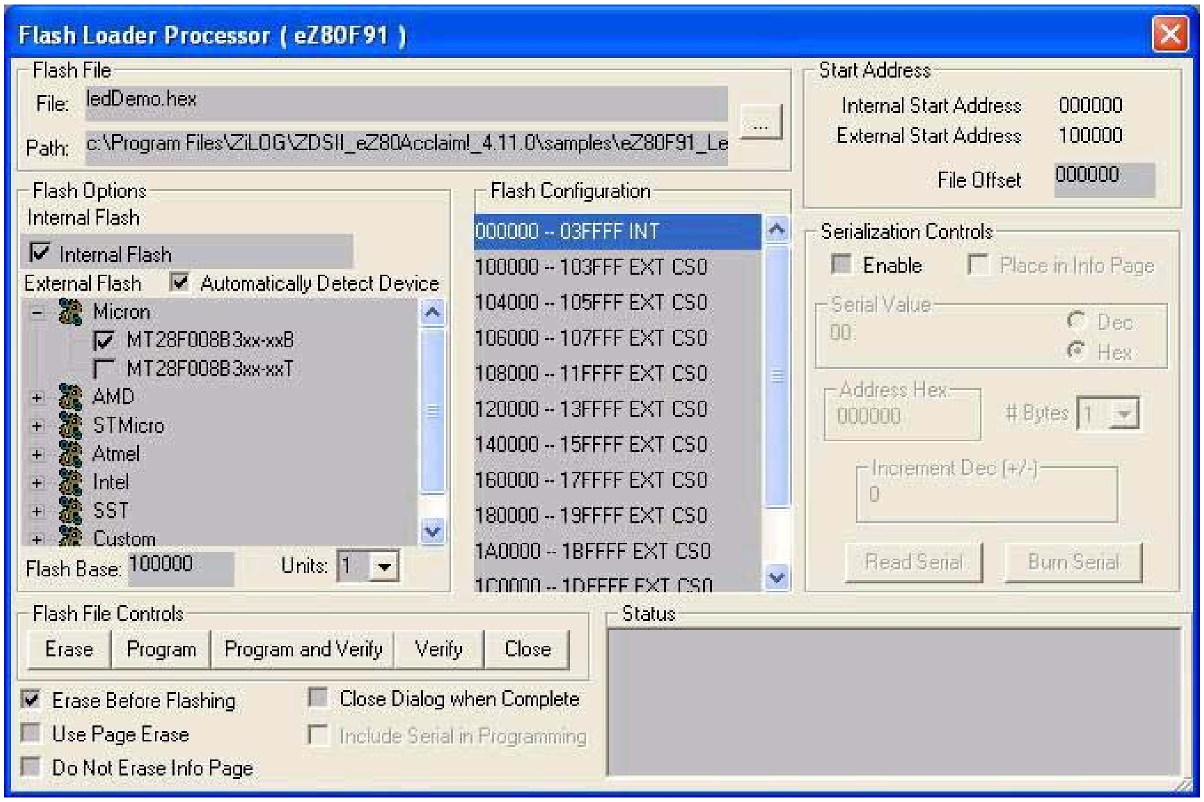
Wait States	System Clock (MHz)
0	<12
1	12–23.9
2	24–35.9
3	36–47.9
4	48–59.9
5	60–71.9
6	72–84
7	>84

   You can select any wait states value; however, 5, 6, and 7 are not recommended for performance reasons. Based on the currently configured system clock frequency, ZDS II suggests the appropriate wait states value by appending an asterisk to it in the Wait States drop-down list box. The asterisk moves to different values when the system clock frequency is changed in the same dialog box. When the target clock frequency is changed, you must update the wait states value if needed.

Caution: The Flash Loader downloads a selected hex file to target Flash. The Flash Loader uses internal RAM if it is available, and the ROM/RAM Chip Select Registers fields in the Configure Target dialog box define where RAM is for a CPU that does not have internal RAM (such as the eZ80L92). The defined RAM addresses must not overlap the defined Flash addresses.

3. 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 with the appropriate Flash target options for the selected CPU.


Flash Loader Processor (eZ80F91)

Flash File
File: ledDemo.hex
Path: c:\Program Files\ZiLOG\ZDSII_eZ80Acclaim!_4.11.0\samples\eZ80F91_Le

Start Address
Internal Start Address 000000
External Start Address 100000
File Offset 000000

Flash Options
Internal Flash
[✓] Internal Flash
External Flash [✓] Automatically Detect Device
- Micron
  [✓] MT28F008B3xx-xxB
  [ ] MT28F008B3xx-xxT
+ AMD
+ STMicro
+ Atmel
+ Intel
+ SST
+ Custom
Flash Base: 100000 Units: 1

Flash Configuration
000000 -- 03FFFF INT
100000 -- 103FFF EXT CS0
104000 -- 105FFF EXT CS0
106000 -- 107FFF EXT CS0
108000 -- 11FFFF EXT CS0
120000 -- 13FFFF EXT CS0
140000 -- 15FFFF EXT CS0
160000 -- 17FFFF EXT CS0
180000 -- 19FFFF EXT CS0
1A0000 -- 1BFFFF EXT CS0
1C0000 -- 1DFFFF EXT CS0

Serialization Controls
[ ] Enable [ ] Place in Info Page
Serial Value 00 ( ) Dec (•) Hex
Address Hex 000000 # Bytes 1
Increment Dec (+/-) 0
Read Serial Burn Serial

Flash File Controls
Erase Program Program and Verify Verify Close

Status

[✓] Erase Before Flashing [ ] Close Dialog when Complete
[ ] Use Page Erase [ ] Include Serial in Programming
[ ] Do Not Erase Info Page

Figure 78. Flash Loader Processor Dialog Box

4. Click on the Browse button (...) to navigate to the hex file to be flashed.
5. Select the Flash targets in the Flash Options area.

NOTE: The Flash Options displayed in the Flash Loader Processor dialog box depend on the CPU you selected in the New Project dialog box (see "New Project" on page 39) or the General page of the Project Settings dialog box (see "Project Settings—General Page" on page 57).

You must select at least one of the following check boxes 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 the internal Flash is selected, the address range is displayed in the Flash Configuration area with an INT extension.

– External Flash

  If you want to use external Flash, select the Automatically Detect Device check box or 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 is 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.

  If you select the Automatically Detect Device check box, the Flash Loader queries the target to attempt to determine the external Flash manufacturer and device on the hardware. If the detection succeeds, the tree selection for the manufacturer and device is changed to match the hardware values. If the attempt fails, the external Flash operations default to the manufacturer and device selected for the target. If these values are not supplied and automatic detection fails or is deselected, external Flash operations do not work. While this check box is selected, any attempt to change the manufacturer and device selections results in an attempt to determine which external Flash device is in use and (upon success) an update to the tree selections.

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.

6. In the Flash Base field, type where you want the Flash programming to start.

   The Flash base defines the start of external Flash.

7. In the Units drop-down list box, 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.

8. Select the pages to erase before flashing in the Flash Configuration area.

   Pages that are grayed out are not available.

   To select all the pages, use the right-click menu or delete the file name in the File field.

9. In the File Offset field, type the appropriate offset values to offset the base address of the hex file.

NOTE: The hex file address is shifted by the offset defined in the Start Address area. You need to allow for the shift in any defined jump table index. This offset value also shifts the erase range for the Flash.

10. Select the Erase Before Flashing check box to erase all Flash memory before writing the hex file to Flash memory.

Caution You can also delete the Flash memory by clicking **ERASE**. Clicking **ERASE** deletes only the pages that are selected.

11. Select the Use Page Erase check box if you want the internal Flash to be page-erased. Deselect this check box if you want the internal Flash to be mass-erased.
12. Select the Do Not Erase Info Page check box to keep the data in the Info page.
13. Select the Close Dialog When Complete check box to close the dialog box after writing the hex file to Flash memory.
14. If you want to use the serialization feature or want to check a serial number that has already been programmed at an address, see “Serialization” on page 119.
15. 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.
16. Verify the Flash memory by clicking **Verify**.

    When you click **Verify**, 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.

NOTE: ZiLOG also provides a target-based external Flash Loader utility for reprogramming Flash memory. It is designed primarily for field upgrades to stand-alone systems. For more information about this external Flash Loader utility, refer to the External Flash Loader Product User Guides (PUG0016 and PUG0018).

Serialization

The general procedure to write a serial number to a Flash device involves the following steps:

1. Choose a location for the serial number inside or outside of the address range defined in the hex file.

NOTE: The serial number must be written to a location that is not being written to by the hex file.

2. Erase the Flash device.
3. Write the hex file to the Flash device and then write the serial number

   or

   write the serial number to the Flash device and then write the hex file.

Use the following procedure if you want to use the serialization feature:

1. Select the Include Serial in Programming check box.

   This option programs the serial number after the selected hex file has been written to Flash.
2. Select the Enable check box.
3. Type 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.
4. Type the location you want the serial number located in the Address Hex field.
5. Select the number of bytes that you want the serial number to occupy in the # Bytes drop-down list box.
6. Type 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, type in 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.
7. Select the Erase Before Flashing check box.

   This option erases the Flash before writing the serial number.
8. 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 then write the serial number.

If you want to check a serial number that has already been programmed at an address, do the following:

1. Select the Enable check box.
2. Type the address that you want to read in the Address Hex field.

3. Select the number of bytes to read from # Bytes drop-down list box.
4. Click **Read Serial** to check the serial number for the current device.

Calculate File Checksum

Use the following procedure to calculate the file checksum:

1. Select **Calculate File Checksum** from the Tools menu.

   The Calculate Checksum dialog box is displayed.

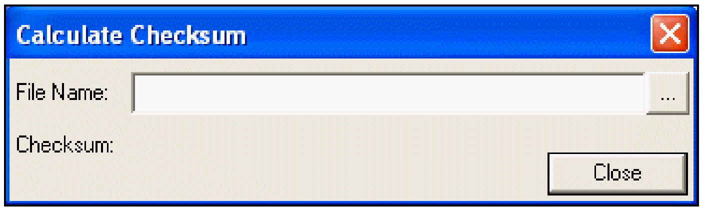
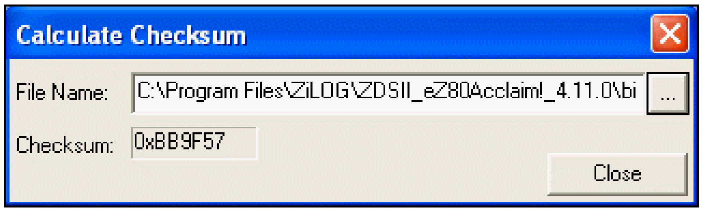


Figure 79. Calculate Checksum Dialog Box

2. Click on 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.



File Name: C:\Program Files\ZiLOG\ZDSII_eZ80Acclaim!_4.11.0\bi

Checksum: 0xBB9F57

Figure 80. Calculate Checksum Dialog Box

3. Click **Close**.

Firmware Upgrade

NOTE: This command is available only when a supporting debug tool is selected ("Debug Tool" on page 102).

- USB Smart Cable

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

- Serial Smart Cable

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

- Ethernet Smart Cable

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

- ZPAK II

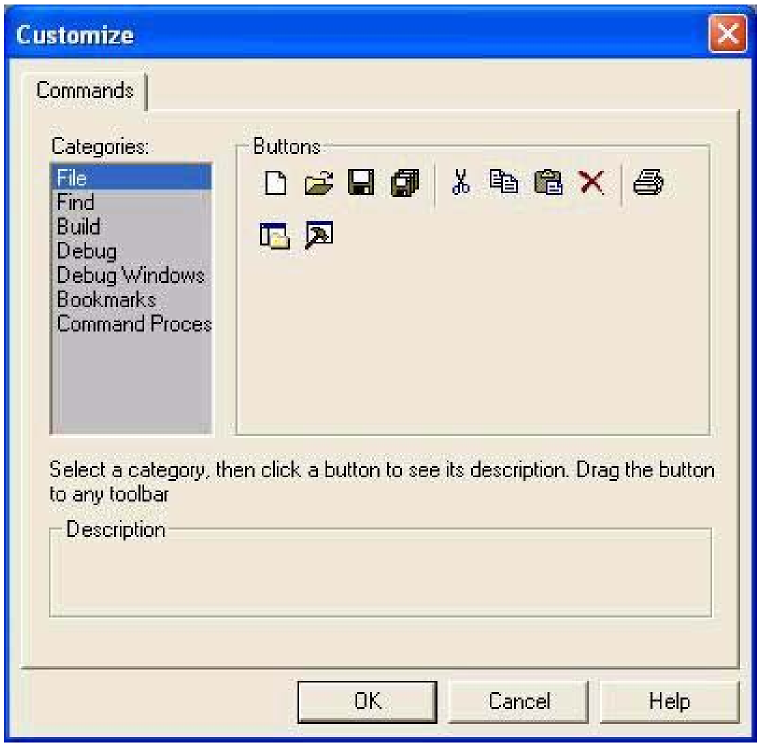
  <ZDS Installation Directory>`\bin\firmware\ZPAKII\`
  `upgrade information.txt`

Customize

The Customize dialog box lets you modify the following items:

- "File Toolbar" on page 16
- "Find Toolbar" on page 21
- "Build Toolbar" on page 18
- "Debug Toolbar" on page 23
- "Debug Windows Toolbar" on page 27
- "Bookmarks Toolbar" on page 22
- "Command Processor Toolbar" on page 22

To see a description of each button on the toolbars, highlight the icon as shown in the following figure.



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

Figure 81. Customize Dialog Bo

To customize a toolbar, use the following procedure:

1. Select **Customize** from the Tools menu.

   The Customize dialog box is displayed.

2. Select a toolbar in the Categories area.
3. Drag buttons from the Buttons area to any toolbar.

   To see a description of each toolbar button, highlight the icon.

4. Click **OK** to apply your changes or **Cancel** to close the dialog box without making any changes.

NOTE: You cannot change the buttons on the default toolbars.

Options

The Options dialog box contains the following tabs:

- “Options—General Tab” on page 123
- “Options—Editor Tab” on page 124
- “Options—Debugger Tab” on page 127

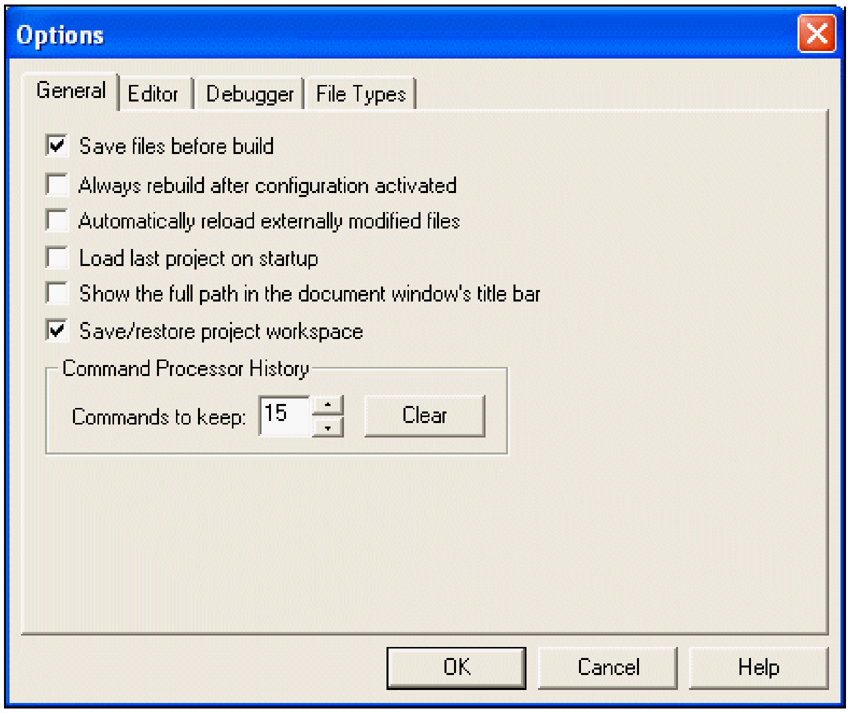
- "Options—File Types Tab" on page 128

Options—General Tab

The General tab contains the following check boxes:

- Select the Save Files Before Build check box to save files before you build. This option is selected by default.
- Select the Always Rebuild After Configuration Activated check box 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 configuration 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 check box to automatically reload externally modified files. This option is not selected by default.
- Select the Load Last Project on Startup check box 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 check box to add the complete path to the name of each file open in the Edit window.
- Select the Save/Restore Project Workspace check box 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

Command Processor History

Commands to keep: 15 Clear

Figure 82. Options Dialog Box—General Tab

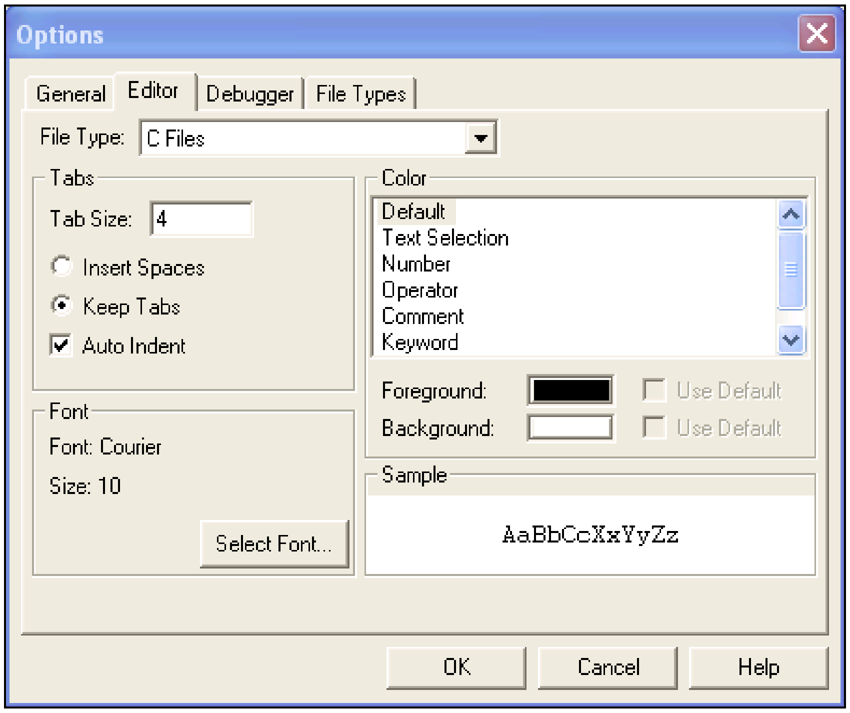
Options—Editor Tab

Use the Editor tab to change the default settings of the editor for your assembly, C, and default files:

1. From the Tools menu, select **Options**.

   The Options dialog box is displayed.

2. Click on the Editor tab.



File Type: C Files

Tabs
Tab Size: 4
Insert Spaces
● Keep Tabs
✓ Auto Indent

Font
Font: Courier
Size: 10

Color
Default
Text Selection
Number
Operator
Comment
Keyword

Foreground: ■ Use Default
Background: □ Use Default

Sample
AaBbCcXxYyZz

Figure 83. Options Dialog Box—Editor Tab

3. Select a file type from the File Type drop-down list box.

   You can select C files, assembly files, or other files and windows.

4. In the Tabs area, do the following:
   – 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 check box if you want the IDE to automatically add indentation to your files.

5. If you want to change the color for any of the items in the Color list box, click the item, make sure the Use Default check boxes are not selected, and then click on the color in the Foreground or Background field to display the Color dialog box (see the following figure). If you want to use the default foreground or background color for the selected item, enable the Use Default check box next to the Foreground or Background check box (see the preceding figure).

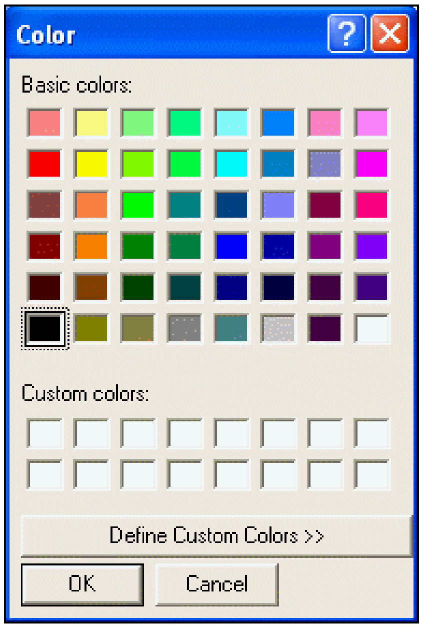


Figure 84. Color Dialog Box

6. Click **OK** to close the Color dialog box.
7. To change the default font and font size, click **Select Font**.

   The Font dialog box is displayed.

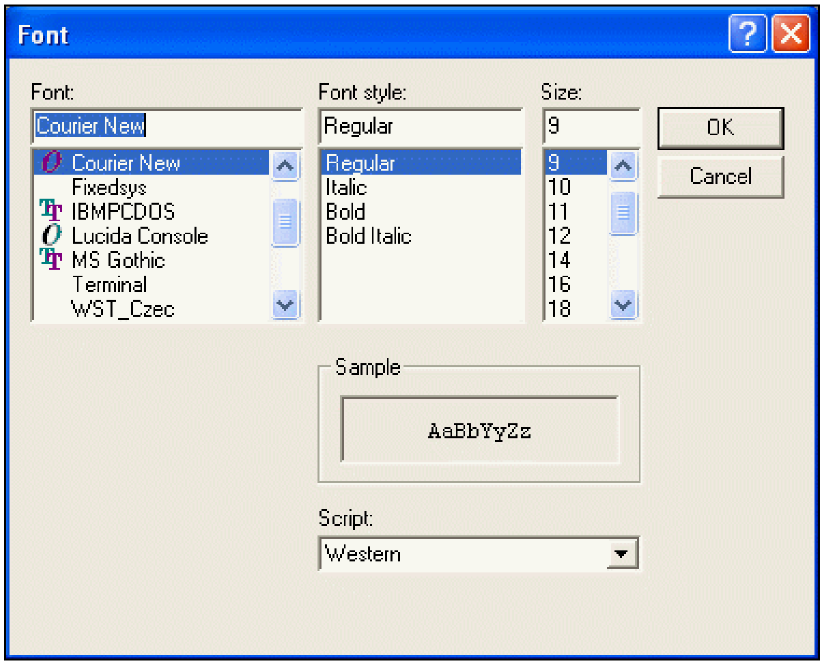


Figure 85. Font Dialog Box

You can change the font, font style, font size, and script style.

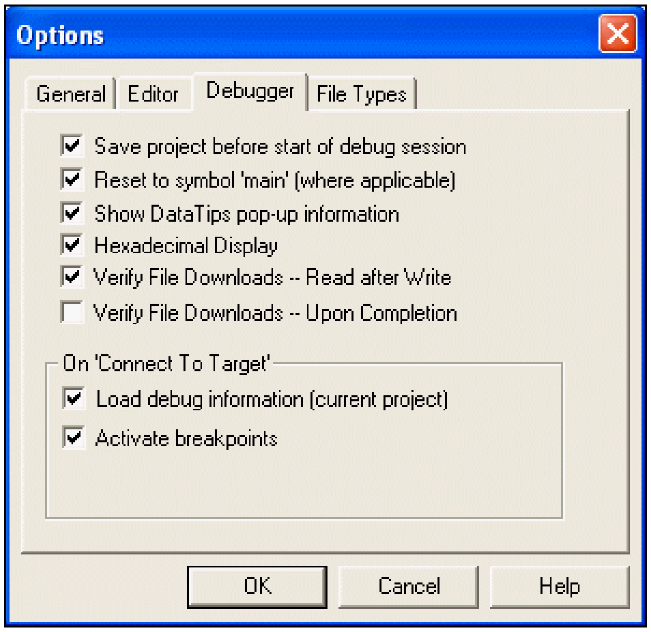
8. Click **OK** to close the Font dialog box.
9. Click **OK** to close the Options dialog box.

Options—Debugger Tab

The Debugger tab contains the following check boxes:

- Select the Save Project Before Start of Debug Session check box to save the current project before entering the Debug mode. This option is selected by default.
- Select the Reset to Symbol 'main' (Where Applicable) check box to skip the startup (boot) code and start debugging at the main function for a project that includes a C language main function. When this check box is selected, a user reset (clicking the Reset button on the Build and Debug toolbars, selecting **Reset** from the Debug submenu, or using the `reset` script command) results in the program counter (PC) pointing to the beginning of the main function. When this check box is not selected, a user reset results in the PC pointing to the first line of the program (the first line of the startup code).
- When the Show DataTips Pop-Up Information check box is selected, you can hold the mouse cursor over a variable in a C file in the Edit window in Debug mode, and the value is displayed.

- Select the Hexadecimal Display check box to change the values in the Watch and Locals windows to hexadecimal format. Deselect the check box to change the values in the Watch and Locals windows to decimal format.
- Select the Verify File Downloads—Read After Write check box to perform a read after write verify of the Code Download function. Selecting this check box increases the time taken for the code download to complete.
- Select the Verify File Downloads—Upon Completion check box to verify the code that you downloaded after it has downloaded.
- Select the Load Debug Information (Current Project) check box to load the debug information for the currently open project when the Connect to Target command is executed (from the Debug submenu or from the Connect to Target button). This option is selected by default.
- Select the Activate Breakpoints check box for the breakpoints in the current project to be active when the Connect to Target command is executed (from the Debug submenu or from the Connect to Target button). This option is selected by default.



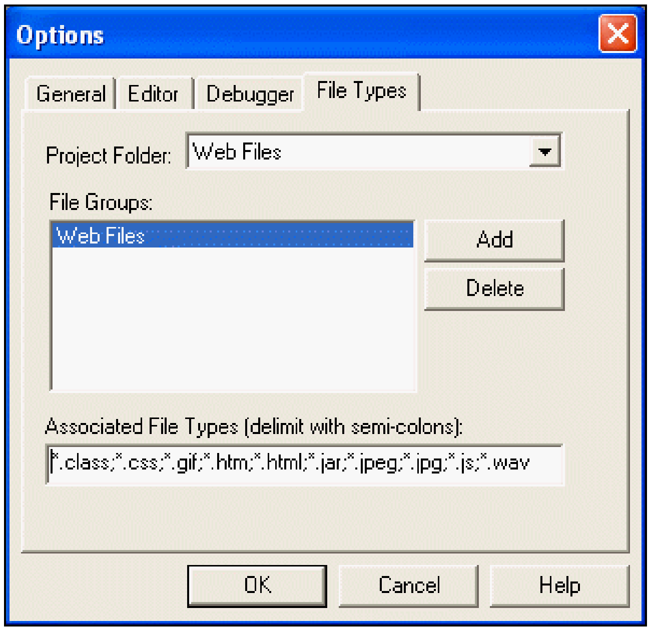
[✓] Save project before start of debug session
[✓] Reset to symbol 'main' (where applicable)
[✓] Show DataTips pop-up information
[✓] Hexadecimal Display
[✓] Verify File Downloads -- Read after Write
[ ] Verify File Downloads -- Upon Completion

On 'Connect To Target'
[✓] Load debug information (current project)
[✓] Activate breakpoints

Figure 86. Options Dialog Box—Debugger Tab

Options—File Types Tab

Use the File Types tab to add a new directory for specified file types in the Project Workspace window.



Project Folder: Web Files

File Groups: Web Files

Associated File Types (delimit with semi-colons): *.class;*.css;*.gif;*.htm;*.html;*.jar;*.jpeg;*.jpg;*.js;*.wav

Figure 87. Options Dialog Box—File Types Tab

1. Click **Add**.

   The Add File Group dialog box is displayed.

2. In the Group Name field, type the name of the new directory.
3. Click **OK**.

   The new file group appears in the File Groups field.

4. Select the new file group name.
5. In the Associated File Types field, type the file extensions to store in the new directory.

   Use a comma to separate the file types.

6. Click **OK**.

Window Menu

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

The Window menu contains the following options:

- "New Window" on page 130
- "Close" on page 130

- “Close All” on page 130
- “Cascade” on page 130
- “Tile” on page 130
- “Arrange Icons” on page 130

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 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 at once.

Arrange Icons

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

Help Menu

The Help menu contains the following options:

- “Help Topics” on page 130
- “Technical Support” on page 130
- “About” on page 130

Help Topics

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

Technical Support

Select **Technical Support** to access ZiLOG’s Technical Support web site.

About

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

SHORTCUT KEYS

The following sections list the shortcut keys for the ZiLOG Developer Studio II:

- “File Menu Shortcuts” on page 131
- “Edit Menu Shortcuts” on page 131
- “Project Menu Shortcuts” on page 132
- “Build Menu Shortcuts” on page 132
- “Debug Menu Shortcuts” on page 132

File Menu Shortcuts

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

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+Alt+L	To save all files in the project.
Print	Ctrl+P	To print a file.

Edit Menu Shortcuts

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

Option	Shortcut	Description
Undo	Ctrl+Z	To undo the last command or action you performed.
Redo	Ctrl+Y	To redo the last command or action you performed.
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 like 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.

Option	Shortcut	Description
Toggle Bookmark	Ctrl+F2	To insert a bookmark in the active file for the line where your cursor is located or to remove the bookmark for the line where your cursor is located.
Next Bookmark	F2	To position the cursor at the line where 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 where 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

There is one shortcut for the options on the Project menu.

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

Build Menu Shortcuts

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

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.

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.

Option	Shortcut	Description
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.

Using the ANSI C-Compiler

The eZ80Acclaim! C-Compiler is a conforming freestanding 1989 ANSI C implementation with some exceptions. These exceptions are described in "ANSI Standard Compliance" on page 164. 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 eZ80Acclaim! compiler release supports more of the standard library than is required of a freestanding implementation, as listed in "Run-Time Library" on page 154.

The eZ80Acclaim! C-Compiler supports language extensions for the easy programming of the eZ80Acclaim! processor architecture. The language extensions are described in "Language Extensions" on page 135.

The following sections describe the various features of the eZ80Acclaim! C-Compiler:

The eZ80Acclaim! C-Compiler is optimized for embedded applications in which execution speed and code size are crucial.

LANGUAGE EXTENSIONS

To provide additional support for some frequently used features of embedded applications, the eZ80Acclaim! C-Compiler implements the following extensions to the ANSI C standard:

- "Interrupt Support" on page 135

  The eZ80Acclaim! CPU supports various interrupts. The C-Compiler provides language extensions to designate a function as an interrupt service routine and provides features to set each interrupt vector.

- "Inline Assembly" on page 143

  The C-Compiler provides directives for embedding assembly instructions and directives into the C program.

- "Placement Directives" on page 142

  The placement directives allow users to place objects at specific hardware addresses and align objects at a given alignment.

- "fract Keyword" on page 144

  The C-Compiler supports numerical types of several sizes that are used for fractional fixed-point representations of values in the range –1 to 1.

- "Char and Short Enumerations" on page 146

  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`.

- "Supported New Features from the 1999 Standard" on page 147

  The eZ80Acclaim! 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.

Interrupt Support

To support interrupts, the eZ80Acclaim! C-Compiler provides two features. These are described in the following sections along with some practical information on how to use interrupts in your C application for the eZ80Acclaim!:

- "interrupt Keyword" on page 135
- "Interrupt Vector Setup" on page 136
- "Using Interrupts in Your Application" on page 138

interrupt Keyword

Functions that are preceded by either of the `#pragma`s `interrupt` or `nested_interrupt` and functions that are associated with the interrupt storage class are designated as interrupt handlers. These functions should neither take parameters nor return a value. Because an interrupt handler is not called from another function but vectored to by the hardware, you cannot pass parameters to it. Interrupt routines can, however, access global variables or static variables. The compiler issues a warning if it detects an interrupt routine with arguments.

The compiler stores the machine state at the beginning of these functions and restores the machine state at the end of these functions. The `interrupt` and `nested_interrupt pragmas` are handled differently in the compiler. You should use `#pragma interrupt` for all interrupt handlers if your application does not make use of nested interrupts; otherwise, use `#pragma nested_interrupt` for all interrupt handlers. It is important to use either `interrupt` or `nested_interrupt` consistently throughout your application.

Specifically, the (non-nested) `interrupt` function executes an `EXX` instruction that preserves all the current registers upon entry to the function and an `EXX` instruction to restore the registers at the function exit. On the other hand, `nested_interrupt` does not use the alternate registers and instead pushes registers on the stack based on their use in the function. In either case, the compiler uses the reti instruction to return from these functions. For example:

```
void interrupt isr_timer0(void)  /* For non-nested interrupt use model */
{}
```

or

```
void nested_interrupt isr_timer0(void)  /* For nested interrupt use model */
{}
```

or

```
#pragma interrupt /* For non-nested interrupt use model */
void isr_timer0(void)
{}
```

or

```
#pragma nested_interrupt     /* For nested interrupt use model */
void isr_timer0(void)
{}
```

If you want the compiler to use the `retn` instruction to return from the interrupt, use `#pragma nmi_interrupt`. For example:

```
#pragma nmi_interrupt  /* ISR for non maskable interrupt */
void isr_nmi(void)
{}
```

Interrupt Vector Setup

The compiler provides the `_set_vector` function for interrupt vector setup. This function can be used to specify the address of an interrupt handler for an interrupt vector. The `_set_vector` function works by dynamically reading the base of the vector table and placing the address of the interrupt handler in it.

The following is the effective `_set_vector` function prototype. The first argument here is an integer defining the interrupt, and the second argument is the name of the associated interrupt handler. The function returns the previous handler that was present in the vector location before writing the new vector. The prototype of the function is included in a CPU-specific header file (such as `eZ80F91.h`) and has the following form:

```
void* _set_vector(int vectnum,void (*hndlr)(void));
```

An example of the use of `_set_vector` is as follows:

```
extern void* interrupt isr_timer0(void);
void main(void)
{
      _set_vector(TIMER0_IVECT, isr_timer0);
}
```

The following values for *vectnum* are supported:

EMACRX_IVECT	UZI1_IVECT	PORTB5_IVECT
EMACTX_IVECT	RTC_IVECT	PORTB6_IVECT
EMACSYS_IVECT	UART0_IVECT	PORTB7_IVECT
MACC_IVECT	UART1_IVECT	PORTC0_IVECT
DMA0_IVECT	I2C_IVECT	PORTC1_IVECT
DMA1_IVECT	SPI_IVECT	PORTC2_IVECT
PLL_IVECT	PORTA0_IVECT	PORTC3_IVECT
FLASH_IVECT	PORTA1_IVECT	PORTC4_IVECT
PRT0_IVECT	PORTA2_IVECT	PORTC5_IVECT
PRT1_IVECT	PORTA3_IVECT	PORTC6_IVECT
PRT2_IVECT	PORTA4_IVECT	PORTC7_IVECT
PRT3_IVECT	PORTA5_IVECT	PORTD0_IVECT
PRT4_IVECT	PORTA6_IVECT	PORTD1_IVECT
PRT5_IVECT	PORTA7_IVECT	PORTD2_IVECT
TIMER0_IVECT	PORTB0_IVECT	PORTD3_IVECT
TIMER1_IVECT	PORTB1_IVECT	PORTD4_IVECT
TIMER2_IVECT	PORTB2_IVECT	PORTD5_IVECT
TIMER3_IVECT	PORTB3_IVECT	PORTD6_IVECT
UZI0_IVECT	PORTB4_IVECT	PORTD7_IVECT

NOTE: The CPU-specific header files, such as <`eZ80F91.h`>, 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_eZ80Acclaim!_`<*version*>, where <*version*> might be `4.11.0` or `5.0.0`.

For clarity, it is recommended to place all the `_set_vector` calls in main. For example:

```
extern void my_zero_timer(); /* declared as interrupt */
extern void my_one_timer(); /* declared as interrupt */
int main() {
_di(); /* disable interrupts */
_set_vector(TMR0, my_zero_timer);
_set_vector(TMR1, my_one_timer);
_ei(); /* enable interrupts */

/* Body of application */
}
```

If you are in the early stages of developing your application, you might need to capture all of the interrupt vectors so that any unexpected interrupts are vectored to a known section of code. The `_init_default_vectors` function can be used for this purpose. It takes no arguments and should be called at the beginning of main. This function sets up all of the entries in the vector table to point to a single interrupt handler. Interrupts must be disabled at the point where this function is called.

Using Interrupts in Your Application

The eZ80Acclaim! family has a number of options for handling interrupts.

If you are using the default boot module, use the following procedure to enable the first interrupt:

1. Call `init_default_vectors()` once
2. Call `set_vector()` with the appropriate parameters.
3. Configure and enable the peripherals interrupt.

For each additional interrupt, use steps 2 and 3 only.

The eZ80F92 and eZ80F93 operate like the eZ80190 in terms of having on-chip SRAM available. If, however, you are only targeting to have your code run within the on-chip Flash memory, you might want to just keep your ISR routines within the first 64 Kbytes and point the default vector addresses right to the ISR routines.

The eZ80Acclaim! eZ80F91 devices, however, have a new feature that has been added to the interrupt controller to help out with interrupts. The default interrupt vectors have been changed from two byte addresses to four bytes. This allows you to point the default ISR vector right to your ISR routine. The following is some simple ISR setup code for the eZ80F91 device:

```
;*********************************
; Program entry point
;*********************************
           .org  %00
           di
           jp.lil_c_int0     ; Jump around the ISR vectors.
```

```
;------------------------------------------------------------
; ISR Vectors
;
            Note the ‘DL’ define - This gives us two words or 4 bytes.
                                   The upper byte is loaded with 00
                                   We only need 24 bits.
            .org  %40
            dl      %000000         ;
            dl      %000000         ;
            dl      %000000         ;
            dl      %000000         ;
            dl      %000000         ;
            dl      _isr_timer0     ;PRT0_ISR
            dl      _isr_timer1     ;PRT1_ISR
            dl      %000000         ;
            dl      %000000         ;
            dl      %000000         ;
            dl      %000000         ;
            dl      %000000         ;
            dl      _isr_uart0      ;UART0_ISR
            dl      _isr_uart1      ;UART1_ISR
            dl      %000000         ;

            ** add all interrupt vectors that you are going to use.

;------------------------------------------------------------------------
; Initialize Stack pointer
            extern  TOSPS
            extern  TOSPL

_c_int0:
            ld.sis sp,TOSPS         ; Setup SPS
            ld.lil sp,TOSPL         ; Setup SPL
;************************************************************************

            ld a, 00h               ; Disable on-chip SRAM
            out (RAM_CTL0), a;     ; depends on what you want to do with the
                                    ; on-chip SRAM.

            **** do other chip init here.
            **** final jump to main

call        _main       ; main()

void main(void)
{
init_com1();                  // Init com port - enable com1 ISR
```

```
            init_timer1();    // Init 100ms Timer - enable timer 1 ISR

_ei();                        // Turn on the master interrupt system.

do
{
            Your code goes here....
}while(1);

/*****************************************************************
 * This will initialize timer1 to interrupt every 10 ms
 *
 * 16 bit time constant is not big enough for 100 ms interrupts,
 * so we will use additional intermediate counter to count
 * every 10 ticks.
 */

void init_timer1(void)
{
            ticks1 = 0x00;
            intermediate_ticks1 = 0x00;

            TMR_CTL1 = 0x00;
            TMR_RRL1 = 0xFF;  //setup timer to interrupt every 10ms
            TMR_RRH1 = 0x1F;
            TMR_CTL1 = 0x0e;  //timer0 = multipass, /16, interrupt enable
            TMR_CTL1 |= 0x01; //enable timer
            TMR_IER1 = 0x01;  // Enables timer 1 interrupt
}

void init_com1(void)
{

            PC_ALT1 &= 0xf0;  //PD0 = uart0_tx, PD1 = uart0_rx
            PC_ALT2 |= 0x0F;

            UART_LCTL1=0x80;  //select dlab to access baud rate generators
            BRG_DLRL1=0x45;   // 9600
            BRG_DLRH1=0x01;
            UART_LCTL1=0x00;  //disable dlab

            UART_FCTL1=0xc7;  //clear tx fifo, clear rx fifo, fifo enable
            UART_LCTL1=0x1B;  //8-N-1
            UART_MCTL1=0x20;  //disable modem flow control
            UART_IER1=0x05;   //rx int enable, master int enable was 1
}

#pragma interrupt
```

```
void isr_timer1(void)
{
            unsigned char temp;
            unsigned int delay;
            temp = TMR_CTL1;//read to clear pending int
            temp = TMR_IIR1;

            intermediate_ticks1++;
            if(intermediate_ticks1 >= 10)
            {
                  intermediate_ticks1 = 0;
                  ticks1++;
            }
}

/*****************************************************************
 * All this ISR should do is put the data into our internal fifos
 *
 */

#pragma interrupt
void isr_uart1(void)
{
     short temp;

     temp = UART_LSR1;

             if ( temp & 0x04 )
                  {
            mdb_buff[byte_pos] = UART_RBR1;
                  byte_pos++;
                  done = 1;
                  }

            if ( temp & 0x01)
                  {
            mdb_buff[byte_pos] = UART_RBR1;
                  byte_pos++;
                  }

            while( UART_LSR1 & 0x20) {//THRE int

                  if( ! fifo_empty(uart1tx->fifo) ) {// and we still have
stuff to send ...
                        UART_THR1=fifo_get(uart1tx->fifo);// send it.
                  } else {                // otherwise ...
                        UART_IER1&=0xfd;// disable tx interrupts
                        break;
                        }
                }
}
}
```

In summary, setting up interrupts is somewhat complex on the eZ80® family as there are a number of things to configure, such as the I register, the long and short jump tables, and so on. Because the eZ80F92 and eZ80F93 only have two byte addresses for the default interrupt vectors, you need to set up other jump tables to bridge the gap into the 24-bit world when using those CPUs. The other thing to keep in mind is that the I register controls the upper 8 bits of this default interrupt vector, allowing you to move the overall interrupt jump table anywhere in the 64K range.

Placement Directives

The eZ80 Acclaim! C-Compiler provides language extensions to declare a variable at an address and to align a variable at a specified alignment. The following sections describe placement directives:

- "Placement of a Variable" on page 142
- "Placement of Consecutive Variables" on page 143
- "Alignment of a Variable" on page 143

Placement of a Variable

The following syntax can be used to declare a global or static variable at an address:

```
char placed_char _At 0xB7E100; // placed_char is assigned an address 0xB7E100

struct {
            char ch;
            int ii;
} ss _At 0xB7E110;      // ss is assigned an address 0xB7E110

const char init_char _At 0x1000 = 33;
                        // init_char is in rom at 0x1000 and initialized to 33
```

NOTE: Only placed variables with the `const` qualifier can be initialized. Non-`const` placed variables are not re-initialized correctly upon program reset.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 0xB7E000;
char ch2 _At ...;
char ch3 _At ...;
```

This places `ch1` at address `0xB7E000`, `ch2` at the next address (`0xB7E001`) after `ch1`, and `ch3` at the next address (`0xB7E002`) 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 `const` qualifier can be initialized. Non-`const` aligned variables would not be re-initialized correctly upon program reset. The uninitialized aligned variables are not initialized to zero by the compiler startup routine.

Inline Assembly

There are two methods of inserting assembly language within C code:

- "Inline Assembly Using the Pragma Directive" on page 143
- "Inline Assembly Using the asm Statement" on page 143

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 to the assembler 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 and function name with "_". Global variables and functions can therefore be accessed in inline assembly by prefixing their name with "_". The local variables and parameters cannot be accessed in inline assembly.

fract Keyword

The compiler extends the ANSI C language to include support for a new base type called `fract` that supports fixed-point fractional numbers. Declaring a `fract` variable is very similar to declaring an integer variable, although the values to be associated with the variable are not integers. Both `signed` and `unsigned fracts` are supported. The following are examples of legal fractional variable declarations:

```
int fract sif;               /* signed integer fract */
unsigned short fract usf;    /* unsigned short fract */
char fract ascf[10];         /* array of signed char fracts */
```

The `char`, `short`, and `int` base types determine the size of the object based upon the default base type sizes for the target processor. For eZ80Acclaim!, they are 8, 16, and 24 bits wide, respectively. The `long fract` type is not supported.

Fractional Fixed-Point Representations

The compiler uses fractional fixed-point arithmetic for improved efficiency over floating-point representations. Both `signed` and `unsigned` fractional numbers are supported. A `signed` fractional variable *n* is always within the following range: `-1 <= n < 1`

When representing a signed fractional number, the most significant bit represents the sign, and the remaining bits represent the two's complement of the fraction. The binary point is immediately after the sign bit.

An unsigned fractional variable *n* is always within the range: 0 <= *n* < 1. The binary point is just before the most significant bit.

When determining the value of a fractional number, look at each bit of the fraction as a negative power of two. For example, consider the following `signed short fract`:

	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	2^{-9}	2^{-10}	2^{-11}	2^{-12}	2^{-13}	2^{-14}	2^{-15}
s	.b	b	b	B	b	b	b	b	b	b	b	b	b	b	b

If the sign bit is negative (in the case of a signed type), take the two's complement of the value before decoding the fractional number. For example, to decode the value `0xb000` into a real number representation, first negate the value to obtain `0x5000`. Then, sum the powers of two for each bit set as follows:

s	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	2^{-9}	2^{-10}	2^{-11}	2^{-12}	2^{-13}	2^{-14}	2^{-15}
0	.1	0	1	0	0	0	0	0	0	0	0	0	0	0	0

Therefore, `signed 0xb000` represents $-(2^{-1} + 2^{-3}) = -0.625$.

The minimum resolution of the `fract` value varies, depending on the size and signedness of the data type, as follows:

`char fract`	: $1/2^7 = 0.0078125$
`unsigned char fract`	: $1/2^8 = 0.0039063$
`short fract`	: $1/2^{15} = 0.000030518$
`unsigned short fract`	: $1/2^{16} = 0.000015259$
`int fract`	: $1/2^{23} = 0.00000011921$
`unsigned int fract`	: $1/2^{24} = 0.000000059605$

Assigning Values to fract Variables

When assigning values of `fract` variables, floating-point constant syntax can be used. For example:

```
short fract f = -0.5;
f= -5/0.9;
```

Assignments between `fracts` of different sizes only move the least significant bytes, and hence the resulting value is distorted. Assignments between `signed` and `unsigned` versions of a `fract` do not preserve the signedness of the value, and hence the resulting value is distorted. Assignment between `fract` and `float` (and vice versa) is allowed—only the fractional part is transacted between the objects, and the integral part (if any) is ignored.

Fractional Expressions

In the fractional fixed-point system, fractional values are considered a higher order type than floating-point values. If `floats` or `doubles` are in an expression containing `fracts`, the `floats` and `doubles` are converted to `fracts` as necessary, even though a loss of accuracy might occur. The fractional fixed-point model assumes that the operation's efficiency is more important than its accuracy.

Basic Fractional Arithmetic

The compiler supports signed/unsigned fractional arithmetic for the +, -, and * operators. The binary operators +, -, and * on the `fract` data type give correct results only if both the operands are of same size and of same sign-type. Otherwise, the values are distorted for the reasons cited in "Assigning Values to fract Variables" on page 145.

All other arithmetic operators on `fract` treat the data as if they were of integral types (ignoring the `fract` keyword), and hence the resulting value might not make sense within the context of `fract`.

Bitwise Logical and Shift Operators

The compiler supports all bitwise logical operators, including &, |, ^, <<, and >> for fractional types, although ANSI does not allow these operations for floating-point types. You can use these operators to vary the position of the binary point in a fractional number. This feature can be very useful when the result of a fractional expression is not within the range of –1 to +1 or 0 to +1 but a wider range such as –3 to +3.

Scaled Fractional Arithmetic

You can alter the range of a fractional fixed-point value by modifying where its implied binary point resides. You can perform this operation using the shift operators as follows:

```
unsigned short fract f1,f2;
int int_part;
unsigned short fract fract_part;
f2 = .55;
f2 >>= 3;               /* move the binary point to right */
f1 = f2 + f2;           /* .55 + .55 = 1.1 */
int_part = (f1 & 0xe000) >> 13;     /* 1 */
fract_part = f1 << 3;   /* 0.1 */
```

Although `fract` arithmetic assumes that the binary point is just after the most significant bit (or after the sign bit in the case of a signed type), you can perform the same operations on `fracts` that have been shifted to move the binary point.

NOTE: However, be careful when dealing with negative `signed fracts` and make sure that all values involved in an operation have their binary points aligned.

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;
}
```

Supported New Features from the 1999 Standard

The eZ80Acclaim! compiler implements the following new features introduced in the ANSI 1999 standard, also known as ISO/IEC 9899:1999:

- C++ Style Comments

  Comments preceded by `//` and terminated by the end of a line, as in C++, are supported.

- Long Long Int Type

  The `long long int` type is allowed. (In the eZ80Acclaim! 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 eZ80Acclaim! C-Compiler are as follows:

`int`	24 bits
`short int`	16 bits
`char`	8 bits
`long`	32 bits
`float`	32 bits
`double`	32 bits

PREDEFINED MACROS

The eZ80Acclaim! 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 eZ80Acclaim! C-Compiler:

__ACCLAIM__	This macro is defined and set to 1 for the eZ80Acclaim! compiler and is otherwise undefined.
__EZ80__	This macro is defined and set to 1 for the eZ80Acclaim! compiler and is otherwise undefined.
__FPLIB__	This macro is defined on all ZiLOG compilers and indicates whether the floating-point library is available.
__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 rather than some other vendor.

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 __EZ80__
    printf("For eZ80 ");
#endif
#ifdef __ZDATE__
    printf("Built on %d.\n", __ZDATE__);
#endif
}
```

CALLING CONVENTIONS

The C-Compiler imposes a strict set of rules on 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.

The following sections describe the calling conventions:

- "Function Call Mechanism" on page 149
- "Special Cases" on page 151

Function Call Mechanism

A function (caller function) performs the following tasks when it calls another function (called function):

1. Save all registers (other than the return value register, to be defined below) that might be needed in the caller function after the return from the call, that is, a "caller save" mechanism is used. The registers are saved by pushing them on the stack before the call.
2. Push all function 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. Then 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 increments the stack pointer.
5. The caller then restores the saved registers by popping them from the stack.

The following example illustrates what must be done in an assembly procedure that calls a C function, including "caller save" of the BC register:

```
LD BC,123456h ; BC is used across function call
PUSH BC; Must be pushed before arguments are pushed
PUSH HL; Push argument
CALL _foo; Might modify BC
POP BC; Remove argument
POP BC; Must be popped after arguments are deallocated
ADD HL,BC; Wrong value of BC could be used if BC not
            ; saved by caller
```

In the eZ80®, a multiple of 3 bytes is always used when pushing arguments on the stack. The following table shows how arguments of different types are passed.

Table 1. Passing Arguments

Type	Size	Memory (Low to High)
`char`	3 bytes	`xx ?? ??`
`short`	3 bytes	`xx xx ??`
`int`	3 bytes	`xx xx xx`
`long`	6 bytes	`xx xx xx xx ?? ??`
`float`	6 bytes	`xx xx xx xx ?? ??`
`double`	6 bytes	`xx xx xx xx ?? ??`
`pointer`	3 bytes	`xx xx xx`

The called function performs the following tasks:

1. Push the frame pointer (that is, the IX register) onto the stack and allocate the local frame:
   – Set the frame pointer to the current value of the stack pointer.
   – Decrement the stack pointer by the size of locals and temporaries, if required.
2. Execute the code for the function.
3. If the function returns a scalar value, place it in the appropriate register as defined in the following table. For functions returning an aggregate, see “Special Cases” on page 151.
4. Deallocate the local frame (set the stack pointer to the current value of frame pointer) and restore the frame pointer register (IX) from stack.
5. Return.

The following table specifies how scalar values (those other than structs or unions) are returned.

Table 2. Returning Values

Type	Register	Register Contents: Most to Least Significant
`char`	A	`xx`
`short`	HL	`?? xx xx`
`int`	HL	`xx xx xx`
`long`	E:HL	`xx: xx xx xx`
`float`	E:HL	`xx: xx xx xx`
`double`	E:HL	`xx: xx xx xx`
`pointer`	HL	`xx xx xx`

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. The following figure shows a diagram of the eZ80Acclaim! C-Compiler dynamic call frame layout.

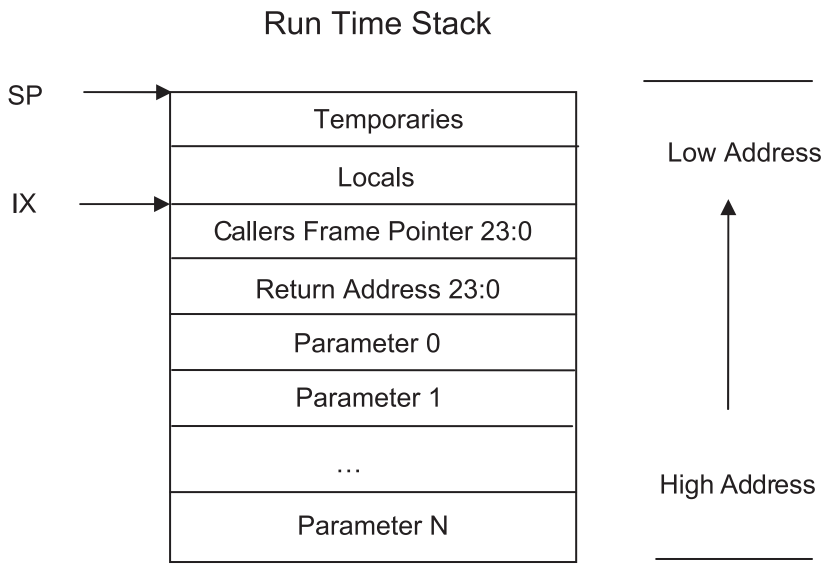


Figure 88. Call Frame Layout

Special Cases

Some function calls do not follow the mechanism described in “Function Call Mechanism” on page 149:

- 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 eZ80Acclaim! 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 "Calling Conventions" on page 149.

The following sections describe how to call assembly functions from C:

- "Function Naming Convention" on page 152
- "Variable Naming Convention" on page 152
- "Argument Locations" on page 153
- "Return Values" on page 153
- "Preserving Registers" on page 153

Function Naming Convention

Assembly function names must be preceded with an _ (underscore) in order to be callable from C. The compiler prefixes C 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 generated assembly by the compiler.

Variable Naming Convention

When the compiler generates an assembly file from C code, all names of global variables are prefixed with an underscore.

Names of local static variables are prefixed with an underscore followed by a function number to avoid assembly errors when the same local static variable occurs more than once in the same file.

Argument Locations

The assembly function assigns the location of the arguments following the C calling conventions as described in "Calling Conventions" on page 149.

For example, if you are using the following C prototype:

```
void myfunc(short arga, long argb, short *argc, char argd, int arge)
```

The arguments are placed on the stack, and their offsets from Stack Pointer (SP) at the entry point of an assembly function are as follows:

arga: –3(SP)

argb: –6(SP)

argc: –12(SP)

argd: –15(SP)

arge: –18(SP)

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 149.

For example, if you are using the following C prototype:

```
long myfunc(short arga, long argb, short *argc)
```

The assembly function returns the long value in registers E:HL.

Preserving Registers

The eZ80Acclaim! C-Compiler implements a caller save scheme. The assembly function is not expected to save and restore the registers it uses (unless it makes calls to C functions; in that case, it must save the registers it is using by pushing them on the stack before the call).

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 function and call them from an assembly program.

NOTE: The C-Compiler precedes the function names with an underscore in the generated assembly. See "Function Naming Convention" on page 152.

The following example shows an assembly source file referencing the `sin` function:

- "Assembly File" on page 154
- "Referenced C Function Prototype" on page 154

The `sin` function is defined in the C math library.

Assembly File

```
                        XREF _sin
            segment DATA
_angle:
            df 0.523599 ; angle in radians
_res:                   ; result
            ds 4

            segment CODE
_myfunc:
            ...
            push DE     ; save the live data, if any
            ld BC,(_angle)
            push BC                 ; push the argument
            ld A,(_angle+3)
            ld C,A
            push BC
            call _sin   ; call the C function
            pop BC      ; restore the stack by popping out the arguments
            pop BC
            ld (_res),HL ; result is in E:HL registers
            ld A,E
            ld (_res+3),A
            pop DE      ; restore the live data
            ...
```

Referenced C Function Prototype

```
double sin (double x);
```

NOTE: As mentioned in “Double Treated as Float” on page 166, the eZ80 Acclaim! C-Compiler treats `double`s as if they were `float`s.

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 383 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 smaller library of functions specific to ZiLOG or to the eZ80Acclaim! is also provided.

As mentioned at the beginning of this chapter, the eZ80Acclaim! 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 “C Standard Library” on page 318. The deviations from the ANSI Standard in these files are summarized in “Library Files Not Required for Freestanding Implementation” on page 166.

In ZDS II for the eZ80Acclaim!, the nonstandard library of ZiLOG-specific header files and functions has been structured into ZSL (the ZiLOG Standard Library). Detailed information about the contents of that library can be found in the *ZSL API Manual*. Additionally, most C projects for the eZ80Acclaim! will find it useful to include the nonstandard header file <`eZ80.h`>. This header in turn includes a processor-specific header file (such as `eZ80F91.h`) using the CPU that has been selected in your project. That file provides preprocessor definitions for the many special-function registers (SFRs) and memory-mapped I/O registers of your selected eZ80Acclaim! CPU so that they can be referred to symbolically in your program.

NOTE: The ZiLOG-specific header file <`eZ80.h`> as well as the CPU-specific files such as <`eZ80F91.h`> 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_eZ80Acclaim!_<`*`version`*`>`, where <*version*> might be `4.11.0` or `5.0.0`.

There is an important interrelationship between the C Standard Library (described in “C Standard Library” on page 318) and the ZSL. Some functions of the C Standard Library make calls into ZSL functions at the point where access to actual eZ80® peripheral hardware is needed. The most common example of this is that the standard function `putchar()` makes a call to the ZSL function `putch()` in order to write a character to the eZ80’s UART device. Therefore, if you make calls to certain standard functions, you will find it necessary either to link the ZSL to your project as well or to provide some other means of getting hardware-related calls resolved. You can do that either by writing your own versions of `putch()` and other ZSL functions that are called from the C Standard Library or by rewriting the C Standard Library functions to call some other function you have provided. For more information about linking in the C Standard Library and/or ZSL to your project, see “Use Default Libraries” on page 87.

NOTE: There is no unnecessary code size penalty associated with linking in the C Standard Library or ZSL. Only those functions that are called in your code are linked to your application.

PSEUDOINSTRUCTION MACROS GENERATED BY THE C-COMPILER

A small number of pseudoinstruction macros are implemented in the C-Compiler so that the compiler can more easily generate efficient assembly code. A pseudoinstruction has the syntax of a single processor instruction, but it is actually a combination of two or more processor instructions that work together. These macros are not primarily intended for use by assembly coders; they are generated by the compiler and so sometimes can be seen in compiler-generated assembly code. These descriptions are provided so that users can understand the translation from these macros to actual assembly instructions. The following pseudoinstruction macros are implemented in the eZ80® C-Compiler:

- "UEXT HL (Unsigned Extension)" on page 156
- "SEXT HL (Signed Extension)" on page 156
- "LD BC,DE" on page 156

UEXT HL (Unsigned Extension)

This macro clears register HL so that later an 8-bit value can be moved to L, resulting in a 24-bit unsigned extension (into HL) of the 8-bit value. Register A is always used as the source. The assembler translates this pseudoinstruction into the following sequence of instructions:

```
OR A,A
SBC HL,HL
```

SEXT HL (Signed Extension)

This macro fills register HL with all 0's or all 1's depending on the sign bit (that is, bit 7) of register A. This is done so that later moving the original value of register A into register L will result in a 24-bit sign extension (into HL) of the 8-bit value in register A. The assembler translates this pseudoinstruction into the following sequence of instructions:

```
RLA
SBC HL,HL
```

LD BC,DE

There is no actual eZ80® instruction to move data between 24-bit general-purpose registers. The assembler implements these pseudoinstructions with a sequence of `push` and `pop` instructions:

```
PUSH DE
POP BC
```

STARTUP FILES

The startup or C run-time initialization module is an assembly program that performs required startup functions and then calls `main`, which is the C entry point. The startup program performs the following initializations of the C run-time environment:

- Configure the external and internal memory interfaces.
- Initialize the stack pointer.
- Clear the uninitialized variables to zero.
- Set the initialized variables to their initial value from ROM.
- Copy code from ROM to RAM if specified in the linker command file.
- Allocate space for the `errno` variable used by the C run-time libraries.
- Allocate space for interrupt vectors and set initial default values for interrupt handlers.
- Initialize peripheral devices used by ZSL if included in the project.

The following table lists the startup files provided with the eZ80Acclaim! C-Compiler. In this table, <*CPU*> stands for any of the strings `F91`, `F92`, `F93`, `L92`, or `190` as appropriate for the CPU you are using in your project.

Table 3. eZ80Acclaim! Startup Files

Name	Description
`lib\zilog\cstartup.obj`	C startup object file for common code
`src\boot\common\cstartup.asm`	C startup source file for common code
`lib\zilog\vectors24.obj`	Object file for 24-bit interrupt vectors
`src\boot\common\vectors24.asm`	Source file for 24-bit interrupt vectors
`lib\zilog\vectors16.obj`	Object file for 16-bit interrupt vectors
`src\boot\common\vectors16.asm`	Source file for 16-bit interrupt vectors
`lib\zilog\init_params_<CPU>.obj`	C startup object file for CPU-specific initializations
`src\boot\eZ80<CPU>\init_params_<CPU>.asm`	C startup source file for CPU-specific initializations
`lib\zilog\zsldevinitdummy.obj`	Object file to provide dummy peripheral initialization for use with ZSL
`src\boot\common\zsldevinitdummy.asm`	Source code to provide dummy peripheral initialization for use with ZSL
`src\boot\common\zsldevinit.asm`	Source code to provide peripheral initialization for use with ZSL
`lib\zilog\startup.obj`	Obsolete C startup object file, provided for backward compatibility

Table 3. eZ80Acclaim! Startup Files

Name	Description
`lib\zilog\startup190.obj`	Obsolete C startup object file for eZ80190, provided for backward compatibility
`src\boot\common\startup.asm`	Obsolete C startup source file, provided for backward compatibility

NOTE: Some users might want to modify the startup modules provided in the ZDS II distribution. Incorporating modified startup modules into your project is discussed in "C Startup Module" on page 86. Linker directive issues related to modifying the startup module are discussed in "Using Modified ZDS II Startup Modules" on page 264.

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 the following table.

Table 4. Segments

Segment	Description
`DATA`	Initialized global and static data
`BSS`	Un-initialized global and static data
`TEXT`	Constant data
`CODE`	Executable code
`STRSECT`	String literals
`.RESET`	Reset handler code
`.IVECTS`	Interrupt vectors
`.STARTUP`	C run-time initialization

LINKER COMMAND FILES FOR C PROGRAMS

The following sections describe how the eZ80Acclaim! linker is used to link a C program:

- "Linker Referenced Files" on page 159
- "Linker Symbols" on page 160
- "Sample Linker Command File" on page 161

For more detailed description of the linker and the various commands it supports, see "Using the Linker/Locator" on page 235. 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.

Linker Referenced Files

The default linker command file generated by the ZDS II IDE references system object files and libraries based on the project options selected by the user. A list of the system object files and libraries is given in the following table. The linker command file automatically selects and links to the appropriate version of the C run-time and (if necessary) floating-point libraries from the list shown in the following table, based on your project settings. In this table, <*CPU*> stands for any of the strings `F91`, `F92`, `F93`, `L92`, or `190`, as appropriate for the CPU you are using in your project.

Table 5. Linker Referenced Files

File	Description
`cstartup.obj`	C startup module object file, common code
`init_params_<`*CPU*`>.obj`	Object file for CPU-specific initialization
`vectors24.obj`	24-bit interrupt vectors object file
`vectors16.obj`	16-bit interrupt vectors object file
`zsldevinitdummy.obj`	Object file for dummy initialization of peripherals
`gpio.lib`	Library for GPIO devices, no debug information; part of ZSL
`gpioD.lib`	Library for GPIO devices, with debug information; part of ZSL
`uart<`*CPU*`>.lib`	CPU-specific library for UART devices, no debug information; part of ZSL
`uart<`*CPU*`>D.lib`	CPU-specific library for UART devices, with debug information; part of ZSL
`uart<`*CPU*`>sim.lib`	CPU-specific library for UART devices to be run on the simulator, no debug information; part of ZSL

Table 5. Linker Referenced Files (Continued)

File	Description
`uart<CPU>simD.lib`	CPU-specific library for UART devices to be run on the simulator, with debug information; part of ZSL
`crt.lib`	C run-time library, no debug information
`crtd.lib`	C run-time library, with debug information
`chelp.lib`	Library of C-Compiler helper functions, no debug information
`chelpD.lib`	Library of C-Compiler helper functions, with debug information
`fplib.lib`	Real floating-point library, no debug information
`fplibd.lib`	Real floating-point library, with debug information
`fpdumy.lib`	Floating-point do-nothing stubs
`crt190.lib`	C run-time library specialized to eZ80190, no debug information
`crt190d.lib`	C run-time library specialized to eZ80190, 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. The following table shows the list of symbols that might be defined in the linker command file, depending on the compilation memory model selected by the user. In this table, <*CSx*> denotes a particular chip select and can be any of the strings `CS1`, `CS2`, `CS3`, or `CS4`.

Table 6. Linker Symbols

Symbol	Description
`__low_data`	Base of DATA segment after linking
`__len_data`	Length of DATA segment after linking
`__low_romdata`	Base of the ROM copy of DATA segment after linking
`__low_bss`	Base of BSS segment after linking
`__len_bss`	Length of BSS segment after linking
`__low_code`	Base of CODE segment after linking
`__len_code`	Length of CODE segment after linking
`__low_romcode`	Base address of the ROM copy of CODE segment

Table 6. Linker Symbols (Continued)

Symbol	Description
`__copy_code_to_ram`	Flag indicating whether code is to be copied to RAM before executing
`__stack`	Top of stack is set as high address of available RAM
`__heapbot`	Base of heap for is set as low address of available RAM
`__heaptop`	Top of heap is set as high address of available RAM
`__crtl`	Flag indicating whether ZiLOG-supplied RTL is used
`_<CSx>_LBR_INIT_PARAM`	Chip select address lower bound initializer
`_<CSx>_UBR_INIT_PARAM`	Chip select address upper bound initializer
`_<CSx>_CTL_INIT_PARAM`	Chip select control initializer
`_<CSx>_BMC_INIT_PARAM`	Chip select bus mode initializer
`__RAM_CTL_INIT_PARAM`	On-chip RAM control initializer
`__RAM_ADDR_U_INIT_PARAM`	On-chip RAM address upper byte initializer
`__FLASH_CTL_INIT_PARAM`	On-chip Flash control initializer
`__FLASH_ADDR_U_INIT_PARAM`	On-chip Flash address upper byte initializer
`_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
`_OSC_FREQ`	Oscillator clock frequency (system clock)
`_OSC_FREQ_MULT`	On-chip phase-locked loop divider initializer
`__PLL_CTL0_INIT_PARAM`	On-chip phase-locked loop control register 0 initializer
`_zsl_g_clock_xdefine`	System clock frequency (this symbol used by ZSL)

Sample Linker Command File

The sample default linker command file for the standard link configuration 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` and your configuration is simply named `debug`, 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 Linker Directives Dialog Box" on page 82). Alternatively, you can define your own linker command file by selecting the Use Existing button (see "Use Existing" on page 83).

The most important of the linker commands and options in the default linker command file are now discussed 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 Output page (see “Project Settings—Output Page” on page 92). The `–quiet`, `-debug`, and `–noigcase` options are generated from the settings on the General page (see “Project Settings—General Page” on page 57). The other options shown here are all generated from the settings selected in the Warnings and Output pages (see “Project Settings—Warnings Page” on page 90 and “Project Settings—Output Page” on page 92).

```
RANGE ROM $000000 : $03FFFF
RANGE RAM $B80000 : $BFFFFF
RANGE EXTIO $0 : $FFFF
RANGE INTIO $0 : $FF
```

The ranges for the four address spaces are defined here. These ranges are taken from the settings in Address Spaces page (see “Project Settings—Address Spaces Page” on page 89).

```
CHANGE STRSECT is ROM
```

The `STRSECT` segment is moved into the ROM space by the preceding command. Because the contents of `STRSECT` are constant strings, this segment should always be placed in ROM in a production build, though for debugging purposes `STRSECT` might sometimes be left in RAM.

```
ORDER .RESET,.IVECTS,.STARTUP,CODE,DATA
```

This `ORDER` command specifies the temporal link order of these segments. The `.RESET` segment is placed at lower addresses with the `.IVECTS` segment immediately following it and so on.

```
COPY DATA ROM
```

This COPY command is a linker directive to make the linker place a copy of the initialized data segment DATA into the ROM address space. At run time, the C startup module then copies the initialized data back from the ROM address space to the RAM 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_romdata = copy base of DATA
DEFINE __low_data = base of DATA
DEFINE __len_data = length of DATA
DEFINE __low_bss = base of BSS
DEFINE __len_bss = length of BSS
DEFINE __stack = highaddr of RAM + 1
DEFINE __heaptop = highaddr of RAM
DEFINE __heapbot = top of RAM + 1
DEFINE __low_romcode = copy base of CODE
DEFINE __low_code = base of CODE
DEFINE __len_code = length of CODE
DEFINE __copy_code_to_ram = 0

DEFINE __crtl = 1
DEFINE __CS0_LBR_INIT_PARAM = $00
DEFINE __CS0_UBR_INIT_PARAM = $00
DEFINE __CS0_CTL_INIT_PARAM = $00
DEFINE __CS0_BMC_INIT_PARAM = $02
DEFINE __CS1_LBR_INIT_PARAM = $00
DEFINE __CS1_UBR_INIT_PARAM = $07
DEFINE __CS1_CTL_INIT_PARAM = $28
DEFINE __CS1_BMC_INIT_PARAM = $02
DEFINE __CS2_LBR_INIT_PARAM = $00
DEFINE __CS2_UBR_INIT_PARAM = $00
DEFINE __CS2_CTL_INIT_PARAM = $00
DEFINE __CS2_BMC_INIT_PARAM = $02
DEFINE __CS3_LBR_INIT_PARAM = $00
DEFINE __CS3_UBR_INIT_PARAM = $00
DEFINE __CS3_CTL_INIT_PARAM = $00
DEFINE __CS3_BMC_INIT_PARAM = $02
DEFINE __RAM_CTL_INIT_PARAM = $00
DEFINE __RAM_ADDR_U_INIT_PARAM = $00
DEFINE __FLASH_CTL_INIT_PARAM = $80
DEFINE __FLASH_ADDR_U_INIT_PARAM = $00

define _SYS_CLK_FREQ = 20000000
define _OSC_FREQ = 20000000
define _SYS_CLK_SRC = 0
define _OSC_FREQ_MULT = 1
define __PLL_CTL0_INIT_PARAM = $00
define _zsl_g_clock_xdefine = 50000000
```

These are the linker symbol definitions described in Table 6. They allow the compiler to know the bounds of the different memory areas that must be initialized in different ways by the C startup module and to configure the chip selects and other implementation details of your project.

```
"C:\PROGRA~1\ZiLOG\ZDSII_~1.1\samples\EZ80F9~1\src\ledDemo"= \
 C:\PROGRA~1\ZiLOG\ZDSII_~1.1\lib\zilog\vectors24.obj, \
 C:\PROGRA~1\ZiLOG\ZDSII_~1.1\lib\zilog\init_params_f91.obj, \
 C:\PROGRA~1\ZiLOG\ZDSII_~1.1\lib\zilog\cstartup.obj, \
 .\Buttons.obj, \
 .\LedMatrix.obj, \
 .\LedTimer.obj, \
 .\main.obj, \
 .\zsldevinit.obj, \
 C:\PROGRA~1\ZiLOG\ZDSII_~1.1\lib\std\chelpD.lib, \
 C:\PROGRA~1\ZiLOG\ZDSII_~1.1\lib\std\crtD.lib, \
 C:\PROGRA~1\ZiLOG\ZDSII_~1.1\lib\std\fplibD.obj, \
 C:\PROGRA~1\ZiLOG\ZDSII_~1.1\lib\zilog\gpioD.lib, \
 C:\PROGRA~1\ZiLOG\ZDSII_~1.1\lib\zilog\uartF91simD.lib
```

This final command shows that, in this example, the linker output file is named `ledDemo.lod`. The source object files (`Buttons.obj`, `LedMatrix.obj`, `LedTimer.obj`, and `main.obj`) are 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 startup module and related initialization modules (`cstartup.obj`, `vectors24.obj`, `init_params_f91.obj`, and `zsldevinit.obj`), the C helper library with debug (`chelpld.lib`), the C run-time library with debug (`crtD.lib`), the floating-point library with debug (`fplibD.lib`), and the relevant ZSL libraries (`gpioD.lib` and `uartF91simD.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, like chelpLD.lib, `crtLD.lib`, and `fpLD.lib` 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 the discussion of “Use Default Libraries” on page 87 for a further discussion of this area.

ANSI STANDARD COMPLIANCE

The ZiLOG eZ80Acclaim! 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 that are described in “Deviations from ANSI C” on page 165.

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 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 implementation” 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 eZ80Acclaim! C-Compiler, for example, supports a number of additional headers from the standard library, as specified in “Library Files Not Required for Freestanding Implementation” on page 166.

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 eZ80Acclaim! 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 eZ80Acclaim! CPU. These extensions are described in “Language Extensions” on page 135.

Deviations from ANSI C

The differences between the ZiLOG eZ80Acclaim! C-Compiler and the freestanding implementation of 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 135.

There are a small number of areas in which the eZ80Acclaim! C-Compiler does not behave as specified by the Standard. These areas are described in the following sections:

- “Prototype of Main” on page 165
- “Double Treated as Float” on page 166
- “Library Files Not Required for Freestanding Implementation” on page 166

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 eZ80Acclaim! C-Compiler uses `main()` as the function called at program startup. Because the eZ80Acclaim! 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 eZ80Acclaim! C startup module is in use, control simply goes to the following statement:

```
jmp $
```

which is equivalent to

```
label1:
            goto label1;
```

For this reason, in the eZ80Acclaim! C-Compiler, `main()` needs to be of type `void`; any returned value is ignored. Also, `main()` is not passed any arguments. In short, the following is the prototype for `main()`:

```
void main (void);
```

Unlike the hosted environment in which the closest allowed form for main is as follows:

```
int main (void);
```

Double Treated as Float

The eZ80Acclaim! 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 164, only four of the standard library header files are required by the standard to be supported in a freestanding compiler such as the eZ80Acclaim! 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 this 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>

  This header is ANSI compliant in the ZiLOG library except that the following functions of limited or no use in an embedded environment are not supplied:

  ```
  strtoul()
  _Exit()
  atexit()
  ```

LOCATING VARIABLES AT SPECIFIC ADDRESSES: OLDER METHOD

Beginning with release 4.11.0 of ZDS II, the eZ80 Acclaim! C-Compiler now provides an easy and natural extension to the C language for placing variables at specific locations, as described in “Placement Directives” on page 142. This is the recommended way to perform this task in newly written code. However, for compatibility with code developed using older versions of the eZ80 Acclaim! C-Compiler, this section describes a technique that can be used in those versions to accomplish the same thing more laboriously. This technique uses the macro-processing capabilities of ANSI C to control variable placement.

To access a specific location in memory using this older technique, you can declare a macro that expands into a pointer reference to the appropriate memory locations as shown in the following example:

```
#define b_x *((char *)0x20)
#define b_y *((int *)0x120)
#define b_array ((unsigned char * )0x30)
char c;
int i;
void main(void)
{
            b_x = 10;
            c = b_x;
b_y = 0x1234;
i = b_y;
c = b_array[c];
}
```

The various Special Function Registers can be defined and accessed in a similar manner, although this can now be done more easily using the placement directives described in “Placement Directives” on page 142.

NOTE: When using this method, check the address ranges on the Project Settings dialog box (“Project Settings—Address Spaces Page” on page 89) so that your program does not inadvertently access data in areas already allocated.

Another method of allocating and accessing variables at a specified location requires a little assembly programming. The idea is to allocate segments in an assembly module that is located using the ORG directive. Labels are placed at the appropriate location and are made public using the XDEF directive. When this module is linked with the rest of your program, the variable is located at the proper position in memory.

Assembly File

```
            xdef _my_data
            define_my_seg,space=ram,org=80h
            segment my_seg
_my_data:
            ds 20; 20 bytes at 80h
```

```
                end
```

C File

```
        extern unsigned char my_data[20];
        void foo(void)
        {
                unsigned char i;
                for (i=0;i<sizeof(my_data);++i)
                        my_data[i] = 0xff;
        }
```

NOTE: The C file refers to the variable as `my_data` while the assembly uses `_my_data`. These names are different because the compiler prefixes all global variable names with an underscore (_). Failing to perform these modifications results in undefined symbol errors from the linker.

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 the following:

- "Preprocessor Warning and Error Messages" on page 168;
- "Front-End Warning and Error Messages" on page 171
- "Optimizer Warning and Error Messages" on page 179
- "Code Generator Warning and Error Messages" on page 181

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 *"pathnam*e" 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 needs to be shortened.

023 "##" cannot be first or last token in replacement string.

The macro definition cannot have "##" 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 un-subscripted 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 type qualifiers 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 "const" and "volatile" 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 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.

199 Data objects cannot have "io" storage class.

Change the storage class.

202 Unrecognized/invalid type specifier

A type specifier was expected, and something different (like a label or symbol) was read. Or, a valid type specifier was read but cannot be used in this context.

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 nonconstant expression.

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 Can't 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 check box (see “Project Settings—Code Generation Page” on page 61) 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 check box 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 Can't 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.

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.

308 Excessive Registers required at line <*num*> of function <*func*>.

Excessive Page 0 registers are required at line number <*num*>. The compiler does not perform register page spilling, so complex expressions that generate this error must be factored into two or more expressions.

309 Interrupt function <*name*> cannot have arguments.

A function declared as an interrupt function cannot have function arguments.

313 Bitfield Length exceeds <*num*> bits.

The compiler only accepts bit-field lengths of 8 bits or less for char bit-fields, 16 bits or less for short bit-fields, and 32 bits or less for int and long bit-fields.

Using the Macro Assembler

You use the Macro Assembler to translate eZ80Acclaim! 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 "Project Settings—Assembler Page" on page 59).

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 using the Command Processor, see "Using the Command Processor" on page 387.

The following topics are covered in this chapter:

NOTE: For more information about eZ80Acclaim! CPU instructions, see the "CPU Instruction Set" section in the *eZ80® CPU User Manual* (UM0077).

ADDRESS SPACES AND SEGMENTS

The eZ80® architecture divides the entire memory space into various memory regions. These memory regions are called 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.

Address spaces and segments are described in the following sections:

- “Allocating Processor Memory” on page 183
- “Address Spaces” on page 183
- “Segments” on page 184
- “Assigning Memory at Link Time” on page 186

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 is defined with the ORG directive. All data and code in an absolute segment are 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 memory regions for the eZ80Acclaim! microprocessor are represented by the address spaces listed in the following table.

Table 7. eZ80Acclaim! Address Spaces

Space ID	Display Prefix	Description	Lowest and Highest Addresses	Size	Maximum That Can Be Retrieved at One Time
ROM	C	Standard memory address space. The ROM memory address space can contain both program code and data. If no address space is associated with a segment, this is the default space.	`00000000-00FFFFFF`	16 MB	3 bytes (24 bits)
RAM	D	Random access memory address space. The RAM memory address space can contain variable data and stack.	`00000000-00FFFFFF`	16 MB	3 bytes (24 bits)

Table 7. eZ80Acclaim! Address Spaces (Continued)

Space ID	Display Prefix	Description	Lowest and Highest Addresses	Size	Maximum That Can Be Retrieved at One Time
INTIO	I	On-chip I/O address space.	00000000–000000FF	256 bytes	1 byte (8 bits)
EXTIO	E	External I/O address space.	00000000–0000FFFF	64 KB	2 bytes (16 bits)

NOTE: The lowest and highest addresses are the lowest and highest *possible* addresses; the C-Compiler uses the values in the preceding table as a default. However, the hardware might not *physically* have all of the possible addresses, so you can use ZDS II to set a different range.

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 currently active segment. Every segment is associated with one and only one address space.

Segments are described in the following sections:

- "Predefined Segments" on page 184
- "User-Defined Segments" on page 185

Predefined Segments

For convenience, the segments listed in the following table are predefined by the assembler. Each segment gets assigned to one of the address spaces in Table 7. If no address space is associated with a segment, ROM is the default space. All of the predefined segments listed here can be aligned on any byte boundary.

Table 8. Predefined Segments

Segment Name	Address Space	Contents
__VECTORS	ROM	Constant data
STRSECT	RAM	Initialized strings
TEXT	ROM	Constant data

Table 8. Predefined Segments (Continued)

Segment Name	Address Space	Contents
BSS	RAM	Uninitialized data
DATA	RAM	Initialized data
CODE	ROM	Code

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 RAM using the following directives:

```
DEFINE MYDATA,SPACE=RAM
SEGMENT MYDATA
```

The `DEFINE` directive creates a new segment and attaches it to a space. For more information about using the `DEFINE` directive, see “DEFINE” on page 203. The `SEGMENT` directive attaches code and data to a segment. The `SEGMENT` directive makes the segment named in the directive the currently active segment. Any code or data following the `SEGMENT` directive resides in the segment until another `SEGMENT` directive is encountered. For more information about the `SEGMENT` directive, see “SEGMENT” on page 208.

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 particular segment many times, the module still has only one instance of that 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. For more information about link commands, see "Linker Commands" on page 247.

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 (see "Source Listing (.lst) Format" on page 186). The assembler creates <*source*>`.lst` unless you deselect the Generate Listing File (.lst) check box in the Assembler page of the Project Settings dialog box (see "Generate Assembly Listing Files (.lst)" on page 60).
- <*source*>`.obj` is an object file in relocatable OMF695 format. The assembler creates <*source*>`.obj`. See "Object Code (.obj) File" on page 187.

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:

- 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:

- 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:

- NEWPAGE
- 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 check box on the Output page (see “Show Absolute Addresses in Assembly Listings” on page 94). This option uses the information in the `.src` file (generated by the compiler when the `-keepasm` option is used or when the Generate Assembly Source check box is selected [see “Generate Assembly Source Code” on page 64]) 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

The following sections describe the form of an assembly source file:

- “General Structure” on page 188

- “Assembler Rules” on page 189

General Structure

Every nonblank 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.

The following sections describe the general structure:

- “Source Line” on page 188
- “Comment Line” on page 188
- “Label Field” on page 188
- “Instruction” on page 189
- “Directive” on page 189
- “Case Sensitivity” on page 189

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).

NOTE:

- Any instruction mnemonic (with no following operands) followed by a colon is treated as a label.

- Any instruction mnemonic not followed by a colon is treated as an instruction, even if it starts in the first column.

The first character of a label can be a letter, an underscore _ , a dollar sign ($), a question mark (?), a period (.), or pound sign (#). Following characters can include letters, digits, underscores, dollar signs ($), 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.

Labels that can be interpreted as hexadecimal numbers are not allowed. For example,

```
ADH:
ABEFH:
```

cannot be used as labels.

See “Labels” on page 222 and “Hexadecimal Numbers” on page 194 for more information.

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 eZ80® CPU instructions, see the “CPU Instruction Set” section in the *eZ80 CPU User Manual* (UM0077).

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 “Directives” on page 196 for more information.

Case Sensitivity

In the default mode, the assembler treats all symbols as case sensitive. Select the Ignore Case of Symbols check box on 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” on page 58. Assembler reserved words are not case sensitive.

Assembler Rules

The following sections describe assembler rules:

- “Reserved Words” on page 190
- “Assembler Numeric Representation” on page 191
- “Character Strings” on page 192

Reserved Words

The following list contains reserved words 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
.assume	.bes	.block	.bss	.byte
.copy	.data	.def	.ELIF	.ELSE
.ELSEIF	.emsg	.ENDIF	.ENDM	.ENDMAC
.ENDMACRO	.ENDSTRUCT	.EQU	.ER	.even
.extern	.FCALL	.file	.FRAME	.global
.IF	.include	.int	.LIST	.long
.MACEND	.MACRO	.MLIST	.mmsg	.MNOLIST
.NEWBLOCK	.ORG	.PAGE	.public	.R
.ref	.RR	.SBLOCK	.sect	.SET
.space	.STRING	.STRUCT	.TAG	.text
.USECT	.VAR	.wmsg	.word	.WRG
_ER	_R	_RR	_WRG	ADC
ALIGN	ASCII	ASCIZ	ASECT	ASSUME
BES	BFRACT	BLKB	BLKL	BLKP
BLKW	BSS	byte	C	C0
C1	C2	C3	CHIP	COMMENT
CONDLIST	COPY	CPU	DATA	DB
DBYTE	DD	DEFB	DEFINE	DF
DL	DMA	DOT_IDENT	DPTR	DS
DW	DW24	ELSEIF	END	ENDC
ENDM	ENDMACRO	ENDMODULE	ENDS	ENDSTRUCT
EQ	ERROR	ESECT	EXIT	F
FCALL	FCB	FILE	FLAGS	FRACT
FRAME	GE	GLOBAL	GLOBALS	GREGISTER
GT	HIGH	I2C	IFDIFF	IFE
IFFALSE	IFNDEF	IFNDIFF	IFNMA	IFNSAME
IFNTRUE	IFZ	IGNORE	INCLUDE	LE
LEADZERO	LFRACT	LIST	long	LONGREG
LOW	LT	MACCNTR	MACDELIM	MACEND
MACEXIT	MACFIRST	MACLIST	MACNOTE	MESSAGE
MI	MLIST	MNOLIST	MODULE	NC

NE	NEWBLOCK	NEWPAGE	NOCONDLIST	NOLIST
NOMACLIST	NOSPAN	NOV	NZ	OFF
ON	ORG	ORIGIN	OV	PAGELENGTH
PAGEWIDTH	PL	POPSEG	PP_ASG	PP_CONCAT
PP_DEF	PP_ELIF	PP_ELSE	PP_ENDIF	PP_ENDMAC
PP_EQU	PP_EVAL	PP_EXPRESSION	PP_GREG	PP_IF
PP_IFDEF	PP_IFMA	PP_IFNDEF	PP_IFNMA	PP_LOCAL
PP_MACEXIT	PP_MACRO	PP_NOSAME	PP_NIF	PP_SAME
PP_SBLOCK	PP_VAR	PRINT	PT	PUSHSEG
PW	r0	r10	r11	r12
r13	r14	r15	r2	r3
r4	r5	r6	r7	r8
r9	RESET	RP	rr0	rr10
rr12	rr14	rr2	rr4	rr6
rr8	SCOPE	SEGMENT	SET	SHORTREG
SPH	SPI	STRING	STRUCT	SUBTITLE
T	TAG	TEXT	TIMER0	TIMER1
TIMER2	TIMER3	TITLE	TRAP	UART0_RX
UART0_TX	UART1_RX	UART1_TX	UBFRACT	UFRACT
UGE	UGT	ULE	ULFRACT	ULT
UN_IF	UNSUPPORTED	USER_ERROR	USER_EXIT	USER_WARNING
VAR	VECTOR	WARNING	WDT	word
XDEF	XREF	Z	ZBREAK	ZCONTINUE
ZELSE	ZELSEIF	ZENDIF	ZIF	ZIGNORE
ZREPEAT	ZSECT	ZUSECT	ZWEND	ZWHILE
ZUNTIL				

NOTE: Additionally, 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. Floating-point numbers are 32-bit IEEE standard single-precision values. 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 or float value can be used as an operand, an expression can also be used. The assembler evaluates expressions in 32-bit signed arithmetic or 64-bit floating-point arithmetic. Logical expressions are bitwise operators.

The assembler detects overflow and division by zero errors in constant expressions. The following sections describe the syntax of writing an expression:

- "Arithmetic Operators" on page 192
- "Relational Operators" on page 193
- "Boolean Operators" on page 193
- "HIGH and LOW Operators" on page 193
- "HIGH16 and LOW16 Operators" on page 194
- "Decimal Numbers" on page 194
- "Hexadecimal Numbers" on page 194
- "Binary Numbers" on page 195
- "Octal Numbers" on page 195
- "Character Constants" on page 195
- "Operator Precedence" on page 195

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

`&`	Bitwise AND	Synonyms: `.and.`, `.AND.`
`|`	Bitwise inclusive OR	Synonyms: `.or.`, `.OR.`
`^`	Bitwise exclusive XOR	Synonyms: `.xor.`, `.XOR.`
`~`	Complement	
`!`	Boolean NOT	Synonyms: `.not.`, `.NOT.`

HIGH and LOW Operators

The HIGH and LOW operators can be used to extract specific bytes from an integer expression. The LOW operator extracts the byte starting at bit 0 of the expression, while the HIGH operator extracts the byte starting at bit 8 of the expression.

HIGH and LOW can also be used to extract portions of a floating-point value.

For example:

```
# LOW (X) ; 8 least significant bits of X
# HIGH (X) ; 8 most significant bits of X
```

HIGH16 and LOW16 Operators

The `HIGH16` and `LOW16` operators can be used to extract specific 16-bit words from an integer expression. The `LOW16` operator extracts the word starting at bit 0 of the expression; the `HIGH16` operator extracts the word starting at bit 16 of the expression.

`HIGH16` and `LOW16` can also be used to extract portions of a floating-point value.

For example:

```
# LOW16 (X) ; 16 least significant bits of X
# HIGH16 (X) ; 16 most significant bits of X
```

NOTE: A combination of the `HIGH16` and `LOW` operators can be used to extract the most significant byte of a 24-byte integer in the eZ80 Acclaim! For example:

```
TT equ %123456
segment code
LD A,HIGH(TT)               ; This loads 34 into A
LD A,LOW(TT)                ; This loads 56 into A
LD A,LOW(HIGH16(TT))        ; This loads 12 into A
```

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
-123_456 ; negative decimal
1_000_000; decimal number with underscores
_123_;  NOT an integer but a name.  Underscore can be neither first
            nor last character.
12E-45 ; decimal float
-123.456 ; decimal float
123.45E6 ; decimal float
```

Hexadecimal Numbers

Hexadecimal numbers are signed 32-bit integers ending with the `h` or `H` suffix (or starting with the `%` prefix) 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, but only when the `%` prefix is used instead of the `H` suffix. For example:

```
ABCDEFFFH ; hexadecimal
%ABCDEFFF ; hexadecimal
-0FFFFh ; negative hexadecimal
%ABCD_EFFF; hexadecimal number with underscore
ADC0D_H; NOT a hexadecimal number but a name; hexadecimal numbers
            cannot contain both underscore and H suffix
```

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:

```
1234o ; octal number
-1234o ; negative octal number
1_234o; 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

The following table 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 9. Operator Precedence

Level 1	`()`						
Level 2	`~`	`unary-`	`!`	`high`	`low`		
Level 3	`**`	`*`	`/`	`%`			
Level 4	`+`	`-`	`&`	`|`	`^`	`>>`	`<<`
Level 5	`<`	`>`	`<=`	`>=`	`==`	`!=`	

NOTE: Shift Left (<<) and OR (|) have the same operator precedence and are evaluated from left to right. If you need to alter the order of evaluation, add parentheses to ensure the desired operator precedence. For example:

```
ld a,  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 a, #(1<<2)|(1<<2)|(1<<1)
```

The modified constant expression evaluates to 6 H.

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. The following sections provide details for each of the supported assembler directives:

- "ALIGN" on page 197
- ".COMMENT" on page 197
- "CPU" on page 198
- "Data Directives" on page 199
- "DEFINE" on page 203
- "DS" on page 205
- "END" on page 205
- "EQU" on page 205
- "INCLUDE" on page 206

- “LIST” on page 207
- “NEWPAGE” on page 207
- “NOLIST” on page 207
- “ORG” on page 208
- “SEGMENT” on page 208
- “TITLE” on page 209
- “VAR” on page 209
- “XDEF” on page 210
- “XREF” on page 210
- “Structures and Unions in Assembly Code” on page 210

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 meaning $
```

This text is a comment, delimited by a dollar sign, and spanning multiple source lines. The dollar sign ($) is a deliminator that marks the line as the end of the comment block.

CPU

Defines to the assembler which member of the eZ80Acclaim! 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 CODE space.

The CPU directive can use the following predefined symbols:

- `EZ80F93`
- `EZ80F92`
- `EZ80F91`
- `EZ80L92`
- `EZ80190`

When eZ80F91, for example, is selected, the following preprocessor symbols are automatically defined (note the leading underscore):

```
_EZ80F91=1
_EZ80F92=0
_EZ80F93=0
_EZ80L92=0
_EZ80190=0
```

The symbols are defined in a similar fashion when any other CPU is selected.

These preprocessor symbols can be used for conditional assembly. For example:

```
.if         _EZ80F91
...
.endif
```

Using the predefined symbols with the `IFDEF` conditional assembly directive in your code does not produce useful results, precisely because the symbols are already defined as either 0 or 1. For example, the following always evaluates to true:

```
IFDEF _EZ80190
```

Instead, use a preprocessor symbol that is not predefined, as in the following code:

```
EZ80190 EQU 1
IFDEF EZ80190
```

NOTE: The `CPU` directive is used to determine the physical location of the interrupt vectors.

Syntax

<cpu_definition> = > `CPU` = *<cpu_name>*

Example

```
CPU = EZ80L92
```

Data Directives

The following data directives allow you to reserve space for specified types of data:

- "BFRACT and UBFRACT Declaration Types" on page 200
- "FRACT and UFRACT Declaration Types" on page 200
- "BLKB Declaration Type" on page 201
- "BLKL Declaration Type" on page 201
- "BLKP Declaration Type" on page 201
- "BLKW Declaration Type" on page 201
- "DB Declaration Type" on page 201
- "DD Declaration Type" on page 202
- "DF Declaration Type" on page 202
- "DL Declaration Type" on page 202
- "DW Declaration Type" on page 203
- "DW24 Declaration Type" on page 203

Syntax

```
<data directive> = > <type> <value>
<type> => BFRACT
       => BLKB
       => BLKL
       => BLKP
       => BLKW
       => DB
       => DD
       => DF
       => DL
       => DW
       => DW24
       => FRACT
       => UBFRACT
       => UFRACT
<value_list> => <value>
```

=> *<value_list>*,*<value>*
<value> => *<expression>*|*<string_const>*

The `BLKB`, `BLKL`, `BLKP`, and `BLKW` directives can be used to allocate a block of byte, long, pointer, or word data, respectively.

BFRACT and UBFRACT Declaration Types

Syntax

`BFRACT`	signed fractional (8 bits)
`UBFRACT`	unsigned fractional (8 bits)

Examples

```
BFRACT [3]0.1, [2]0.2 ; Reserve space for five 8 bit
                      ; signed fractional numbers.
                      ; Initialize first 3 with 0.1,
                      ; last 2 with a 0.2.
UBFRACT [50]0.1,[50]0.2 ; Reserve space for 100 8 bit
                        ; unsigned fractional numbers.
                        ; Initialize first 50 with a
                        ; 0.1, second 50 with a 0.2
BFRACT 0.5 ; Reserve space for one 8-bit signed, fractional number
           ; and initialize it to 0.5.
```

FRACT and UFRACT Declaration Types

Syntax

`FRACT`	signed fractional (8 bits)
`UFRACT`	unsigned fractional (8 bits)

Examples

```
FRACT [3]0.1, [2]0.2 ; Reserve space for five 16 bit
                     ; signed fractional numbers.
                     ; Initialize first 3 with 0.1,
                     ; last 2 with a 0.2.

UFRACT [50]0.1,[50]0.2 ; Reserve space for 100 16 bit
                       ; unsigned fractional numbers.
                       ; Initialize first 50 with a
                       ; 0.1, second 50 with a 0.2

FRACT 0.5 ; Reserve space for one 16-bit signed, fractional number
          ; and initialize it to 0.5.
```

BLKB Declaration Type

Syntax

`BLKB` number of bytes (8 bits each) [, <*init_value*>]

Example

```
BLKB 16 ; Allocate 16 uninitialized bytes.
BLKB 16, -1 ; Allocate 16 bytes and initialize them to -1.
```

BLKL Declaration Type

Syntax

`BLKL` number of longs (32 bits each) [, <*init_value*>]

Example

```
BLKL 16 ; Allocate 16 uninitialized longs.
BLKL 16, -1 ; Allocate 16 longs and initialize them to -1.
```

BLKP Declaration Type

Syntax

`BLKP` number of pointers (24 bits each) [, <*init_value*>]

Example

```
BLKP 16 ; Allocate 16 uninitialized pointers.
BLKP 16, 0 ; Allocate 16 pointers and initialize them to 0.
```

BLKW Declaration Type

Syntax

`BLKW` number of words (16 bits each) [, <*init_value*>]

Example

```
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`

Syntax

`DB` byte data (8 bits)

Example

```
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.

DD Declaration Type

Synonym

`.double`

Syntax

`DD` double signed fractional (32 bits)

Examples

```
DD 0.1,0.2 ; Reserve space for 2 double word signed fractional
           ; numbers.  Initialize the first with a 0.1 and
           ; the last with a 0.2.
DD 0.5 ; Reserve space for 1 double word signed fractional
       ; number and initialize it to 0.5.
```

DF Declaration Type

Synonym

`.float`

Syntax

`DF` word signed floating-point constant (32 bits)

Examples

```
DF 0.1,0.2 ; Reserve space for 2 word signed
           ; floating-point numbers. Initialize
           ; first with a 0.1 and last with a 0.2.
DF .5 ; Reserve space for 1 word signed
      ; floating-point number and initialize it to 0.5.
```

DL Declaration Type

Synonyms

`.long, long`

Syntax

`DL` long (32 bits)

Example

```
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)

Example

```
DW "Hello World" ; Reserve and initialize 11 words.
DW "Hello" ; Reserve 12 words, initialize 6.
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, which are called "clauses" of the `DEFINE` directive. You must define a segment before you can use it, unless it is a predefined segment. If a clause is not given, the default for that clause is used in the definition. Clauses are described in the following sections:

- "ALIGN Clause" on page 204
- "ORG Clause" on page 204
- "SPACE Clause" on page 205

For more information on the SEGMENT directive, see “Segments” on page 184.

Synonym

.define

Syntax

<*segment_definition*> =>
DEFINE<*ident*>[<*space_clause*>][*align_clause*>][<*org_clause*>]

Example

```
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 boundary, expressed in bytes, must be a power of two (1, 2, 4, 8, and so on).

Syntax

<*align_clause*> => ,ALIGN = <*int_const*>

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 clause.The default is no ORG, and thus the segment is relocatable.

Syntax

<*org_clause*> => ,ORG = <*int_const*>

Examples

```
DEFINE near_data
; Uses the defaults of ROM, byte alignment and relocatable.


DEFINE far_data,SPACE = RAM,ALIGN = 2
; Aligns on a 2-byte boundary, uses RAM space, relocatable.


DEFINE near_code,ORG = %FFF8
```

```
; Uses ROM, 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 7, "eZ80Acclaim! Address Spaces," on page 183 for available spaces.

Syntax

<*space_clause*> => ,SPACE = <*indent*>

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.

NOTE: Any text found after an END directive is ignored.

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. Labels are not allowed in the expression.

Caution

You *cannot* export EQU using the XDEF directive or import using the XREF directive.

Synonyms

.equ, .EQU, EQUAL

Syntax

<*label*> EQU <*expression*>

Example

```
length EQU 6 ; first dimension of rectangle
width  EQU 11; second dimension of rectangle
area   EQU length * width; area of the rectangle
myreg EQU HL ; 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

Syntax

<*include_directive*> => INCLUDE[<*string_const*>]

Example

```
INCLUDE "calc.inc" ; include calc header file
INCLUDE "\test\calc.inc" ; contains a path name
INCLUDE calc.inc ; 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
```

NEWPAGE

Causes the assembler to start a new page in the output listing. This directive has no effect if `NOLIST` is active. No operand is allowed.

Synonyms

`.page`, `PAGE`

Syntax

<*newpage_directive*> => `NEWPAGE`

Example

```
NEWPAGE
```

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 the 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 8, "Predefined Segments," on page 184 for the names of predefined segments.

Synonyms

.section, SECTION

Syntax

<*segment_directive*> => SEGMENT <*ident*>

Example

```
SEGMENT code ; predefined segment
DEFINE data ; user-defined
```

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 just like an `EQU` directive except you are allowed to change the value of the label. In the 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    STRVAR    VAR FFH
000000 FF                   A     9    DB        STRVAR
                            A    10    SEGMENT TEXT
000000                      A    11    L__0:
000000 4641494C 4544        A    12    DB        "FAILED"
000006 00                   A    13    DB        0
                            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    DB        STRVAR
                            A    20    SEGMENT TEXT
000007                      A    21    L__1:
000007 50415353 4544        A    22    DB        "PASSED"
00000D 00                   A    23    DB        0
       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    DB        STRVAR
```

XDEF

Defines a label or list of labels in the current module as external symbols 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*>

Example

```
XDEF label
XDEF label1,label2,label3
```

XREF

Specifies that a label or 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 list*>

Example

```
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 the following table:

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 directives are described in the following sections:

- ".STRUCT and .ENDSTRUCT Directives" on page 211
- ".TAG Directive" on page 213
- ".UNION Directive" on page 214
- ".WITH and .ENDWITH Directives" on page 215

.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 as follows:

```
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` assembler 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.

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
DAYDS 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 as follows:

```
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.

For example, the `COMPUTE_PAY` routine below:

```
COMPUTE_PAY:
;  Enter with pointer to an EMPLOYEE in R2, days in R1
;  Return with pay in R0,R1

 LD     R0,EMPLOYEE.SALARY(R2)
 MULT   RR0
 RET
```

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,R1
```

```
.WITH EMPLOYEE
LD     R0, SALARY(R2)
MULT   RR0
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.

The following conditional assembly directives are allowed:

- "IF" on page 216
- "IFDEF" on page 217
- "IFSAME" on page 218
- "IFMA" on page 218

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 evaluate to 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`, `.$IF`, `.$if, .IFTRUE`

Syntax

IF [<*cond_expression*> <*code_body*>]

[ELIF <*cond_expression*> <*code_body*>]

[ELSE <*code_body*>]

ENDIF

Example

```
IF XYZ ; process code body 0 if XYZ is not 0
  .
  .
  .
<Code Body 0>
  .
  .
ENDIF
IF XYZ !=3 ; process code body 1 if XYZ is not 3
  .
  .
  .
<Code Body 1>
  .
  .
  .
ELIF ABC ; process code body 2 if XYZ=3 and ABC is not 0
  .
  .
  .
<Code Body 2>
  .
  .
  .
ELSE ; otherwise process 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.

Syntax

IFMA <*arg_number*>

<*code_body*>

[ELSE

<*code_body*>]

ENDIF

MACROS

Macros allow a sequence of assembly source lines to be represented by a single assembler symbol. In addition, arguments can be supplied to the macro in order 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" on page 219
- "Concatenation" on page 219
- "Macro Invocation" on page 220
- "Local Macro Labels" on page 220
- "Optional Macro Arguments" on page 221
- "Exiting a Macro" on page 221
- "Delimiting Macro Arguments" on page 222

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

```
add3: MACRO reg1,reg2,reg3
            LD reg1,reg2
            ADD reg1,reg3
            ENDMAC add3
```

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 listing 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

```
add3 A,B,C
```

This macro invocation causes registers B and C to be added, and the result stored in register A.

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
      JR 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 ARG1,ARG2,ARG3
            IFMA 2
            LD ARG1,ARG2
            ELSE
            LD ARG1,ARG3
            ENDIF
            ENDMACRO MISSING_ARG
```

Invocation

```
MISSING_ARG A, ,(HL) ; second argument is not defined
```

Result

```
LD A,(HL)
```

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 MAXEXIT to terminate the macro.

Example

```
RECURS_MAC: MACRO ARG1,ARG2
            IFMA 0
                MACEXIT
            ELSE
              RECURS_MAC ARG2
              DB ARG1
            ENDIF
```

```
ENDMACRO RECURS_MAC
```

Delimiting Macro Arguments

Macro arguments can be delimited by using the current macro delimiter characters defined using the `MACDELIM` directive. The delimiters can be used to include commas and spaces that are not normally allowed as part of an argument. The default delimiters are brackets `{ }`, but braces `[ ]` and parentheses `( )` are also allowed.

Example

```
BRA: MACRO ARG1
     JP ARG1
     ENDMAC LJMP
```

Invocation

```
BRA {dummy,X}
```

Result

```
JP dummy,X
```

LABELS

Labels are considered symbolic representations of memory locations and can be used to reference that memory location within an expression. See "Label Field" on page 188 for the form of a legal label.

The following sections describe labels:

- "Anonymous Labels" on page 222
- "Local Labels" on page 223
- "Importing and Exporting Labels" on page 223
- "Label Spaces" on page 223

Anonymous Labels

The ZDS II assembler supports anonymous labels. The following table lists the reserved symbols provided for this purpose.

Table 10. Anonymous Labels

Symbol	Description
$$	Anonymous label. This symbol can be used as a label an arbitrary number of times.

Table 10. Anonymous Labels (Continued)

Symbol	Description
$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 of the segment where the `EXTERN` directive was encountered is used as the location of the label.

ADDRESSING MODES

This section discusses the addressing modes supported by the eZ80Acclaim! Macro Assembler.

Table 11. eZ80Acclaim! Addressing Modes

Addressing Modes	Symbolic Name	Notes
8-Bit Register Mode	A	
	B	
	C	
	D	
	E	
	H	
	L	
16/24-Bit Register Mode	BC	
	DE	
	HL	
	IX	
	IY	
Indirect Register Mode	(BC)	
	(DE)	
	(HL)	
	(IX)	
	(IY)	
Indirect Register Mode Plus Offset	(IX+dd)	dd must be in the range of $-128 \leq dd \leq 127$.
	(IY+dd)	dd must be in the range of $-128 \leq dd \leq 127$.
16/24-Bit Direct Addressing Mode	xxxxH or xxxxxxH	For 24-bit values, ADL must equal 1.
PC Relative Addressing Mode	PC	
Single-Bit Flags	ADL	Not directly accessible but can be changed by suffixed JP and CALL instructions.
	IEF1	
	IEF2	
	MADL	

Table 11. eZ80Acclaim! Addressing Modes (Continued)

Addressing Modes	Symbolic Name	Notes
	F	Only used as part of the AF or AF' register pair. The F register contains the six status bits used by the eZ80Acclaim!.
Register Pair Addressing Mode	AF	Only used with PUSH and POP instructions
	AF'	Only used in `EX AF,AF'` instruction
8-Bit Control Registers	I	
	IXH	
	IXL	
	IYH	
	IYL	
	MBASE	Only used in `LD A,MB` instruction where MB stands for MBASE
	R	
	SP	The ADL mode indicates whether SPS or SPL is actually used.

Representing Immediate Value

The following are examples of instructions of the `ld r, n` type:

```
1. ld a, 1*2
2. ld a, #1*2
3. ld a, #(1*2)
```

The eZ80Acclaim! assembler accepts an optional prefix character # for the immediate addressing mode as given in (1) and (2).

For the third case, the prefix character # is mandatory because it conflicts with the indirect addressing mode. For example:

`ld a, (1*2)  ;3A0200`	Assembled as `ld a, (Mmn)`
`ld a, #(1*2) ;3E02`	Assembled as `ld a, n`

SOURCE LANGUAGE SYNTAX

The syntax description that follows is given 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`)
	=>	`NEWPAGE`
	=>	`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>	=>	BFRACT
	=>	BLKB
	=>	BLKL
	=>	BLKP
	=>	BLKW
	=>	DB
	=>	DD
	=>	DF
	=>	DL
	=>	DW
	=>	DW24
	=>	FRACT
	=>	UBFRACT

	=>	UFRACT
<*elements*>	=>	[<*int_const*>]
<*initvalue*>	=>	['['<*instances*>']']<*value*>
<*instances*>	=>	<*int_const*>
<*value*>	=>	<*expression*>|<*string_const*>
<*expression*>	=>	'('<*expression*>')'
	=>	<*expression*><*binary_op*><*expression*>
	=>	<*unary_op*><*expression*>
	=>	<*int_const*>
	=>	<*float_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*'
<*float_const*>	=>	<*decfloat*>
<*decfloat*>	=>	<*float_frac*>|<*float_power*>
<*float_frac*>	=>	<*float_const*>[<*exp_part*>]
<*frac_const*>	=>	*digit*|'_') . (*digit*|'_')@
<*exp_part*>	=>	`E`['+'|-]*digit+*

<float_power>	=>	*digit*(*digit*|'_')@<*exp_part*>
<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>	=>	(`RDATA|XDATA|ROM`)

COMPATIBILITY ISSUES

Compatibility between eZ80Acclaim! assembler directives and those of other assemblers supported by the eZ80Acclaim! assembler are listed in “Asssembler Compatibility Issues” on page 416. If you are developing new code for the eZ80Acclaim!, it is recommended that you use the eZ80Acclaim! directives described previously in this chapter because the behavior of these directives is thoroughly validated with each release of the eZ80Acclaim! assembler.

TROUBLESHOOTING THE ASSEMBLER

There are several instructions that you must be careful about when you use them. For example, you can run into problems with instructions that have implicit registers (such as `DJNZ`, `LDI`, `LDIR`, `LDD`, `LDDR`, and `CPI CPIR`).

Arithmetic instructions on register pairs (like `DEC BC`) do not set the condition codes; however, single-register instructions (like `DEC B` or `DEC` C) do set the condition codes.

Be careful with handling register pairs in ADL mode because the highest-order byte is not directly accessible.

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.

413 Illegal floating point expression.

An illegal value was found in a floating-point expression. This error is normally caused by the use of labels in the expression.

414 Illegal floating point initializer in scalar directive.

You cannot use floating-point values in scalar storage directives.

415 Illegal relocatable initialization in float directive.

You cannot use relocatable labels in a float storage 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 needed or vice versa.

439 Vector not recognized.

The vector name must be `IRQ0`, `IRQ1`, `IRQ2`, `IRQ3`, `IRQ4`, `IRQ5`, or `RESET`

442 Missing delay slot instruction.

Add a delay slot instruction such as BRANCH or LD.

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.

446 Missing `.$wend` at EOF.

There is no `.$wend` directive.

447 Missing `.$repeat` at EOF.

There is no matching `.$repeat` for the `.$while` directive.

448 Segment stack overflow.

Do not allocate returned structures on the stack.

450 Floating point precision error.

The floating-point value cannot be represented to the precision given. The value is rounded to fit within the allowed precision.

451 Floating point over/under flow.

The floating-point value cannot be represented.

452 General floating point error.

The assembler detects an expression operand that is out of range for the intended field and generates appropriate error messages.

453 Fractional number too big/small.

The fractional number cannot be represented.

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 needs to 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.

Using the Linker/Locator

The eZ80Acclaim! 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 are 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 ZiLOG Developer Studio II.

The following sections describe the linker/locator:

- "Linker Interactions with the Compiler and Assembler" on page 236
- "Linker Configurations" on page 240
- "Invoking the Linker" on page 246
- "Linker Commands" on page 247
- "Linker Expressions" on page 258
- "Using Modified ZDS II Startup Modules" on page 264
- "Sample Linker Map File" on page 267
- "Troubleshooting the Linker" on page 283
- "Warning and Error Messages" on page 286

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 using the Command Processor, see "Using the Command Processor" on page 387.

The following six major types of objects are manipulated during the linking process:

- Modules

  Modules are created by assembling a file with the assembler or compiling a file with the compiler.

- Libraries

  Object libraries are collections of object modules created by the Librarian.

- Segments

  A segment is a named logical partition of data or code that forms 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" on page 184 for more information about using predefined segments, defining new segments, and attaching code and data to segments.

The memory range for the default segments depend on the particular eZ80Acclaim! family member. The memory available to your application is given by the linker address spaces, which are described next and whose configuration through the IDE is covered in "Project Settings—Address Spaces Page" on page 89.

- Address spaces

  An address space, as this term is used in the linker, corresponds to a logical block of memory on the target processor that is used for a broad functional purpose. These "logical" address spaces are often correlated with the physical types of memory available to the processor. However, the address space names used by the linker really refer just to the way the memory is intended to be used in your application and do not necessarily reside in the physical memory type that are expected from their names. As an example, in the eZ80Acclaim!'s All RAM Link Configuration, both the RAM address space (used for data storage) and the ROM address space (used for code storage) are physically mapped to RAM memory. In this case, the address spaces partition that physical memory into two logical address spaces. For more information about address spaces, see "Address Spaces" on page 183.

- Groups

  Groups are collections of logical address spaces. They are typically used to conveniently handle a set of address spaces. For example, in some project configurations, the ROM and RAM address spaces are grouped together to form the MEMORY group.

- Copy segments

  Copy segments are segments of initialized data that are re-assigned to another address space (the copy segment) by the linker. A startup routine typically moves data from the copy segment into the original address space to initialize data before invoking a program.

NOTE: 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. For more information, see "BASE OF Versus LOWADDR OF" on page 260.

LINKER INTERACTIONS WITH THE COMPILER AND ASSEMBLER

The ZiLOG linker enables developers to control how source code is loaded into a target processor. This section provides an overview of what the linker does and how it interacts with the ZDS II compiler and assembler and with target hardware. For details of linker command syntax, see "Linker Commands" on page 247.

The ZDS II compiler and assembler use a hierarchical processor model containing spaces and segments. Each space has a range associated with it that identifies valid addresses for that space. For example, Space A might be associated with the addresses from `100h` to `7FFFh`.

Each object file, called a module, contains several segments. See the following two figures. The linker performs the following functions:

• It resolves all external references in each module.

• It assigns (or locates) physical hardware memory addresses into which each segment will be loaded.

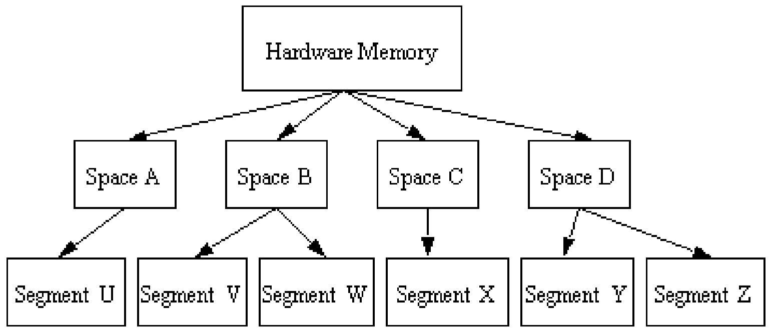


Figure 89. Linker Segments

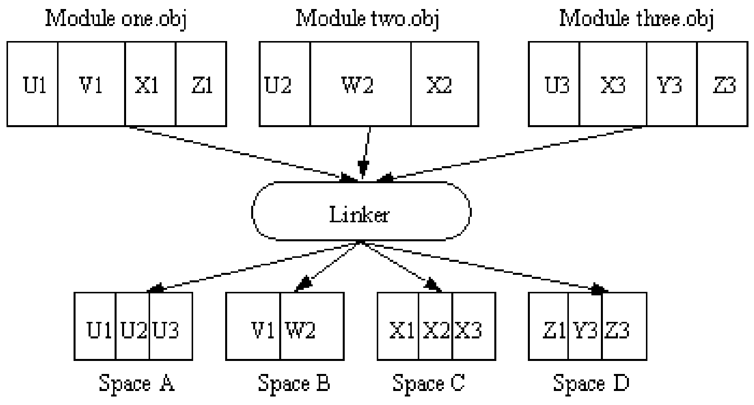


Figure 90. Modules and Segments

Because the linker both links names and locates modules, it is referred to as a linker/locator. The locator part of the linker assigns segments to spaces. In Figure 90, three modules are shown (`one.obj`, `two.obj`, and `three.obj`), each module containing a combination of U, V, W, X, Y, and Z segments.

The linker has no information about the source of the object (`*.obj`) modules but processes a sequence of modules (in alphabetical order, by default), resolving references between the modules and then loading the segments into the appropriate address spaces. The resolution and loading occurs after the linker has read in the linker command file. This file, containing commands that control linker actions such as the assignment of segments to spaces, is automatically generated by ZDS II when a project is built for the first time. You can then edit the linker command file to change linker behavior if needed. The linker command file gives a description of how to link the modules, providing a non-procedural description of the linking process. The elements of a typical linker command file for a C project are discussed in "Linker Command Files for C Programs" on page 158.

eZ80Acclaim! Address Spaces

Every byte generated by the compiler or assembler eventually gets assigned to an address space by the linker. Address spaces represent physical or logical architecture divisions. Each space has physical or logical attributes that characterize it. A space might, for example, use special addressing modes or have alignment restrictions.

Spaces can also indicate separate physical memories. Within each space, segments can be defined that can affect instruction speed or availability.

The ZDS II compiler and assembler for eZ80Acclaim! predefine four address spaces:

- ROM (read-only space)
- RAM (random-access/writable space)
- EXTIO (16-bit addressable space for I/O)
- INTIO (8-bit addressable space for I/O)

These spaces contain the *logical* components of an application. The linker command file specifies how these logical spaces are mapped to the *physical* addresses on the target board. For more information about address spaces, see "Address Spaces" on page 183.

Segments

The ZDS II compiler and assembler define the following segments:

- .IVECTS (segment type for interrupt vectors; address space is ROM)
- .RESET (segment type for reset handler code; address space is ROM)
- CODE (segment type for CODE; address space is ROM)

  This is the default segment for assembler code.
- BSS (segment type for uninitialized data; address space is RAM)

- DATA (segment type for initialized data; address space is RAM)
- STRSECT (segment type for string constants; address space is RAM by default)
- TEXT (segment type for constant data; address space is RAM by default)

The default relationships of these segments to the four address spaces are shown in the following figure.

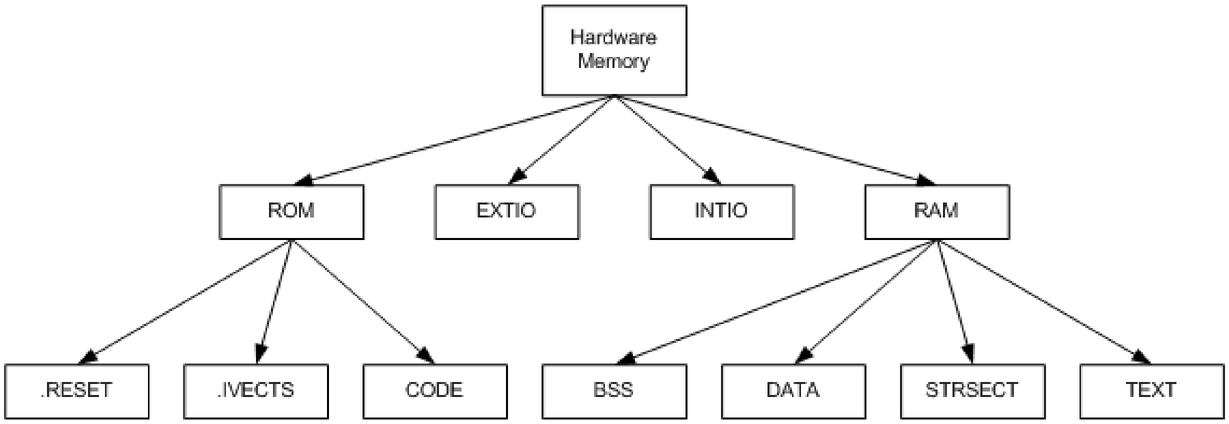


Figure 91. Segment and Space Relationships

The address spaces listed in "eZ80Acclaim! Address Spaces" on page 238 are those used for the run-time addresses of the segments. Some segments can be copied from one space to another, depending on the memory configuration (described in "Linker Configurations" on page 240). For example, in a typical configuration, the DATA segment is placed in ROM at load time and then copied to RAM on application reset; it is the copy in RAM that is actually used when the program runs, but the ROM version is necessary so that the program can be re-initialized at reset.

The address spaces listed in "eZ80Acclaim! Address Spaces" on page 238 define the logical spaces to which the segments are assigned. The relationship between the logical RAM/ROM spaces and physical RAM or ROM memory depends on your system configuration and might be more complicated. For more information, see "Link Configuration" on page 77.

You have complete control over the order in which these segments are loaded into the RAM address space.

You can define additional CODE segments in C by marking the One Code Segment Per Module check box on the Advanced page of the Project Settings dialog box (see "Project Settings—Advanced Page" on page 66). For the `myfile.c` module, the code segment is given the name `myfile_TEXT` ("text" refers to code, unlike the TEXT segment that contains constant data). All of the procedures in the `myfile_TEXT` module are assigned to that same segment, and the segment is assigned to ROM space. All of the uninitialized data, initialized data, and string initialization remain assigned to BSS, DATA, and STRSECT, respectively.

In addition to these segments, the linker must also determine the location of the following components:

- the C stack
- the C heap for `malloc/free`
- the code that sets up the run-time environment
- the vector table

"Components Used in All Linker Configurations" on page 245 describes how to set these locations.

LINKER CONFIGURATIONS

The Link Configuration drop-down list box (see "Link Configuration" on page 77) allows you to set up linker parameters related to the memory map. You can also select the linker configuration when you create a new project. The following linker configurations are available:

- "Standard Configuration" on page 241
- "All RAM Configuration" on page 242
- "Copy to RAM Configuration" on page 244
- "Custom Configuration" on page 244
- "Deprecated Custom Configuration" on page 245

You can select one of these configurations using the Link Configuration drop-down list box. The diagrams in the following figure schematically represent the most common physical memory configurations. For example, some boards only have physical RAM on them (All RAM), while some have both ROM and RAM (Standard). (The EXTIO and INTIO spaces are not shown in these diagrams. Those spaces are logically distinct from the ROM and RAM spaces and are accessed only by special instructions used for that purpose.)

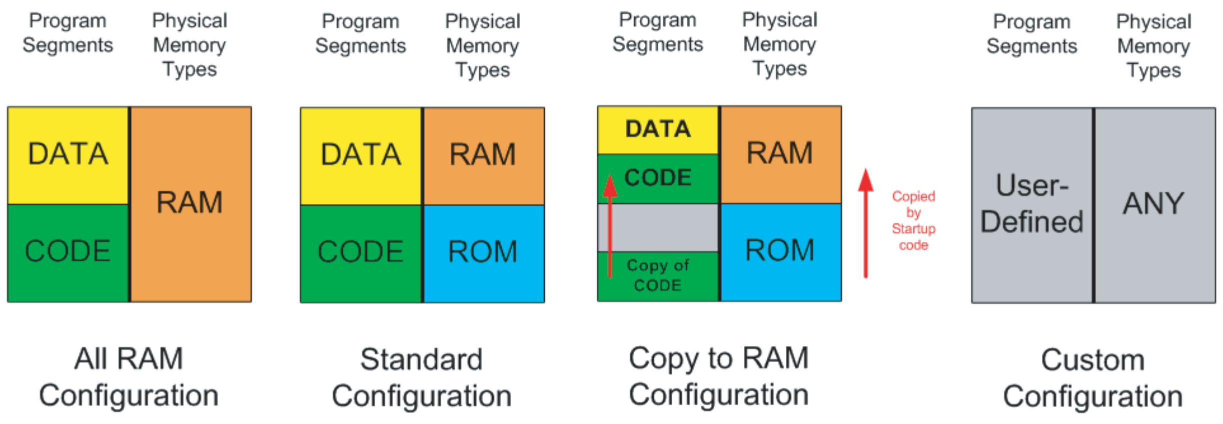


Figure 92. Linker Configurations

The preceding figure does not show the addresses for the beginning and end of physical ROM and RAM. You can enter the address information on the Address Spaces page (see "Project Settings—Address Spaces Page" on page 89). The linker configuration and Address Spaces page information represent the hardware memory map.

As an example of what the linker command file does to support the linker configurations, suppose your hardware only has RAM (the All RAM configuration). The linker therefore must map logical ROM space to some part of physical RAM. In a 64K RAM configuration, you might specify a RAM space of `0000h–FFFFh`. The linker command file contains commands generated by ZDS II that group the ROM and RAM spaces together.

In most embedded applications, the logical ROM space resides in physical (hardware) ROM, and the logical RAM space resides in physical RAM. For this (Standard) configuration, the ROM and RAM memory spaces map directly to the corresponding ROM and RAM hardware addresses. For example, in an architecture with a 24-bit addressing range, you might have RAM at `C00000h–FFFFFFh` and ROM at `0h–00FFFFh`.

In the Copy to RAM configuration, the hardware has both ROM and RAM. However, performance or operational considerations require that the code and data in physical ROM be copied to a physical RAM address upon startup (to accommodate, for example, initialized data that needs to be re-initialized when the board is reset). The Copy to RAM configuration causes the startup code to copy some part of the ROM to a location in RAM.

Standard Configuration

In the Standard configuration, the hardware has both physical RAM and physical ROM. The linker commands generated by ZDS II map logical RAM segments to physical RAM and logical ROM segments to physical ROM. If the memory fields in the New Project

Wizard dialog box or the Address Spaces page of the Project Settings dialog box contained the following values:

ROM: `0-7FFF`

RAM: `A000-FFFF`

ZDS II then generates the following code for the linker command file:

```
RANGE ROM $0 : $7FFF
RANGE RAM $A000 : $FFFF
```

The linker uses the COPY command on segments to better support standalone C programs. When running a C program under an operating system such as Windows or UNIX, all initialized variables are set to their starting values upon the start of program execution. In a standalone C implementation, however, there is no operating system to reload variables with their initial values. With no operating system, if an embedded application runs for a while and is then restarted at `main()`, the values of initialized variables are not restored to their original value. The linker's COPY command, together with the startup module, provides a means to reinitialize variables.

The linker normally loads the DATA segment (initialized data) into RAM. When the initialized data has been loaded into RAM and modified by program execution, in the absence of the COPY mechanism, the only way to reload the initial DATA segment is to download the code to the target board again. This approach is not practical for most embedded systems. (The embedded application would have to save all the initialized data and reload the initial values upon RESET, for example.)

The COPY command (typically, `COPY DATA ROM`) causes the linker to put a copy of the DATA segment in the ROM space at load time. The standard startup module always copies the DATA segment to RAM before calling `main()`. The COPY command copies segments into spaces only. Any other copy combination generates an error.

The startup module requires additional linker commands to perform the copy. "Components Used in All Linker Configurations" on page 245 describes these commands.

The ORDER command allows you to define the sequence of segments within a memory space. In the Standard Configuration, ZDS II generates the following command to put the initialized data segment at a lower address than the uninitialized data segment:

```
ORDER DATA,BSS
```

All RAM Configuration

In this configuration, the linker maps all segments associated with the logical ROM address space to physical RAM. ZDS II therefore automatically generates two linker commands. One command combines the spaces into one, and the other defines the physical address range for the combined spaces. The two linker commands are:

- `GROUP MEMORY=ROM,RAM`

This command defines a new address space (MEMORY) that contains the existing logical address spaces RAM and ROM.

- `RANGE MEMORY $0 : $FFFF`

  This command defines the valid addresses for the new space.

ZDS II creates the RANGE command starting with the lowest address specified in the New Project Wizard dialog box or the Address Spaces page of the Project Settings dialog box and ending with the highest address. The lowest address is `min (min(ROM), min(RAM))` and the highest address is `max (max(ROM),max(RAM))`. These two ZDS II-generated commands are critical for building an All RAM configuration.

Both the GROUP MEMORY and RANGE MEMORY commands are required for the All RAM configuration. A common error made by users trying to set up this configuration manually is to omit the RANGE MEMORY command. To illustrate the effects of such an error, suppose the New Project Wizard dialog box or the Address Spaces page of the Project Settings dialog box has the following values for a 64K memory machine:

ROM: `0-7FFF`

RAM: `8000-FFFF`

If the All RAM configuration is not used to create the linker command file, ZDS II converts the values in the New Project Wizard dialog box or the Address Spaces page of the Project Settings dialog box to the following linker commands:

```
RANGE ROM $0 : $7FFF
RANGE RAM $8000 : $FFFF
```

which, if followed by

```
GROUP MEMORY=ROM,RAM
```

results in ROM segments starting at address `$0` and RAM segments starting at `$8000`, potentially wasting space if the ROM segments do not fully occupy the range from `$0 - $7FFF`. The following command

```
RANGE MEMORY $0 : $FFFF
```

overrides the RANGE commands for the component address spaces and binds the RAM segments immediately after the ROM segments.

NOTE: The names following the = in the GROUP command define an ordering for the new GROUP. In the preceding example, all of the ROM segments are allocated memory at lower addresses than the RAM segments. However, the following GROUP statement locates all of the RAM segments first:

```
GROUP MEMORY=RAM,ROM
```

Copy to RAM Configuration

This configuration provides support for copying code as well as data segments from physical ROM to physical RAM. The idea is to compensate for the performance penalty often associated with running code from ROM. A ROM instruction fetch, for example, might require more wait states than a RAM instruction fetch. It is therefore more efficient to run code from RAM than to run it from ROM, provided the target system has enough RAM for the program code and data. As in the other configurations, ZDS II automatically generates linker commands in the linker command file to support this operating mode when Copy to RAM is selected.

To run code from the RAM space, the linker must do two things:

- Reassign all of the CODE segment addresses to RAM instead of ROM.
- Place a copy of the CODE segment in ROM for an application restart (the startup module re-copies it from ROM to RAM upon restart).

The linker CHANGE command allows you either to rename a segment or reassign a segment to another space. For example,

```
CHANGE TEXT is DATA
CHANGE CODE is RAM
CHANGE STRSECT is CODE
```

causes the linker to do the following:

1. Combine the TEXT segment into the DATA segment.
2. Reassign the CODE segment to the RAM space.
3. Combine the STRSECT segment into the CODE segment.

These three CHANGE commands reassign all addresses in the CODE, TEXT, and STRSECT segments into RAM for the fastest possible execution.

The final step requires the RAM space to be copied in ROM space so that it can be reloaded from ROM when the application starts. Adding the following commands

```
COPY DATA ROM
COPY CODE ROM
```

causes both the CODE and DATA segments to be copied to ROM. These segments are then copied by the startup code to the appropriate RAM addresses, completing the actions associated with the COPY to RAM configuration.

Custom Configuration

In this configuration, you define all of the segment to space mappings, depending on the hardware configuration of the application board.

Deprecated Custom Configuration

This configuration was provided in earlier releases of ZDS II for eZ80Acclaim! before the 4.10.0 release. It is supported for backward compatibility. In those earlier releases, this configuration was called “Custom,” but, in fact, it was nearly identical to the Standard configuration. The only difference was that the ordering of segments was left to the user and that the `STRSECT` segment was left in RAM instead of ROM. That segment contains string literals and should always be placed in ROM for production code but can be kept in RAM at times for debugging.

In the 4.10.0 and subsequent releases of ZDS II, the Custom link configuration is used to support truly customized link command files. ZiLOG recommends that if you have an older project that used the Deprecated Custom configuration, you convert it either to the Standard configuration (by changing to that setting and verifying that the minor changes in link commands cause you no problems) or to the Custom configuration (by saving your existing link command file and selecting the Use Existing button (see “Use Existing” on page 83). At some future time, the Deprecated Custom configuration might no longer be supported in ZDS II.

Components Used in All Linker Configurations

In a standalone application, components usually managed by an operating system must be set up and managed as part of the C runtime. In particular, an embedded C application might use the following components:

- Stack
- Heap
- Other segments
- Startup

The location of the stack and heap depend on the size of the application and the amount of physical RAM available. Generally, the heap starts at the next address just beyond the last allocated address in the RAM space and grows towards higher numbered addresses. The stack generally starts at the highest possible address in physical RAM and grows towards lower numbered addresses. The linker provides expressions that simplify placing the stack and heap at appropriate addresses. For example, the linker `TOP OF RAM` expression represents the address of the last byte allocated in the RAM space. The following expression can be used to set the lowest address of the heap:

```
define __heapbot = top of RAM + 1
```

Similarly, the linker `HIGHADDR OF RAM` command represents the highest numbered physical address in the RAM space. Setting the starting address for the stack can be done with the following:

```
define __stack = HIGHADDR of RAM
```

Both of these linker commands define locations that are used by the startup module to initialize the C runtime for an application. In configurations where code or data is copied, the linker commands for setting up the stack and heap can be more complicated. When a configuration has been selected, ZDS II automatically generates the appropriate DEFINEs and linker expressions. In addition to the stack and heap, the startup routines handle all of the copying for special segments and setting up the initial vector table.

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 “Build Toolbar” 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 (see “Project Settings—Objects and Libraries Page” on page 84).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 “Always Generate from Settings” on page 81). You can add additional linker commands with the Additional Linker Directives dialog box (page 82). If you want to override the automatically generated linker command file, select the Use Existing button (see “Use Existing” on page 83).

The linker can also be invoked from the DOS command line or through the ZDS II Command Processor. For more information on invoking the linker from the DOS command line, see “Running ZDS II from the Command Line” on page 379. To invoke the linker through the ZDS II Command Processor, see “Using the Command Processor” on page 387.

Caution

If you select the Included in Project button *and* use the startup source or object provided with the tools (`init_params_f91.asm`, which includes `sysclk.asm` or corresponding `.obj` files), building the project might result in linker errors and warnings. If you experience this problem, follow these steps to add the correct linker directives to initialize variables in your project:

1. Open your project.
2. From the Project menu, select **Settings**.
3. In the Objects and Libraries page, select the Standard button and click **OK**.
4. Click **Yes** to the warning message: `"The project settings have changed since the last build. Would you like to rebuild the affected files?"`

   The linker command file is generated even if the build is not successful.

5. Open the generated linker command file (*project_name*`.linkcmd`) and copy everything from `CHANGE STRSECT is CODE` to the fifth blank line or before this line:

   ```
   "C:\Program
   Files\ZiLOG\ZDSII_eZ80Acclaim!_x.x.x\samples\StarterProject\sta
   rter"= \
   ```
6. Paste the directives into the Additional Linker Directives dialog box ("Additional Directives" on page 82).
7. Click **OK** to return to the Project Settings dialog box.
8. Click **OK** to save your settings.
9. Click **Yes** to the warning message: `"The project settings have changed since the last build. Would you like to rebuild the affected files?"`
10. Reset your project settings and rebuild your project.

LINKER COMMANDS

The following sections describe the commands of a linker command file:

- "<outputfile>=<module list>" on page 248
- "CHANGE" on page 248
- "COPY" on page 249
- "DEBUG" on page 251
- "DEFINE" on page 251
- "FORMAT" on page 251
- "GROUP" on page 252
- "HEADING" on page 252
- "LOCATE" on page 252
- "MAP" on page 253
- "MAXHEXLEN" on page 253
- "MAXLENGTH" on page 253
- "NODEBUG" on page 254
- "NOMAP" on page 254
- "NOWARN" on page 254
- "ORDER" on page 254

- “RANGE” on page 255
- “SEARCHPATH” on page 255
- “SEQUENCE” on page 256
- “SORT” on page 256
- “SPLITTABLE” on page 256
- “UNRESOLVED IS FATAL” on page 257
- “WARN” on page 257
- “WARNING IS FATAL” on page 258
- “WARNOVERLAP” on page 258

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_eZ80Acclaim!_4.10.0\lib\zilog\vectors24.obj, \
          c:\ZDSII_eZ80Acclaim!_4.10.0\lib\zilog\init_params_f91.obj, \
          c:\ZDSII_eZ80Acclaim!_4.10.0\lib\zilog\cstartup.obj, \
          test.obj, \
          c:\ZDSII_eZ80Acclaim!_4.10.0\lib\std\chelpD.lib, \
          c:\ZDSII_eZ80Acclaim!_4.10.0\lib std\crtD.lib, \
          c:\ZDSII_eZ80Acclaim!_4.10.0\lib\std\fpdumy.obj
```

This command links the five object modules and two library modules to generate the linked output file, which will be called `sample.lod` if the IEEE 695 format has been selected.

NOTE: In the preceding example, the \ (backslash) at the end of every line except the last line allows the <*module list*> to extend over several lines. 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.

Examples

NOTE: See also the examples for the `COPY` command ("COPY" on page 249).

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
```

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 is copied to `NULL` as a command to the linker, the segment is deleted from the linking process. This can be useful if you need to 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.

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, use 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 need to copy the `codeseg` contents from ROM to RAM at run time as part of the startup 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 RAM, use 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 RAM
   ```

   The linker renames `strseg` and `text` as `codeseg` and performs linking as described in the previous example. You need to 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, use the following procedure:

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 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 needed for linking and debugging.

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 is copied to `NULL` as a command to the linker, the segment is deleted from the linking process. This can be useful if you need to 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.

Refer to "Linker Expressions" on page 258 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 format 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 at assembly 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 "Linker Expressions" on page 258 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 Hex	`INTEL32`	`.hex`

The default setting is `IEEE 695`.

Syntax

[-]`FORMAT=`*<type>*

Example

```
FORMAT = 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.

NOTE: The names given after the = sign will be grouped together in order, from lower to higher numbered addresses.

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 $10000
```

NOTE: Refer to "Linker Expressions" on page 258 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.

In the link map file, the C prefix indicates Code, and the D prefix indicates Data.

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" on page 267.

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 "Linker Expressions" on page 258 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 CODE $100 : $1FF,$300 : $3FF
```

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 "Linker Expressions" on page 258 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 as follows.

1. Current directory
2. Environment path
3. Search path

Syntax

`SEARCHPATH ="`*<path>*`"`

Example

```
SEARCHPATH="C:\ZDSII_eZ80Acclaim!_4.10.0\lib\std"
```

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 code,xdata
```

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 need to take one of the 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 needed in which case multiple segments can be marked splittable

Syntax

`SPLITTABLE` *segment_list*

Example

```
SPLITTABLE DATA, 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

The following sectiona describe the operators and operands that form legal linker expressions:

- "+ (Add)" on page 259
- "& (And)" on page 260
- "BASE OF" on page 260
- "COPY BASE" on page 261
- "COPY TOP" on page 261
- "/ (Divide)" on page 261

- "FREEMEM" on page 261
- "HIGHADDR" on page 262
- "LENGTH" on page 262
- "LOWADDR" on page 262
- "* (Multiply)" on page 262
- "Decimal Numeric Values" on page 262
- "Hexadecimal Numeric Values" on page 263
- "| (Or)" on page 263
- "<< (Shift Left)" on page 263
- ">> (Shift Right)" on page 263
- "- (Subtract)" on page 263
- "TOP OF" on page 264
- "^ (Bitwise Exclusive Or)" on page 264
- "~ (Not)" on page 264

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 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 EXTIO $0 : $FFFF
RANGE INTIO $0 : $FF
RANGE ROM $0 : $1FFFF
RANGE RAM $5000 : $1FFFF
RANGE s_checksum $0 : $FFFF
RANGE s_nvrblock $5000 : $1FFFF

/* RAM allocation */
LOCATE s_uninit_data at $5000
LOCATE BSS at (TOP OF s_uninit_data)+1
LOCATE s_nvrblock at $FE00
DEFINE __low_data = BASE OF s_uninit_data
DEFINE __copy_code_to_ram = 0
```

Using

```
LOCATE s_uninit_data at $5000
```

or

`LOCATE s_uninit_data at LOWADDR OF RAM`

gives the same address (the lowest possible address) when `RANGE RAM $5000:$1FFFF`.

If

`LOCATE s_uninit_data at $5000`

is changed to

`LOCATE s_uninit_data at BASE OF RAM`

the lowest used address is `$FE00` (because `LOCATE s_nvrblock at $FE00` 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.

See “BASE OF Versus LOWADDR OF” on page 260 for an explanation of the difference between these two operators.

* (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*>

USING MODIFIED ZDS II STARTUP MODULES

This section explains what steps you need to take to use modified C startup modules in different eZ80Acclaim! linker configurations.

The following ZDS II startup modules are required as part of correctly initializing the C run-time environment for the eZ80®, as discussed in "Startup Files" on page 157:

- `src\boot\common\vectors16.asm` for eZ80190, eZ80L92, eZ80F92, and eZ80F93 devices
- `src\boot\common\vectors24.asm` for eZ80F91

- `src\<device>\init_params_<device>.asm` (for example, `init_params_f91.asm` for eZ80F91)
- `src\boot\common\cstartup.asm`

The corresponding object files are included by ZDS II for the default boot module project setup. Some users might find they need to customize the startup for their particular project. Customizing these standard startup modules requires the following steps:

1. Copy the above files to the project directory.
2. Add those files to the project.
3. Select the Included in Project button in the Objects and Libraries page (see "C Startup Module" on page 86).
4. Add linker directives as described in this section.

The linker directives in "Directives for All Configurations" on page 265 must be added to comply with the ZDS II startup code and to avoid any linker warnings. The Additional Linker Directives dialog box is the place where you need to add these directives (see "Additional Directives" on page 82).

These directives can be classified into four categories:

- Renaming sections (CHANGE)
- Ordering sections (ORDER)
- Section copy (COPY)
- Link time variable definitions (DEFINE)

Directives for All Configurations

The following linker directives need to be included for all linker configurations. The additional directives for each configuration are given in the following sections:

- "Directives for the All RAM Configuration" on page 266
- "Directives for the Standard Configuration" on page 266
- "Directives for the Copy to RAM Configuration" on page 266
- "Directives for the Custom Configuration" on page 267

Those additional directives can be added at the end of these common directives:

```
/* Segment order */
ORDER .RESET,.IVECTS,.STARTUP,CODE,DATA

/* DATA segment initialization */
COPY DATA ROM   /* DATA copy at ROM */
DEFINE __low_romdata = copy base of DATA
DEFINE __low_data = base of DATA
DEFINE __len_data = length of DATA

/* BSS segment initialization */
DEFINE __low_bss = base of BSS
DEFINE __len_bss = length of BSS

/* Copy code to RAM */
DEFINE __low_romcode = copy base of CODE
DEFINE __low_code = base of CODE
DEFINE __len_code = length of CODE
```

Directives for the All RAM Configuration

Apart from the common directives, the following needs to be added for this configuration:

```
/* Stack pointer initialization */
DEFINE __stack = highaddr of MEMORY + 1

/* Required for malloc() and free() */
DEFINE __heaptop = highaddr of MEMORY
DEFINE __heapbot = top of MEMORY + 1

/* This is set to 1 only for Copy To RAM configuration */
DEFINE __copy_code_to_ram = 0
```

Directives for the Standard Configuration

Apart from the common directives, the following needs to be added for this configuration:

```
/* Renaming STRSECT as ROM to place it in ROM - default is RAM */
CHANGE STRSECT is ROM

/* Stack pointer initialization */
DEFINE __stack = highaddr of RAM + 1

/* Required for malloc() and free() */
DEFINE __heaptop = highaddr of RAM
DEFINE __heapbot = top of RAM + 1

/* This is set to 1 only for Copy To RAM configuration */
DEFINE __copy_code_to_ram = 0
```

Directives for the Copy to RAM Configuration

Apart from the common directives, the following needs to be added for this configuration:

```
/* Binding CODE, STRSECT segments in RAM */
CHANGE CODE is RAM
CHANGE STRSECT is CODE

/* CODE copy at ROM */
```

```
COPY CODE ROM

/* Stack pointer initialization */
DEFINE __stack = highaddr of RAM + 1

/* Required for malloc() and free() */
DEFINE __heaptop = highaddr of RAM
DEFINE __heapbot = top of RAM + 1

/* Copy CODE to RAM */
DEFINE __copy_code_to_ram = 1
```

Directives for the Custom Configuration

Apart from the common directives, the following needs to be added for this configuration:

```
/* Stack pointer initialization */
DEFINE __stack = highaddr of RAM + 1

/* Required for malloc() and free() */
DEFINE __heaptop = highaddr of RAM
DEFINE __heapbot = top of RAM + 1

/* This is set to 1 only for Copy To RAM configuration */
DEFINE __copy_code_to_ram = 0
```

SAMPLE LINKER MAP FILE

```
IEEE 695 OMF Linker Version 6.20 (06030104)
Copyright (C) 1999-2004 ZiLOG, Inc. All Rights Reserved

LINK MAP:

DATE:      Fri Mar 03 11:49:44 2006
PROCESSOR: assembler
FILES:     C:\PROGRA~1\ZiLOG\UNDERD~1\ZDSII_~2.0\lib\zilog\vectors24.obj
           C:\PROGRA~1\ZiLOG\UNDERD~1\ZDSII_~2.0\lib\zilog\init_params_f91.obj
           C:\PROGRA~1\ZiLOG\UNDERD~1\ZDSII_~2.0\lib\zilog\cstartup.obj
          .\Buttons.obj          .\LedMatrix.obj        .\LedTimer.obj
           .\main.obj
           .\zsldevinit.obj
           C:\PROGRA~1\ZiLOG\UNDERD~1\ZDSII_~2.0\lib\std\chelpD.lib
           C:\PROGRA~1\ZiLOG\UNDERD~1\ZDSII_~2.0\lib\std\crtD.lib
           C:\PROGRA~1\ZiLOG\UNDERD~1\ZDSII_~2.0\lib\std\fpdumy.obj
           C:\PROGRA~1\ZiLOG\UNDERD~1\ZDSII_~2.0\lib\zilog\gpioD.lib
           C:\PROGRA~1\ZiLOG\UNDERD~1\ZDSII_~2.0\lib\zilog\uartF91simD.lib
```

```
COMMAND LIST:
=============
/* Linker Command File - ledDemo Debug - RAM */

/* Generated by: */
/*  ZDS II - eZ80Acclaim! 4.10.0 (Build 06030104) */
/*  IDE component: b:4.10:06030104 */

/* compiler options */
/* -define:_EZ80F91 -define:_EZ80ACCLAIM! -define:ZSL_DEVINIT */
/* -define:ZSL_DEVICE_PORTB -define:_ZSL_PORT_USED */
/* -define:ZSL_DEVICE_PORTD -define:ZSL_DEVICE_UART0 */
/* -define:_ZSL_UART_USED -define:_SIMULATE -genprintf -NOkeepasm */
/* -NOkeeplst -NOlist -NOlistinc -NOmodsect -optsize -promote */
/* -reduceopt */
/* -stdinc:"D:;..\..\..\include\std;..\..\..\include\zilog" */
/* -usrinc:"..\include" -debug -cpu:eZ80F91 */
/* -asmsw:" -cpu:eZ80F91 -define:_EZ80ACCLAIM!=1 -define:ZSL_DEVINIT=1 -
define:ZSL_DEVICE_PORTB=1 -define:_ZSL_PORT_USED=1 -define:ZSL_DEVICE_PORTD=1
-define:ZSL_DEVICE_UART0=1 -define:_ZSL_UART_USED=1 -define:_SIMULATE=1 -
include:C:\PROGRA~1\ZiLOG\UNDERD~1\ZDSII_~2.0\include\std;C:\PROGRA~1\ZiLOG\UN
DERD~1\ZDSII_~2.0\include\zilog" */

/* assembler options */
/* -define:_EZ80ACCLAIM!=1 -define:ZSL_DEVINIT=1 */
/* -define:ZSL_DEVICE_PORTB=1 -define:_ZSL_PORT_USED=1 */
/* -define:ZSL_DEVICE_PORTD=1 -define:ZSL_DEVICE_UART0=1 */
/* -define:_ZSL_UART_USED=1 -define:_SIMULATE=1 */
/* -include:"..\..\..\include\std;..\..\..\include\zilog" -NOlist */
/* -NOlistmac -name -pagelen:56 -pagewidth:80 -quiet -sdiopt -warn */
/* -debug -NOigcase -cpu:eZ80F91 */

-FORMAT=OMF695,INTEL32
-map -maxhexlen=64 -NOquiet -NOwarnoverlap -NOxref -unresolved=fatal
-sort NAME=ascending -warn -debug -NOigcase

RANGE ROM $000000 : $03FFFF
RANGE RAM $B80000 : $BFFFFF
RANGE EXTIO $0 : $FFFF
RANGE INTIO $0 : $FF

CHANGE STRSECT is ROM

ORDER .RESET,.IVECTS,.STARTUP,CODE,DATA
COPY DATA ROM

DEFINE __low_romdata = copy base of DATA
DEFINE __low_data = base of DATA
DEFINE __len_data = length of DATA
DEFINE __low_bss = base of BSS
DEFINE __len_bss = length of BSS
DEFINE __stack = highaddr of RAM + 1
DEFINE __heaptop = highaddr of RAM
DEFINE __heapbot = top of RAM + 1
DEFINE __low_romcode = copy base of CODE
DEFINE __low_code = base of CODE
DEFINE __len_code = length of CODE
DEFINE __copy_code_to_ram = 0
DEFINE __crtl = 1
DEFINE __CS0_LBR_INIT_PARAM = $10
DEFINE __CS0_UBR_INIT_PARAM = $1f
DEFINE __CS0_CTL_INIT_PARAM = $a8
DEFINE __CS0_BMC_INIT_PARAM = $02
DEFINE __CS1_LBR_INIT_PARAM = $00
DEFINE __CS1_UBR_INIT_PARAM = $07
DEFINE __CS1_CTL_INIT_PARAM = $28
DEFINE __CS1_BMC_INIT_PARAM = $02
DEFINE __CS2_LBR_INIT_PARAM = $80
DEFINE __CS2_UBR_INIT_PARAM = $bf
DEFINE __CS2_CTL_INIT_PARAM = $28
DEFINE __CS2_BMC_INIT_PARAM = $02
DEFINE __CS3_LBR_INIT_PARAM = $00
DEFINE __CS3_UBR_INIT_PARAM = $00
DEFINE __CS3_CTL_INIT_PARAM = $00
DEFINE __CS3_BMC_INIT_PARAM = $02
DEFINE __RAM_CTL_INIT_PARAM = $C0
DEFINE __RAM_ADDR_U_INIT_PARAM = $B7
DEFINE __FLASH_CTL_INIT_PARAM = $60
DEFINE __FLASH_ADDR_U_INIT_PARAM = $00

define _SYS_CLK_FREQ = 20000000
define _OSC_FREQ = 20000000
define _SYS_CLK_SRC = 0
define _OSC_FREQ_MULT = 1
define __PLL_CTL0_INIT_PARAM = $00
define _zsl_g_clock_xdefine = 50000000

"C:\PROGRA~1\ZiLOG\UNDERD~1\ZDSII_~2.0\samples\EZ80F9~1\src\ledDemo"=
C:\PROGRA~1\ZiLOG\UNDERD~1\ZDSII_~2.0\lib\zilog\vectors24.obj,
C:\PROGRA~1\ZiLOG\UNDERD~1\ZDSII_~2.0\lib\zilog\init_params_f91.obj,
C:\PROGRA~1\ZiLOG\UNDERD~1\ZDSII_~2.0\lib\zilog\cstartup.obj,  .\Buttons.obj,
.\LedMatrix.obj,  .\LedTimer.obj,  .\main.obj,  .\zsldevinit.obj,
C:\PROGRA~1\ZiLOG\UNDERD~1\ZDSII_~2.0\lib\std\chelpD.lib,
C:\PROGRA~1\ZiLOG\UNDERD~1\ZDSII_~2.0\lib\std\crtD.lib,
C:\PROGRA~1\ZiLOG\UNDERD~1\ZDSII_~2.0\lib\std\fpdumy.obj,
C:\PROGRA~1\ZiLOG\UNDERD~1\ZDSII_~2.0\lib\zilog\gpioD.lib,
C:\PROGRA~1\ZiLOG\UNDERD~1\ZDSII_~2.0\lib\zilog\uartF91simD.lib



SPACE ALLOCATION:
=================

Space               Base        Top         Size
------------------- ----------- ----------- ---------
RAM                    D:B80000    D:B801D9      1dah
ROM                    C:000000    C:001E79     1e7ah


SEGMENTS WITHIN SPACE:
======================

RAM                 Type                 Base        Top         Size
------------------- -------------------- ----------- ----------- ---------
.IVECTS             normal data             D:B80000    D:B800FF      100h
BSS                 normal data             D:B80115    D:B801D9       c5h
DATA                normal data             D:B80100    D:B80114       15h


ROM                 Type                 Base        Top         Size
------------------- -------------------- ----------- ----------- ---------
.RESET              normal data             C:000000    C:00006A       6bh
.STARTUP            normal data             C:00006B    C:00024F      1e5h
CODE                normal data             C:000250    C:001A5B     180ch
DATA                * segment copy *        C:001E65    C:001E79       15h
STRSECT             normal data             C:001DE0    C:001E64       85h
TEXT                normal data             C:001A5C    C:001DDF      384h


SEGMENTS WITHIN MODULES:
========================

Module: ..\..\..\..\..\..\..\..\eZ8c:ABS.C (Library: crtD.lib) Version: 1:0
03/01/2006 16:41:11

    Name                                        Base        Top         Size
    ---------------------------------------- ----------- ----------- ---------
    Segment: CODE                               C:00168D    C:0016BE       32h


Module: ..\..\..\..\..\..\..\..\eZ8c:GETCHAR.C (Library: crtD.lib) Version:
1:0 03/01/2006 16:41:11
```

```
    Name                                       Base        Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: CODE                             C:0008C8    C:0008D9       12h


Module: ..\..\..\..\..\..\..\..\eZ8c:PRINT_GLOBALS.C (Library: crtD.lib)
Version: 1:0 03/01/2006 16:41:13

    Name                                       Base        Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: BSS                              D:B801A2    D:B801C5       24h
    Segment: DATA                             D:B8010E    D:B80110        3h


Module: ..\..\..\..\..\..\..\..\eZ8c:PRINT_PUTCH.C (Library: crtD.lib)
Version: 1:0 03/01/2006 16:41:13

    Name                                       Base        Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: CODE                             C:001953    C:001970       1eh


Module: ..\..\..\..\..\..\..\..\eZ8c:PRINT_PUTSTRING.C (Library: crtD.lib)
Version: 1:0 03/01/2006 16:41:13

    Name                                       Base        Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: CODE                             C:000ED6    C:001091      1bch


Module: ..\..\..\..\..\..\..\..\eZ8c:PRINT_SPUTCH.C (Library: crtD.lib)
Version: 1:0 03/01/2006 16:41:13

    Name                                       Base        Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: CODE                             C:000C0A    C:000C30       27h


Module: ..\..\..\..\..\..\..\..\eZ8c:PRINT_UPUTCH.C (Library: crtD.lib)
Version: 1:0 03/01/2006 16:41:13

    Name                                       Base        Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: CODE                             C:0009E9    C:000A05       1dh


Module: ..\..\..\..\..\..\..\eZ8c:CLOSEPORTB.C (Library: gpioD.lib) Version:
1:0 03/01/2006 16:42:19
```

```
    Name                                       Base        Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: CODE                             C:0009CE    C:0009E8       1bh


Module: ..\..\..\..\..\..\..\eZ8c:CLOSEPORTD.C (Library: gpioD.lib) Version:
1:0 03/01/2006 16:42:19

    Name                                       Base        Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: CODE                             C:000A51    C:000A6B       1bh


Module: ..\..\..\..\..\..\..\eZ8c:CLOSEUART0.C (Library: uartF91simD.lib)
Version: 1:0 03/01/2006 16:42:32

    Name                                       Base        Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: CODE                             C:00087A    C:0008B1       38h


Module: ..\..\..\..\..\..\..\eZ8c:CONTROLPORTB.C (Library: gpioD.lib) Version:
1:0 03/01/2006 16:42:19

    Name                                       Base        Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: CODE                             C:000BCF    C:000C09       3bh


Module: ..\..\..\..\..\..\..\eZ8c:CONTROLPORTD.C (Library: gpioD.lib) Version:
1:0 03/01/2006 16:42:20

    Name                                       Base        Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: CODE                             C:000CA2    C:000CDC       3bh


Module: ..\..\..\..\..\..\..\eZ8c:CONTROLUART0.C (Library: uartF91simD.lib)
Version: 1:0 03/01/2006 16:42:32

    Name                                       Base        Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: BSS                              D:B801C6    D:B801D9       14h
    Segment: CODE                             C:00128B    C:0015CA      340h
    Segment: DATA                             D:B80111    D:B80114        4h
```

```
Module: ..\..\..\..\..\..\..\eZ8c:FIFOADD.C (Library: uartF91simD.lib)
Version: 1:0 03/01/2006 16:42:33

    Name                                       Base        Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: CODE                             C:000C31    C:000CA1       71h


Module: ..\..\..\..\..\..\..\eZ8c:FIFOEMPTY.C (Library: uartF91simD.lib)
Version: 1:0 03/01/2006 16:42:33

    Name                                       Base        Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: CODE                             C:001773    C:00179A       28h


Module: ..\..\..\..\..\..\..\eZ8c:FIFOFULL.C (Library: uartF91simD.lib)
Version: 1:0 03/01/2006 16:42:33

    Name                                       Base        Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: CODE                             C:00179B    C:0017EB       51h


Module: ..\..\..\..\..\..\..\eZ8c:FIFOGET0.C (Library: uartF91simD.lib)
Version: 1:0 03/01/2006 16:42:33

    Name                                       Base        Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: CODE                             C:00183E    C:001913       d6h


Module: ..\..\..\..\..\..\..\eZ8c:FLUSHUART0.C (Library: uartF91simD.lib)
Version: 1:0 03/01/2006 16:42:34

    Name                                       Base        Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: CODE                             C:001623    C:001686       64h


Module: ..\..\..\..\..\..\..\eZ8c:GETCH.C (Library: uartF91simD.lib) Version:
1:0 03/01/2006 16:42:34

    Name                                       Base        Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: CODE                             C:001713    C:001772       60h
```

```
Module: ..\..\..\..\..\..\..\eZ8c:ISRUART0.C (Library: uartF91simD.lib)
Version: 1:0 03/01/2006 16:42:34

    Name                                      Base        Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: BSS                             D:B801A1    D:B801A1        1h
    Segment: CODE                            C:000A6C    C:000BCE      163h


Module: ..\..\..\..\..\..\..\eZ8c:KBHIT.C (Library: uartF91simD.lib) Version:
1:0 03/01/2006 16:42:34

    Name                                      Base        Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: CODE                            C:0015CB    C:001622       58h


Module: ..\..\..\..\..\..\..\eZ8c:OPENPORTB.C (Library: gpioD.lib) Version:
1:0 03/01/2006 16:42:20

    Name                                      Base        Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: CODE                            C:0007FC    C:00083A       3fh


Module: ..\..\..\..\..\..\..\eZ8c:OPENPORTD.C (Library: gpioD.lib) Version:
1:0 03/01/2006 16:42:20

    Name                                      Base        Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: CODE                            C:00083B    C:000879       3fh


Module: ..\..\..\..\..\..\..\eZ8c:OPENUART0.C (Library: uartF91simD.lib)
Version: 1:0 03/01/2006 16:42:35

    Name                                      Base        Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: CODE                            C:0007E6    C:0007FB       16h


Module: ..\..\..\..\..\..\..\eZ8c:PUTCH.C (Library: uartF91simD.lib) Version:
1:0 03/01/2006 16:42:35

    Name                                      Base        Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: CODE                            C:0017EC    C:00183D       52h
```

```
Module: ..\..\..\..\..\..\..\eZ8c:READUART0.C (Library: uartF91simD.lib)
Version: 1:0 03/01/2006 16:42:35

    Name                                       Base         Top       Size
    --------------------------------------- ----------- ----------- ---------
    Segment: CODE                             C:00197C    C:001A5B       e0h


Module: ..\..\..\..\..\..\..\eZ8c:SETMODEPORTB.C (Library: gpioD.lib) Version:
1:0 03/01/2006 16:42:21

    Name                                       Base         Top       Size
    --------------------------------------- ----------- ----------- ---------
    Segment: CODE                             C:000CDD    C:000ED5      1f9h


Module: ..\..\..\..\..\..\..\eZ8c:SETMODEPORTD.C (Library: gpioD.lib) Version:
1:0 03/01/2006 16:42:21

    Name                                       Base         Top       Size
    --------------------------------------- ----------- ----------- ---------
    Segment: CODE                             C:001092    C:00128A      1f9h


Module: ..\..\..\..\..\..\..\eZ8c:WRITEUART0.C (Library: uartF91simD.lib)
Version: 1:0 03/01/2006 16:42:37

    Name                                       Base         Top       Size
    --------------------------------------- ----------- ----------- ---------
    Segment: CODE                             C:001914    C:001952       3fh


Module: ..\..\src\boot\common\cstartup.asm (File: cstartup.obj) Version: 1:0
03/01/2006 16:41:09

    Name                                       Base         Top       Size
    --------------------------------------- ----------- ----------- ---------
    Segment: .STARTUP                         C:000202    C:00024F       4eh
    Segment: DATA                             D:B80100    D:B80102        3h


Module: ..\..\src\boot\common\vectors24.asm (File: vectors24.obj) Version: 1:0
03/01/2006 16:41:09

    Name                                       Base         Top       Size
    --------------------------------------- ----------- ----------- ---------
    Segment: .IVECTS                          D:B80000    D:B800FF      100h
    Segment: .RESET                              C:000000    C:00006A       6bh
    Segment: .STARTUP                            C:00006B    C:0000CD       63h


Module: ..\..\src\boot\eZ80F91\init_params_f91.asm (File: init_params_f91.obj)
Version: 1:0 03/01/2006 16:41:10

    Name                                          Base        Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: .STARTUP                            C:0000CE    C:000201      134h


Module: ..\..\src\rtl\common\case.asm (Library: chelpD.lib) Version: 1:0 03/
01/2006 16:41:57

    Name                                          Base        Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: CODE                                C:0016BF    C:0016EA       2ch


Module: ..\..\src\rtl\common\fpdumy.asm (File: fpdumy.obj) Version: 1:0 03/01/
2006 16:41:10

    Name                                          Base        Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: CODE                                C:0007DE    C:0007DE        1h


Module: ..\..\src\rtl\common\iand.asm (Library: chelpD.lib) Version: 1:0 03/
01/2006 16:41:57

    Name                                          Base        Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: CODE                                C:0016F8    C:001712       1bh


Module: ..\..\src\rtl\common\imulu.asm (Library: chelpD.lib) Version: 1:0 03/
01/2006 16:41:57

    Name                                          Base        Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: CODE                                C:0008DA    C:000915       3ch


Module: ..\..\src\rtl\common\indcall.asm (Library: chelpD.lib) Version: 1:0
03/01/2006 16:41:57

    Name                                          Base        Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: CODE                              C:001971    C:001972        2h


Module: ..\..\src\rtl\common\ineg.asm (Library: chelpD.lib) Version: 1:0 03/
01/2006 16:41:58

    Name                                        Base        Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: CODE                              C:001973    C:00197B        9h


Module: ..\..\src\rtl\common\ishrs.asm (Library: chelpD.lib) Version: 1:0 03/
01/2006 16:41:58

    Name                                        Base        Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: CODE                              C:0009B1    C:0009CD       1dh


Module: ..\..\src\rtl\common\itol.asm (Library: chelpD.lib) Version: 1:0 03/
01/2006 16:41:58

    Name                                        Base        Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: CODE                              C:0007DF    C:0007E5        7h


Module: ..\..\src\rtl\common\lcmpu.asm (Library: chelpD.lib) Version: 1:0 03/
01/2006 16:41:58

    Name                                        Base        Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: CODE                              C:0008B2    C:0008C7       16h


Module: ..\..\src\rtl\common\ldivs.asm (Library: chelpD.lib) Version: 1:0 03/
01/2006 16:41:58

    Name                                        Base        Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: CODE                              C:000983    C:0009B0       2eh


Module: ..\..\src\rtl\common\ldivu.asm (Library: chelpD.lib) Version: 1:0 03/
01/2006 16:41:58

    Name                                        Base        Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: CODE                                   C:000A1B    C:000A50       36h


Module: ..\..\src\rtl\common\lmulu.asm (Library: chelpD.lib) Version: 1:0 03/
01/2006 16:41:58

    Name                                             Base         Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: CODE                                   C:000916    C:000982       6dh


Module: ..\..\src\rtl\common\lneg.asm (Library: chelpD.lib) Version: 1:0 03/
01/2006 16:41:58

    Name                                             Base         Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: CODE                                   C:0016EB    C:0016F7        dh


Module: ..\..\src\rtl\common\stoiu.asm (Library: chelpD.lib) Version: 1:0 03/
01/2006 16:41:59

    Name                                             Base         Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: CODE                                   C:001687    C:00168C        6h


Module: .\BUTTONS.C (File: Buttons.obj) Version: 1:0 03/03/2006 11:49:41

    Name                                             Base         Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: CODE                                   C:000250    C:000329       dah


Module: .\LEDMATRIX.C (File: LedMatrix.obj) Version: 1:0 03/03/2006 11:49:42

    Name                                             Base         Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: BSS                                    D:B80115    D:B8011B        7h
    Segment: CODE                                   C:00032A    C:000586      25dh
    Segment: TEXT                                   C:001A5C    C:001DDB      380h


Module: .\LEDTIMER.C (File: LedTimer.obj) Version: 1:0 03/03/2006 11:49:43

    Name                                             Base         Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: BSS                                 D:B8011C    D:B80120        5h
    Segment: CODE                                C:000587    C:0006C4      13eh


Module: .\MAIN.C (File: main.obj) Version: 1:0 03/03/2006 11:49:43

    Name                                          Base        Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: CODE                                C:0006C5    C:00077E       bah
    Segment: DATA                                D:B80103    D:B8010A        8h
    Segment: STRSECT                             C:001DE0    C:001E64       85h
    Segment: TEXT                                C:001DDC    C:001DDF        4h


Module: .\strlen.asm (Library: crtD.lib) Version: 1:0 03/01/2006 16:41:19

    Name                                          Base        Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: CODE                                C:000A06    C:000A1A       15h


Module:
C:\PROGRA~1\ZiLOG\UNDERD~1\ZDSII_~2.0\samples\EZ80F9~1\src\zsldevinit.asm
(File: zsldevinit.obj) Version: 1:0 03/03/2006 11:49:43

    Name                                          Base        Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: BSS                                 D:B80121    D:B801A0       80h
    Segment: CODE                                C:00077F    C:0007DD       5fh
    Segment: DATA                                D:B8010B    D:B8010D        3h


EXTERNAL DEFINITIONS:
=====================

Symbol                              Address   Module          Segment
-------------------------------- ----------- --------------- -----------------
---------------
___print_buff                 D:B801A2 eZ8c:PRINT_GLOB BSS
___print_fmt                  D:B801AE eZ8c:PRINT_GLOB BSS
___print_leading_char           D:B801C2 eZ8c:PRINT_GLOB BSS
___print_len                  D:B801C1 eZ8c:PRINT_GLOB BSS
___print_out                  D:B801C3 eZ8c:PRINT_GLOB BSS
___print_putch                C:001953 eZ8c:PRINT_PUTC CODE
___print_sendstring            C:000ED6 eZ8c:PRINT_PUTS CODE
___print_sputch               C:000C0A eZ8c:PRINT_SPUT CODE
___print_uputch               C:0009E9 eZ8c:PRINT_UPUT CODE
___print_xputch               D:B8010E eZ8c:PRINT_GLOB DATA
```

```
__c_startup                  C:000202 cstartup      .STARTUP
__case                       C:0016BF case          CODE
__close_periphdevice           C:0007D1 zsldevinit    CODE
__copy_code_to_ram                  00000000 (User Defined)
__crtl                              00000001 (User Defined)
__CS0_BMC_INIT_PARAM                00000002 (User Defined)
__CS0_CTL_INIT_PARAM                000000A8 (User Defined)
__CS0_LBR_INIT_PARAM                00000010 (User Defined)
__CS0_UBR_INIT_PARAM                0000001F (User Defined)
__CS1_BMC_INIT_PARAM                00000002 (User Defined)
__CS1_CTL_INIT_PARAM                00000028 (User Defined)
__CS1_LBR_INIT_PARAM                00000000 (User Defined)
__CS1_UBR_INIT_PARAM                00000007 (User Defined)
__CS2_BMC_INIT_PARAM                00000002 (User Defined)
__CS2_CTL_INIT_PARAM                00000028 (User Defined)
__CS2_LBR_INIT_PARAM                00000080 (User Defined)
__CS2_UBR_INIT_PARAM                000000BF (User Defined)
__CS3_BMC_INIT_PARAM                00000002 (User Defined)
__CS3_CTL_INIT_PARAM                00000000 (User Defined)
__CS3_LBR_INIT_PARAM                00000000 (User Defined)
__CS3_UBR_INIT_PARAM                00000000 (User Defined)
__cstartup                   D:000001 cstartup      DATA
__default_mi_handler          C:00006F vectors24     .STARTUP
__default_nmi_handler         C:00006D vectors24     .STARTUP
__exit                      C:0001B6 init_params_f91 .STARTUP
__fadd                       C:0007DE fpdumy        CODE
__fcmp                       C:0007DE fpdumy        CODE
__fdiv                       C:0007DE fpdumy        CODE
__FLASH_ADDR_U_INIT_PARAM           00000000 (User Defined)
__FLASH_CTL_INIT_PARAM              00000060 (User Defined)
__fmul                       C:0007DE fpdumy        CODE
__fneg                       C:0007DE fpdumy        CODE
__fppack                     C:0007DE fpdumy        CODE
__fsub                       C:0007DE fpdumy        CODE
__ftol                       C:0007DE fpdumy        CODE
__heapbot                           00B801DA (User Defined)
__heaptop                           00BFFFFF (User Defined)
__iand                       C:0016F8 iand          CODE
__imuls                      C:0008DA imulu         CODE
__imulu                      C:0008DA imulu         CODE
__indcall                    C:001971 indcall       CODE
__ineg                       C:001973 ineg          CODE
__init                      C:0000CE init_params_f91 .STARTUP
__init_default_vectors         C:000072 vectors24     .STARTUP
__ishrs                      C:0009B1 ishrs         CODE
__itol                       C:0007DF itol          CODE
__lcmps                      C:0008B2 lcmpu         CODE
__lcmpu                      C:0008B2 lcmpu         CODE
```

```
__ldivs                      C:000983 ldivs         CODE
__ldivu                      C:000A1B ldivu         CODE
__ldvrmu                     C:000A2D ldivu         CODE
__len_bss                          000000C5 (User Defined)
__len_code                         0000180C (User Defined)
__len_data                         00000015 (User Defined)
__lmuls                      C:000916 lmulu         CODE
__lmulu                      C:000916 lmulu         CODE
__lneg                       C:0016EB lneg          CODE
__low_bss                          00B80115 (User Defined)
__low_code                         00000250 (User Defined)
__low_data                         00B80100 (User Defined)
__low_romcode                      00000000 (User Defined)
__low_romdata                      00001E65 (User Defined)
__ltof                       C:0007DE fpdumy        CODE
__nvectors                   C:00006B vectors24      .STARTUP
__open_periphdevice            C:00077F zsldevinit     CODE
__PLL_CTL0_INIT_PARAM              00000000 (User Defined)
__RAM_ADDR_U_INIT_PARAM            000000B7 (User Defined)
__RAM_CTL_INIT_PARAM               000000C0 (User Defined)
__set_vector                 C:0000A3 vectors24      .STARTUP
__sneg                       C:001973 ineg          CODE
__stack                            00C00000 (User Defined)
__stoiu                      C:001687 stoiu         CODE
__u_dtoe                     C:0007DE fpdumy        CODE
__u_dtof                     C:0007DE fpdumy        CODE
__u_dtog                     C:0007DE fpdumy        CODE
__u_flt_info                  C:0007DE fpdumy        CODE
__u_flt_rnd                   C:0007DE fpdumy        CODE
__ultof                      C:0007DE fpdumy        CODE
__vector_table                D:B80000 vectors24      .IVECTS
_abort                     C:0001B6 init_params_f91 .STARTUP
_abs                         C:00168D eZ8c:ABS       CODE
_acos                        C:0007DE fpdumy        CODE
_asin                        C:0007DE fpdumy        CODE
_atan                        C:0007DE fpdumy        CODE
_atan2                       C:0007DE fpdumy        CODE
_buttons_init                 C:0002D2 BUTTONS       CODE
_ceil                        C:0007DE fpdumy        CODE
_close_periphdevice            C:0007D1 zsldevinit     CODE
_close_PortB                 C:0009CE eZ8c:CLOSEPORTB CODE
_close_PortD                 C:000A51 eZ8c:CLOSEPORTD CODE
_close_UART0                 C:00087A eZ8c:CLOSEUART0 CODE
_control_PortB               C:000BCF eZ8c:CONTROLPOR CODE
_control_PortD               C:000CA2 eZ8c:CONTROLPOR CODE
_control_UART0               C:00128B eZ8c:CONTROLUAR CODE
_cosh                        C:0007DE fpdumy        CODE
_device_name                  D:B80103 MAIN          DATA
```

```
_errno                       D:B80100 cstartup      DATA
_exp                         C:0007DE fpdumy        CODE
_fabs                        C:0007DE fpdumy        CODE
_FifoAdd                     C:000C31 eZ8c:FIFOADD   CODE
_FifoEmpty                   C:001773 eZ8c:FIFOEMPTY  CODE
_FifoFull                    C:00179B eZ8c:FIFOFULL   CODE
_FifoGet0                    C:00183E eZ8c:FIFOGET0   CODE
_floor                       C:0007DE fpdumy        CODE
_flush_UART0                 C:001623 eZ8c:FLUSHUART0 CODE
_fmod                        C:0007DE fpdumy        CODE
_g_clock0                    D:B80111 eZ8c:CONTROLUAR DATA
_g_fifosize                  D:B8010B zsldevinit     DATA
_g_HWflowctl_UART0            D:B801D8 eZ8c:CONTROLUAR BSS
_g_mode_UART0                D:B801D9 eZ8c:CONTROLUAR BSS
_g_recverr0                  D:B801A1 eZ8c:ISRUART0   BSS
_g_RxBuffer_UART0             D:B80121 zsldevinit     BSS
_g_RxFIFO_UART0              D:B801CF eZ8c:CONTROLUAR BSS
_g_TxBuffer_UART0             D:B80161 zsldevinit     BSS
_g_TxFIFO_UART0              D:B801C6 eZ8c:CONTROLUAR BSS
_getch                       C:001713 eZ8c:GETCH     CODE
_getchar                     C:0008C8 eZ8c:GETCHAR   CODE
_init_default_vectors         C:000072 vectors24     .STARTUP
_init_sys_clk_pll            C:0001BC init_params_f91 .STARTUP
_InitSysClk                  C:0001EA init_params_f91 .STARTUP
_InitSysClkDone              C:0001F6 init_params_f91 .STARTUP
_isr_uart0                   C:000A6C eZ8c:ISRUART0   CODE
_kbhit                       C:001611 eZ8c:KBHIT     CODE
_ldexp                       C:0007DE fpdumy        CODE
_ledmatrix_clear              C:00034A LEDMATRIX      CODE
_ledmatrix_fill               C:000361 LEDMATRIX      CODE
_ledmatrix_flash              C:0004E4 LEDMATRIX      CODE
_ledmatrix_init               C:00032A LEDMATRIX      CODE
_ledmatrix_putc               C:000378 LEDMATRIX      CODE
_ledmatrix_puts               C:0003EC LEDMATRIX      CODE
_ledmatrix_spin               C:00045B LEDMATRIX      CODE
_ledmatrix_test               C:000539 LEDMATRIX      CODE
_ledtimer_init                C:0005FB LEDTIMER      CODE
_log                         C:0007DE fpdumy        CODE
_log10                       C:0007DE fpdumy        CODE
_main                        C:0006C5 MAIN          CODE
_matrix_char_map              C:001A5C LEDMATRIX      TEXT
_open_periphdevice            C:00077F zsldevinit     CODE
_open_PortB                  C:0007FC eZ8c:OPENPORTB  CODE
_open_PortD                  C:00083B eZ8c:OPENPORTD  CODE
_open_UART0                  C:0007E6 eZ8c:OPENUART0  CODE
_OSC_FREQ                           01312D00 (User Defined)
_OSC_FREQ_MULT                      00000001 (User Defined)
_OscFreq                    C:0001F7 init_params_f91 .STARTUP
```

```
_p_user_input                D:B80118 LEDMATRIX      BSS
_pb0_isr                     C:000250 BUTTONS        CODE
_pb1_isr                     C:00027A BUTTONS        CODE
_pb2_isr                     C:0002A6 BUTTONS        CODE
_pcolumn                     D:B80115 LEDMATRIX      BSS
_pow                         C:0007DE fpdumy         CODE
_putch                       C:0017EC eZ8c:PUTCH     CODE
_read_UART0                  C:00197C eZ8c:READUART0  CODE
_reset                       C:000000 vectors24      .RESET
_set_vector                  C:0000A3 vectors24      .STARTUP
_setmode_PortB               C:000CDD eZ8c:SETMODEPOR CODE
_setmode_PortD               C:001092 eZ8c:SETMODEPOR CODE
_sinh                        C:0007DE fpdumy         CODE
_sqrt                        C:0007DE fpdumy         CODE
_strlen                      C:000A06 strlen         CODE
_SYS_CLK_FREQ                      01312D00 (User Defined)
_SYS_CLK_SRC                       00000000 (User Defined)
_SysClkFreq                  C:0001FD init_params_f91 .STARTUP
_SysClkSrc                   C:000201 init_params_f91 .STARTUP
_tan                         C:0007DE fpdumy         CODE
_tanh                        C:0007DE fpdumy         CODE
_timer                       D:B8011C LEDTIMER       BSS
_tmr2_isr                    C:000587 LEDTIMER       CODE
_user_input                  D:B8011B LEDMATRIX      BSS
_version                     C:001DDC MAIN           TEXT
_wait                        C:00066B LEDTIMER       CODE
_write_UART0                 C:001914 eZ8c:WRITEUART0 CODE
_zsl_g_clock_xdefine               02FAF080 (User Defined)

182 external symbol(s).


END OF LINK MAP:
================
0 Errors
0 Warnings
```

TROUBLESHOOTING THE LINKER

Review these questions to learn more about common situations you might encounter when using the linker:

- "How do I speed up the linker?" on page 284
- "How do I generate debug information without generating code?" on page 284
- "How can I debug code already programmed in ROM?" on page 284
- "How much memory is my program using?" on page 286

- "How do I determine the size of my actual hex code?" on page 286

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 check box in the Project Settings dialog box (Output page). See "Generate Map File" on page 93.
- 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:

```
COPY       myseg   NULL
CHANGE      myseg = NULL
```

How can I debug code already programmed in ROM?

This solution has two parts:

- "Part 1: Create and Program the Hex File with Debug Information" on page 284.
- "Part 2: Create a Build Configuration that Mimics the Original for Debugging" on page 285

Part 1 produces code that can be debugged and places it on the target, and Part 2 is necessary to get the IDE in sync with the code on the target. Both parts are necessary to make this solution work.

NOTE: Part 2 essentially produces a build configuration within the original project that allows the IDE to connect to the target for debugging. The new build configuration contains virtually the same project settings, except the executable format, as the project you used for creating your hex file.

Part 1: Create and Program the Hex File with Debug Information

1. From the File menu, select **Open Project** to open the project that was used to create the original hex file with ZDS II.
2. Select the build configuration used to create the original hex file from the Select Build Configuration drop-down list box in the Build toolbar.
3. Select **Settings** from the Project menu.

The Project Settings dialog box is displayed.

4. Select the General page.
5. Select the Generate Debug Information check box.
6. Click **OK** to close the Project Settings dialog box.
7. From the Build menu, select **Rebuild All** to build the project.
8. Program the newly generated hex file in the target.

Part 2: Create a Build Configuration that Mimics the Original for Debugging

1. Select **Manage Configurations** from the Build menu.

   The Manage Configurations dialog box is displayed.
2. Click **Add**.

   The Add Project Configuration dialog box is displayed.
3. In the Configuration Name field, enter a new configuration name (for example, Debug – Flash).
4. Verify that the target is eZ80Acclaim!.
5. In the Copy Settings From drop-down list box, select the build configuration used to create the original hex file.
6. Click **OK** to close the Add Project Configuration dialog box.
7. Click **Close** to close the Manage Configurations dialog box.
8. Select the new build configuration (for example, Debug – Flash) from the Select Build Configuration drop-down list box in the Build toolbar.
9. Select **Settings** from the Project menu.

   The Project Settings dialog box is displayed.
10. In the General page, make sure that the Generate Debug Information check box is selected.
11. In the Output page, select **IEEE 695** in the Executable Formats area.
12. In the Debugger page, select the appropriate target in the Target area and click **Setup**.

    The Configure Target dialog box is displayed.
13. Verify that the initialization parameters mimic the hardware parameters used by the original project.
14. Enter `0` in the Program Counter (hex) field.
15. Click **OK** to close the Configure Target dialog box.
16. Click **OK** to close the Project Settings dialog box.

17. From the Build menu, select **Rebuild All** to build the project.
18. From the Debug menu, select **Connect to Target** to connect to the target.

Caution: Do not establish a target connection using the Reset or Go commands/buttons and do not invoke the Download Code command/button. Doing so might corrupt the ROM/Flash device. The Reset and Go commands/buttons can be used in all other cases.

19. Continue debugging as usual.

NOTE: For the solution given here, it is assumed that the source and project files are available to recreate the hex image with ZDS II and that the target can be reprogrammed. If the source and project files are *not* available to recreate the hex image with ZDS II or the target *cannot* be reprogrammed, a new project can be created for debugging at the disassembly level. In that case, create a new project for the target that contains a blank source file.

How much memory is my program using?

Unless the Generate Map File check box is deselected 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 “Generate Map File” on page 93 and “MAP” on page 253.

How do I determine the size of my actual hex code?

Refer to the map file. Unless the Generate Map File check box is deselected in the Project Settings dialog box (“Generate Map File” on page 93), 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 on 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 startup code might assume that it is contiguous.

Using the Debugger

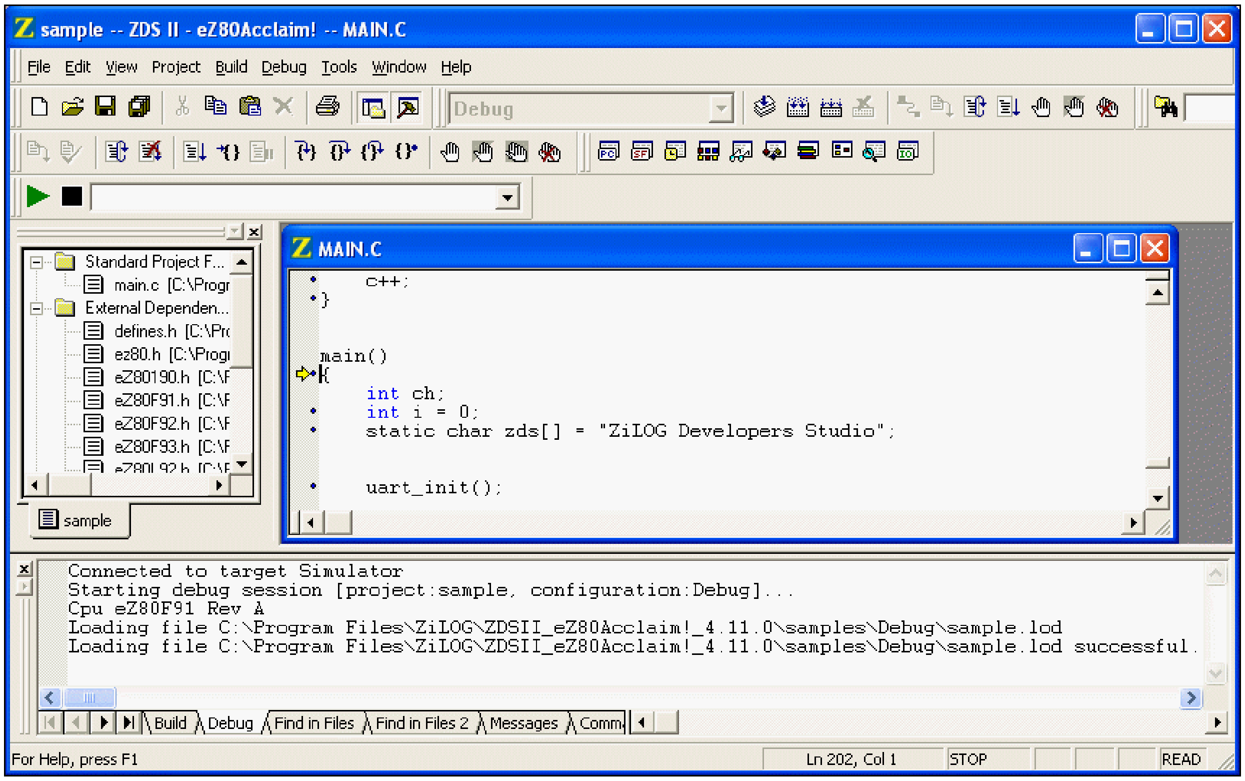
The source-level debugger is a program that allows you to find problems in your code at the C or assembly level. You can also write batch files to automate debugger tasks (see "Using the Command Processor" on page 387). The following topics are covered in this section:

- "Status Bar" on page 291
- "Code Line Indicators" on page 292
- "Debug Windows" on page 292
- "Using Breakpoints" on page 306
- "Debug Tools" on page 310
- "Targets" on page 311

From the Debug menu, select **Reset** (or any other execution command) to enter Debug mode.

You are now in Debug mode as shown in the Debug Output window. (For a description of this window, see "Debug Output Window" on page 35.)

The Debug toolbar and Debug Windows toolbar are displayed as shown in the following figure. The Debug toolbar is described in "Debug Toolbar" on page 23; the Debug Windows toolbar is described in "Debug Windows Toolbar" on page 27.



```
    c++;
}

main()
{
    int ch;
    int i = 0;
    static char zds[] = "ZiLOG Developers Studio";

    uart_init();
```

```
Connected to target Simulator
Starting debug session [project:sample, configuration:Debug]...
Cpu eZ80F91 Rev A
Loading file C:\Program Files\ZiLOG\ZDSII_eZ80Acclaim!_4.11.0\samples\Debug\sample.lod
Loading file C:\Program Files\ZiLOG\ZDSII_eZ80Acclaim!_4.11.0\samples\Debug\sample.lod successful.
```

Figure 93. Debug and Debug Window Toolbars

NOTE: Project code cannot be rebuilt while in Debug mode. The Development Environment 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, the Development Environment 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. The STOP mode indicates that your program is not executing. The STEP mode indicates that a Step operation (using the Step Into, Step Over, or Step Out command) is in progress. The RUN mode indicates that the program is executing after a Go command has been issued. In RUN mode, the following debug operations are available: Reset, Stop Debugging, Break, and enabling/disabling a breakpoint. Note that enabling/disabling a breakpoint temporarily stops program execution; program execution resumes after the breakpoint is enabled or disabled. View/read memory, Step Into, Step Over, Step Out, and Go are disabled in RUN mode.

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 have 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 a 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.

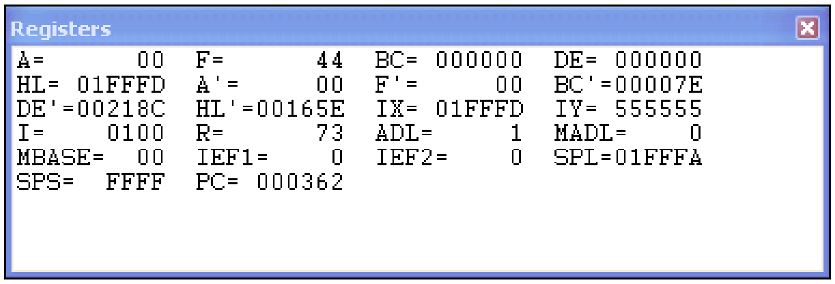
DEBUG WINDOWS

The Debug Windows toolbar (described in “Debug Windows Toolbar” on page 27) allows you to display the following Debug windows:

- “Registers Window” on page 292
- “Special Function Registers Window” on page 293
- “Clock Window” on page 294
- “Memory Window” on page 295
- “Watch Window” on page 300
- “Locals Window” on page 302
- “Call Stack Window” on page 303
- “Symbols Window” on page 304
- “Disassembly Window” on page 305
- “Simulated UART Output Window” on page 306

Registers Window

Click the Registers Window button to show or hide the Registers window. The Registers window displays all the registers in the standard eZ80Acclaim! architecture.



```
A=       00  F=       44  BC= 000000  DE= 000000
HL= 01FFFD  A'=       00  F'=       00  BC'=00007E
DE'=00218C  HL'=00165E  IX= 01FFFD  IY= 555555
I=     0100  R=       73  ADL=       1  MADL=     0
MBASE=   00  IEF1=     0  IEF2=     0  SPL=01FFFA
SPS=   FFFF  PC= 000362
```

Figure 94. Registers Window

To change register values, do the following:

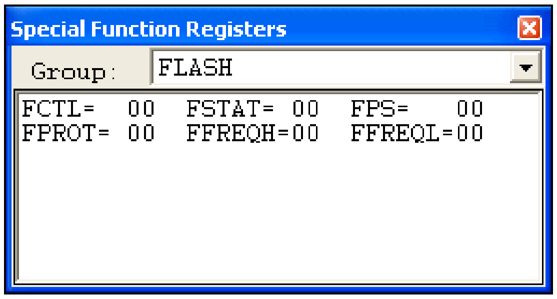
1. In the Registers window, highlight the value you want to change.
2. Type the new value and press the Enter key.

   The changed value is displayed in red.

Special Function Registers Window

Click the Special Function Registers Window button to open up to 20 Special Function Registers windows. Each Special Function Registers window displays the special function registers in the standard eZ80Acclaim! 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.



Group: FLASH

```
FCTL=   00  FSTAT= 00  FPS=    00
FPROT=  00  FFREQH=00  FFREQL=00
```

Figure 95. Special Function Registers Window

NOTE: 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 change special function register values, do the following:

1. In the Special Function Registers window, highlight the value you want to change.
2. Type the new value and press the Enter key.

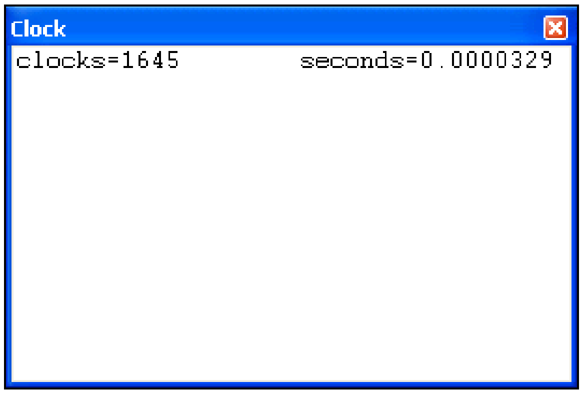
   The changed value is displayed in red.

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 last reset. You can reset the contents of the Clock window at any time by selecting the number of cycles (`1645` in the following figure), type `0`, and press 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=1645          seconds=0.0000329
```

Figure 96. Clock Window

Memory Window

Click the Memory Window button to open up to ten Memory windows.

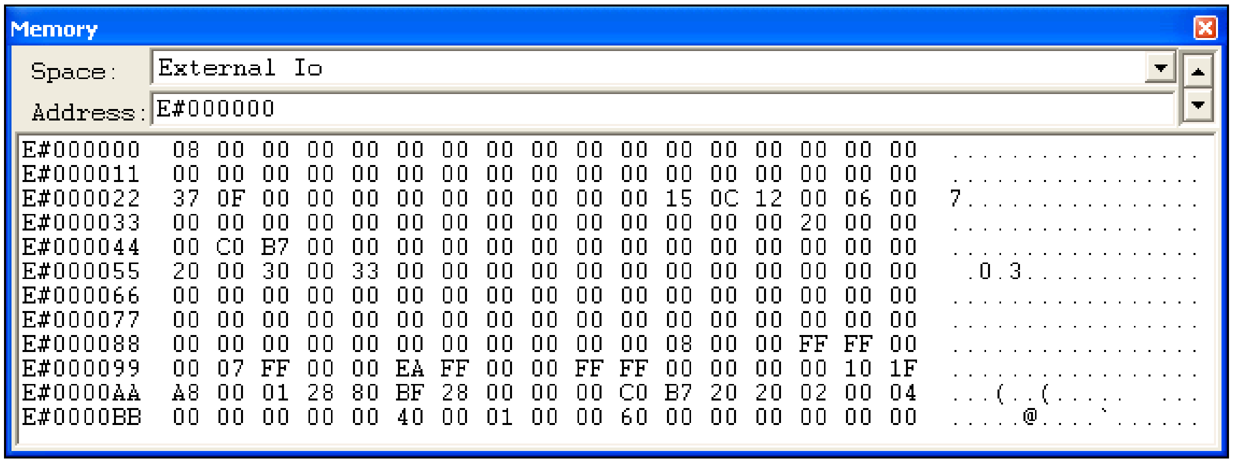

```
Space:   External Io
Address: E#000000

E#000000  08 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .................
E#000011  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .................
E#000022  37 0F 00 00 00 00 00 00 00 00 00 15 0C 12 00 06 00  7................
E#000033  00 00 00 00 00 00 00 00 00 00 00 00 00 00 20 00 00  .............. ..
E#000044  00 C0 B7 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .................
E#000055  20 00 30 00 33 00 00 00 00 00 00 00 00 00 00 00 00   .0.3............
E#000066  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .................
E#000077  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .................
E#000088  00 00 00 00 00 00 00 00 00 00 00 08 00 00 FF FF 00  .................
E#000099  00 07 FF 00 00 EA FF 00 00 FF FF 00 00 00 00 10 1F  .................
E#0000AA  A8 00 01 28 80 BF 28 00 00 00 C0 B7 20 20 02 00 04  ...(..(.....  ...
E#0000BB  00 00 00 00 00 40 00 01 00 00 60 00 00 00 00 00 00  .....@....`......
```

Figure 97. 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 E# to denote the External Io address space, F# to denote the Flash Info address space, I# to denote the Internal Io address space, or M# to denote the Memory address space.

Use the Memory window to do the following:

- "Change Values" on page 295
- "View the Values for Other Memory Spaces" on page 296
- "View or Search for an Address" on page 296
- "Fill Memory" on page 297
- "Save Memory to a File" on page 298
- "Load a File into Memory" on page 299

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, do the following:

1. In the window, highlight the value you want to change.

The values begin in the second column after the Address column.

2. Type 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.

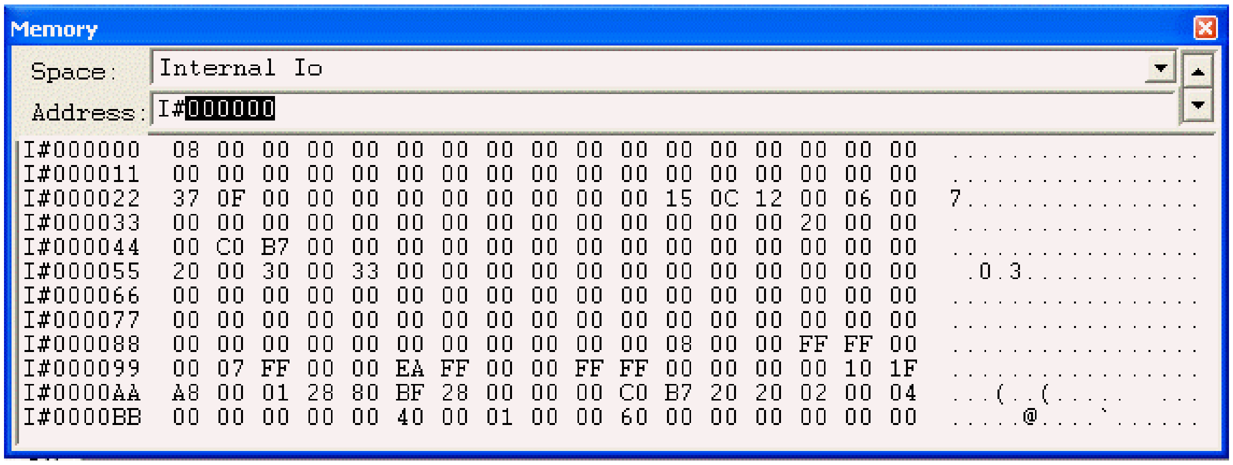
  For example, type I for the Internal Io memory space.

- 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, do the following:

1. In the Memory window, highlight the address in the Address field.



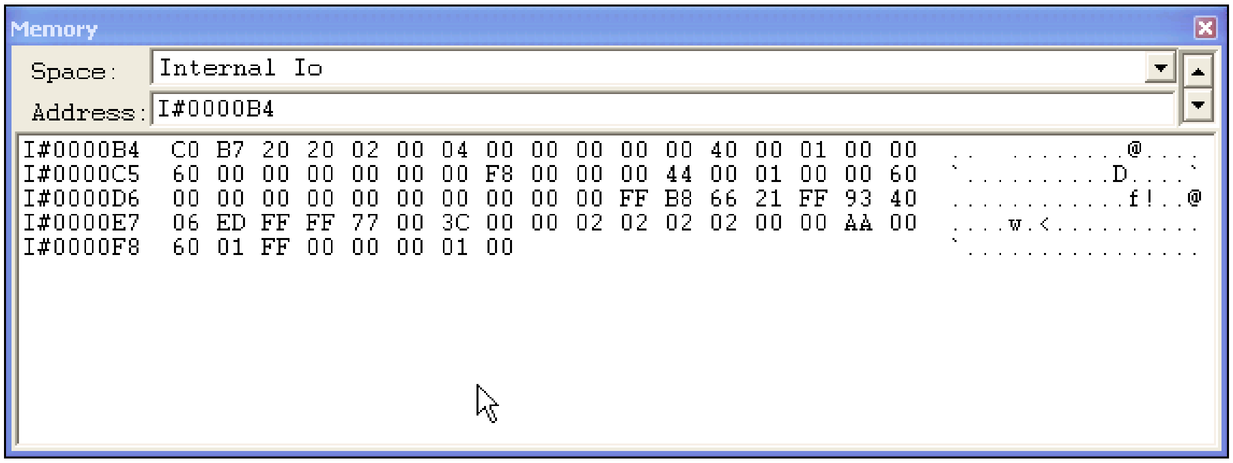
```
Space:    Internal Io
Address:  I#000000

I#000000  08 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   .................
I#000011  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   .................
I#000022  37 0F 00 00 00 00 00 00 00 00 00 15 0C 12 00 06 00   7................
I#000033  00 00 00 00 00 00 00 00 00 00 00 00 00 00 20 00 00   ...............  ..
I#000044  00 C0 B7 00 00 00 00 00 00 00 00 00 00 00 00 00 00   .................
I#000055  20 00 30 00 33 00 00 00 00 00 00 00 00 00 00 00 00    .0.3............
I#000066  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   .................
I#000077  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   .................
I#000088  00 00 00 00 00 00 00 00 00 00 00 08 00 00 FF FF 00   .................
I#000099  00 07 FF 00 00 EA FF 00 00 FF FF 00 00 00 00 10 1F   .................
I#0000AA  A8 00 01 28 80 BF 28 00 00 00 C0 B7 20 20 02 00 04   ...(..(.....   ...
I#0000BB  00 00 00 00 00 40 00 01 00 00 60 00 00 00 00 00 00   .....@....`......
```

Figure 98. Memory Window—Starting Address

2. Type the address you want to find and press the Enter key.

   For example, find `0000B4`.

   The system moves the selected address to the top of the Memory window, as shown in the following figure.



Space: Internal Io
Address: I#0000B4

```
I#0000B4  C0 B7 20 20 02 00 04 00 00 00 00 00 40 00 01 00 00   ..  ........@....
I#0000C5  60 00 00 00 00 00 00 F8 00 00 00 44 00 01 00 00 60   `..........D....`
I#0000D6  00 00 00 00 00 00 00 00 00 00 FF B8 66 21 FF 93 40   ............f!..@
I#0000E7  06 ED FF FF 77 00 3C 00 00 02 02 02 02 00 00 AA 00   ....w.<..........
I#0000F8  60 01 FF 00 00 00 01 00                              `................
```

Figure 99. Memory Window—Requested Address

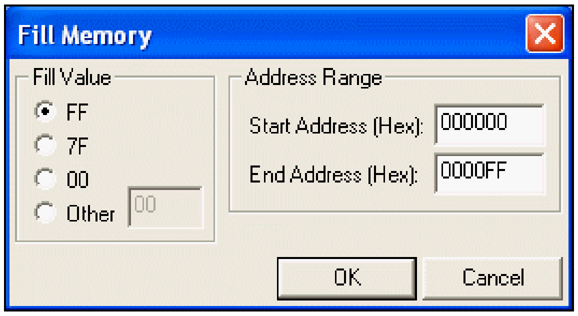
Fill Memory

Use this procedure to write a common value in all 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, do the following:

1. Select the memory space in the Space drop-down list.
2. Right-click in the list box to display the context menu.
3. Select **Fill Memory**.

   The Fill Memory dialog box is displayed.



Fill Value: [✓ FF] 7F 00 Other 00

Address Range
Start Address (Hex): 000000
End Address (Hex): 0000FF

Figure 100. 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, type the fill characters in the Other field.

5. Type the start address in hexadecimal format in the Start Address (Hex) field and type 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

Use this procedure to save memory specified by an address range to a binary, hexadecimal, or text file.

Perform the following steps to save memory to a file:

1. Select the memory space in the Space drop-down list.
2. Right-click in the list box to display the context menu.
3. Select **Save to File**.

   The Save to File dialog box is displayed.

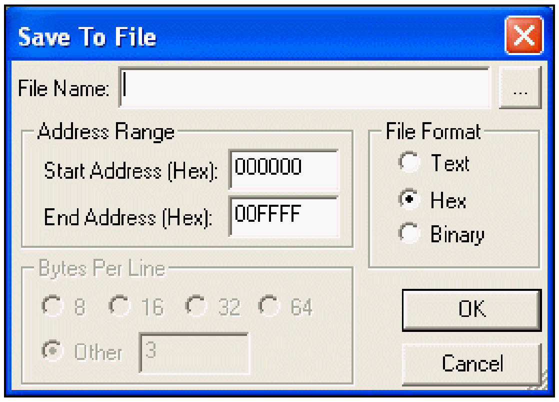


Figure 101. Save to File Dialog Box

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. Type the start address in hexadecimal format in the Start Address (Hex) field and type the end address in hexadecimal format in the End Address (Hex) field.

   This specifies the address range of memory to save to the specified file.

6. Select how many bytes there are in each line or enter a number in the Other field.
7. Select whether to save the file as text, hex (hexadecimal), or binary.

8. Click **OK** to save the memory to the specified file.

Load a File into Memory

Use this procedure to load or to initialize memory from an existing binary, hexadecimal, or text file.

Perform the following steps to load a file into memory:

1. Select the memory space in the Space drop-down list.
2. Right-click in the list box to display the context menu.
3. Select **Load from File**.

   The Load from File dialog box is displayed.

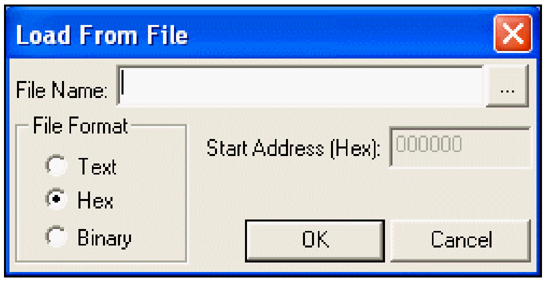
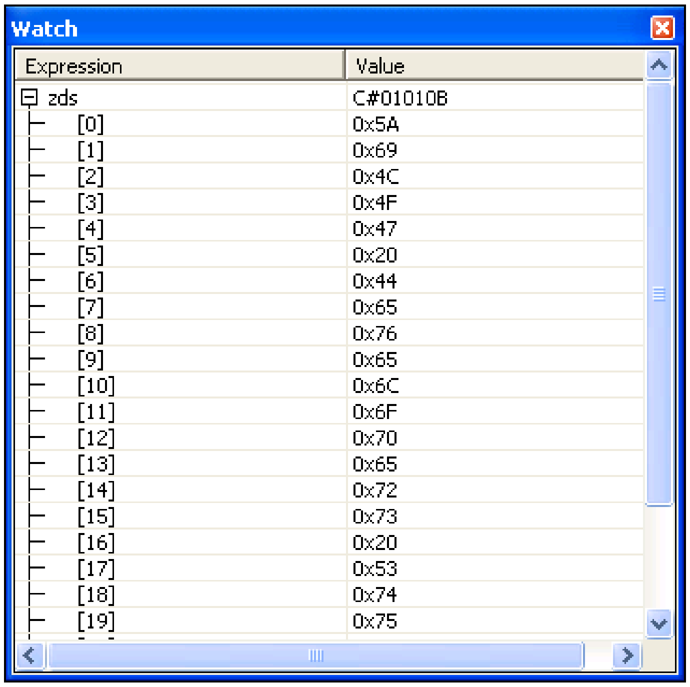


Figure 102. Load from File Dialog Box

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.

Watch Window

Click the Watch Window button to show or hide the Watch window.



Expression	Value
zds	C#01010B
[0]	0x5A
[1]	0x69
[2]	0x4C
[3]	0x4F
[4]	0x47
[5]	0x20
[6]	0x44
[7]	0x65
[8]	0x76
[9]	0x65
[10]	0x6C
[11]	0x6F
[12]	0x70
[13]	0x65
[14]	0x72
[15]	0x73
[16]	0x20
[17]	0x53
[18]	0x74
[19]	0x75

Figure 103. Watch Window

The Watch window displays all 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.

Use the Watch window to do the following:

- “Add New Variables” on page 301
- “Change Values” on page 301
- “Remove an Expression” on page 301
- “View a Hexadecimal Value” on page 301
- “View a Decimal Value” on page 301
- “View an ASCII Value” on page 302

- "View a NULL-Terminated ASCII (ASCIZ) String" on page 302

Add New Variables

To add new variables in the Watch window, use one of the following procedures:

- Click once on `<new watch>` in the Expression column, type the expression, and press the Enter key.
- Select the variable in the source file, drag, and drop it into the Watch window.

Change Values

To change values in the Watch window, do the following:

1. In the window, highlight the value you want to change.
2. Type 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, do the following:

1. In the Expression column, click once on 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, do the following:

1. Type `hex` *expression* in the Expression column

   For example, type `hex tens`.

NOTE: You can also type just the expression (for example, type `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.

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, do the following:

1. Type `dec` *expression* in the Expression column

   For example, type `dec huns`.

2. Press the Enter key.

The decimal value displays in the Value column.

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, do the following:

1. Type `ascii` *expression* in the Expression column.

   For example, type `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, do the following:

1. Type `asciz` *expression* in the Expression column

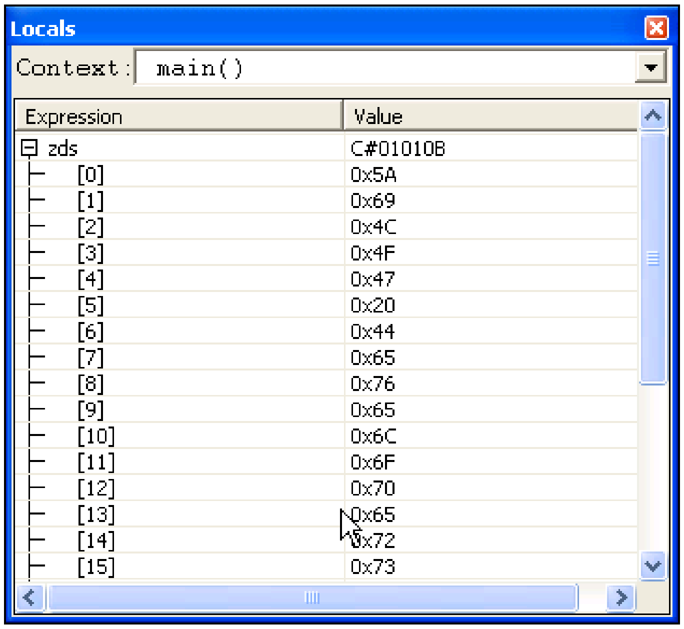
   For example, type `asciz ones`.
2. Press the Enter key.

   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 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.



Locals

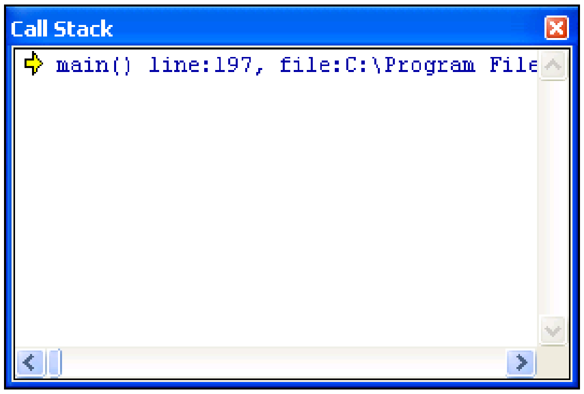
Context: main()

Expression	Value
zds	C#01010B
[0]	0x5A
[1]	0x69
[2]	0x4C
[3]	0x4F
[4]	0x47
[5]	0x20
[6]	0x44
[7]	0x65
[8]	0x76
[9]	0x65
[10]	0x6C
[11]	0x6F
[12]	0x70
[13]	0x65
[14]	0x72
[15]	0x73

Figure 104. Locals Window

Call Stack Window

Click the Call Stack Window button to show or hide the Call Stack window. If you want to copy text from the Call Stack Window, select the text and then select **Copy** from the context menu.



Call Stack

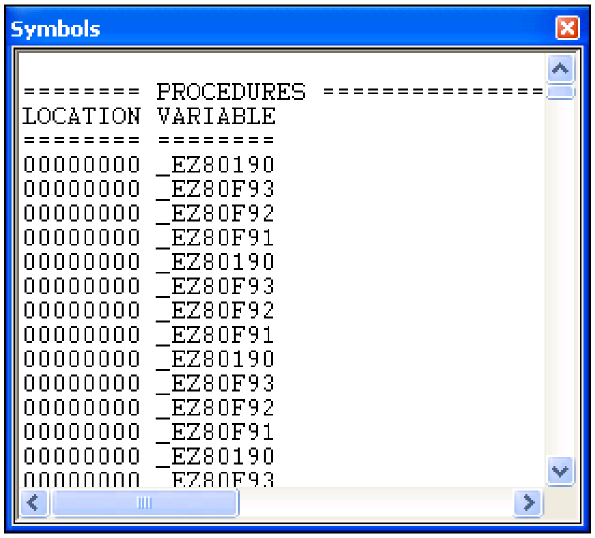
```
main() line:197, file:C:\Program File
```

Figure 105. 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.



```
======== PROCEDURES ===============
LOCATION VARIABLE
======== ========
00000000 _EZ80190
00000000 _EZ80F93
00000000 _EZ80F92
00000000 _EZ80F91
00000000 _EZ80190
00000000 _EZ80F93
00000000 _EZ80F92
00000000 _EZ80F91
00000000 _EZ80190
00000000 _EZ80F93
00000000 _EZ80F92
00000000 _EZ80F91
00000000 _EZ80190
00000000 _EZ80F93
```

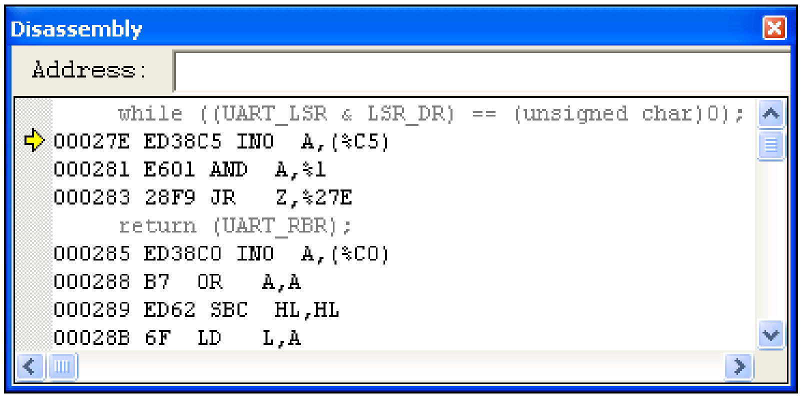
Figure 106. Symbols Window

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.



```
      while ((UART_LSR & LSR_DR) == (unsigned char)0);
00027E ED38C5 IN0  A,(%C5)
000281 E601 AND  A,%1
000283 28F9 JR   Z,%27E
      return (UART_RBR);
000285 ED38C0 IN0  A,(%C0)
000288 B7  OR   A,A
000289 ED62 SBC  HL,HL
00028B 6F  LD   L,A
```

Figure 107. 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.

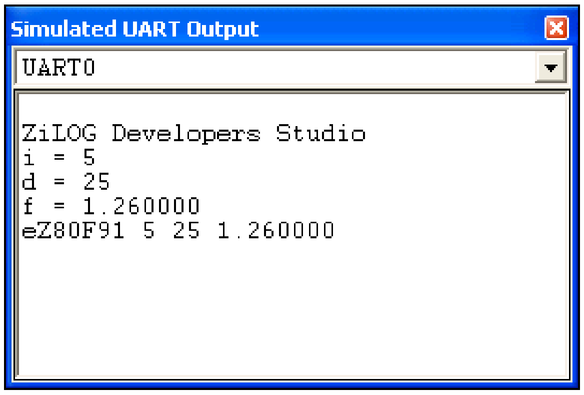
When you right-click in the Disassembly window, the context menu allows you to do the following:

- Copy text
- Go to the source code
- Insert, edit, enable, disable, or remove breakpoints

  For more information on breakpoints, see “Using Breakpoints” on page 306.
- 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

Simulated UART Output Window

Click the Simulated UART Output Window button to show or hide the Simulated UART Output window.



```
ZiLOG Developers Studio
i = 5
d = 25
f = 1.260000
eZ80F91 5 25 1.260000
```

Figure 108. 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

The following sections describeshow to work with breakpoints while you are debugging:

- "Inserting Breakpoints" on page 307
- "Viewing Breakpoints" on page 307
- "Moving to a Breakpoint" on page 308
- "Enabling Breakpoints" on page 308
- "Disabling Breakpoints" on page 309
- "Removing Breakpoints" on page 309

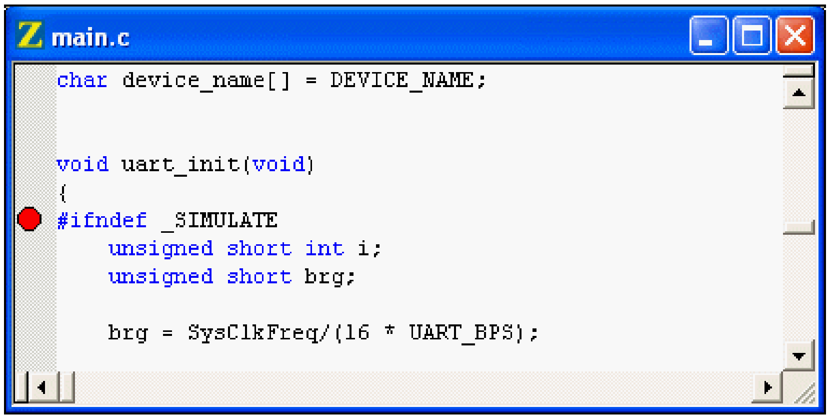
Inserting Breakpoints

There are three ways to place a breakpoint in your file:

- Click on the line of code where 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).

  Click the Insert/Remove Breakpoint button ( ) on the Build or Debug toolbar.
- Click on the line where 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 where 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.



```
char device_name[] = DEVICE_NAME;


void uart_init(void)
{
#ifndef _SIMULATE
    unsigned short int i;
    unsigned short brg;

    brg = SysClkFreq/(16 * UART_BPS);
```

Figure 109. 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.
- 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.

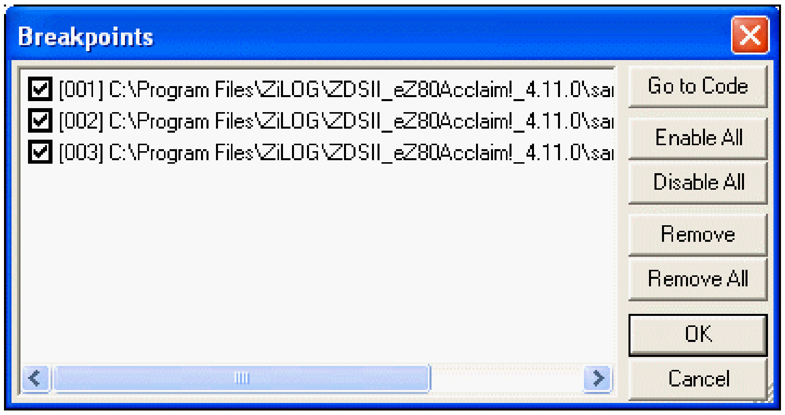

Breakpoints
[001] C:\Program Files\ZiLOG\ZDSII_eZ80Acclaim!_4.11.0\sa
[002] C:\Program Files\ZiLOG\ZDSII_eZ80Acclaim!_4.11.0\sa
[003] C:\Program Files\ZiLOG\ZDSII_eZ80Acclaim!_4.11.0\sa
Go to Code
Enable All
Disable All
Remove
Remove All
OK
Cancel

Figure 110. Viewing Breakpoints

Moving to a Breakpoint

To quickly move the cursor to a breakpoint you have previously set in your project, do the following:

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 where the breakpoint is set.

Enabling Breakpoints

To make all breakpoints in a project active, do the following:

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 on 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** and then click **OK**. All breakpoints are removed from the Breakpoints dialog box and 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 on the red octagon to remove the breakpoint.
- Select **Manage Breakpoints** from the Edit menu to display the Breakpoints dialog box. Click **Remove** and 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.

DEBUG TOOLS

Debug tools interface with the debugger to execute programs on physical or simulated target hardware. The following sections describe the available debug tools:

- "Cycle-Accurate Instruction Set Simulator" on page 310
- "Non-Simulator Debug Tools" on page 310

Refer to "Project Settings—Debugger Page" on page 95 for information on selecting and configuring debug tools. Debug tool configuration must be done before entering Debug mode.

Cycle-Accurate Instruction Set Simulator

The eZ80Acclaim! Instruction Set Simulator interfaces with the debugger to simulate the execution of programs without using an emulator or target hardware. The Instruction Set Simulator uses the commands of the debugger to perform tasks such as simulating timers and interrupts.

The Instruction Set Simulator also supports the programmable reload timer peripheral simulation for eZ80Acclaim! devices. You can operate the timers by using the eZ80Acclaim! I/O instructions to write into the respective control registers. The peripheral registers can be viewed in the I/O space of the Watch window (see "Watch Window" on page 300). Currently, viewing the registers in the Special Function Registers window is not supported. Refer to the individual device product specification for a detailed description on timers.

Non-Simulator Debug Tools

Non-Simulator debug tools interface with the debugger to execute programs on physical target hardware. These debug tools manage the communication between the host and the target hardware.

ZDS for the eZ80Acclaim! supports the following non-Simulator debug tools:

- USB Smart Cable
- Ethernet Smart Cable
- Serial Smart Cable
- ZPAK II

Refer to "Project Settings—Debugger Page" on page 95 for a description of the Debugger page of the Project Settings dialog box. See "Debug Tool" on page 102 for more details about individual debug tools configuration options.

TARGETS

A target is a logical representation of a physical hardware. Targets must be configured properly to enable the debugger to correctly execute a program on the physical hardware. the following sections describe the available targets:

- “RAM-Based Targets” on page 311
- “ROM/Flash-Based Targets” on page 311

Refer to “Project Settings—Debugger Page” on page 95 for more details about selecting, creating, and configuring targets. Target configuration must be done before entering Debug mode.

RAM-Based Targets

RAM-based debugging is typically used for development purposes. Debugging in RAM allows you to peek and poke memory and use software breakpoints.

ROM/Flash-Based Targets

ROM/Flash-based targets are used with production hardware or hardware configurations that do not have RAM.

NOTE: You cannot poke (write) during debugging; peek (view) might work depending on the Flash configurations. This limitation is not related to the eZ80F91 registers.

ZiLOG Standard Library Notes and Tips

Review the following questions to learn more about ZSL:

WHAT IS ZSL?

The ZiLOG Standard Library (ZSL) is a set of library files that provides an interface between the user application and the on-chip peripherals of the ZDS II microprocessors/ controllers.

WHICH ON-CHIP PERIPHERALS ARE SUPPORTED?

Version 1.0 of ZSL supports UARTs and GPIO peripherals.

WHERE CAN I FIND THE HEADER FILES RELATED TO ZILOG STANDARD LIBRARIES?

The header files related to ZiLOG Standard Libraries can be found under the following directory:

`ZILOGINSTALL\ZDSII_product_version\include\zilog`

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 `Z8Encore!`, `eZ80Acclaim!`, `Crimzon`, or `Z8GP`.
- *version* is the ZDS II version number. For example, *version* might be `4.11.0` or `5.0.0`.

WHAT IS THE ZSLDEVINIT.ASM FILE?

`zsldevinit.asm` is a device initialization file. It contains routines to initialize the devices you have selected in the ZSL page of the Project Settings dialog box.

WHAT INITIALIZATIONS ARE PERFORMED IN THE ZSLDEVINIT.ASM FILE?

The `open_periphdevice()` routine in `zsldevinit.asm` initializes the GPIO ports and UART devices. The functions in the file also initialize other dependent parameters like the clock speeds and UART FIFO sizes.

WHAT CALLS THE OPEN_PERIPHDEVICE() FUNCTION?

If the standard startup files are used, the `open_periphdevice()` function is called by the startup routine just before calling the main function.

WHEN I USE ZILOG STANDARD LIBRARIES IN MY APPLICATION AND BUILD FROM THE COMMAND LINE, WHY DO I SEE UNRESOLVED ERRORS?

Include `zsldevinit.asm` in your project.

The `open_periphdevice()` function has some external definitions like transmit and receive FIFO size required by the ZSL UART library. If you do not want to include this file, copy the logic that initializes the FIFO from the `zsldevinit.asm` file and include it in one of your project files, preferably in the boot-up module.

I DO NOT USE THE STANDARD BOOT-UP MODULE, BUT I HAVE MANUALLY INCLUDED ZILOG STANDARD LIBRARIES. WHEN I LINK MY CODE WITH THE LIBRARY, WHY DO I GET AN UNRESOLVED SYMBOLS ERROR?

Include `zsldevinit.asm` in your project.

The `open_periiphdevice()` function has some external definitions like transmit and receive FIFO size required by the ZSL UART library. If you do not want to include this file, copy the logic that initializes the FIFO from the `zsldevinit.asm` file and include it in one of your project files, preferably in the boot-up module.

WHERE CAN I GET THE ZSL SOURCE FILES?

The source files for ZSL can be found under the following directory:

ZILOGINSTALL`\ZDSII_`*product_version*`\src`

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 `Z8Encore!`, `eZ80Acclaim!`, `Crimzon`, or `Z8GP`.
- *version* is the ZDS II version number. For example, *version* might be `4.11.0` or `5.0.0`.

I NEED TO CHANGE THE ZSL SOURCE CODE. HOW CAN I GENERATE A NEW LIBRARY WITH THESE CHANGES INCLUDED?

A new library can be generated either by building the project under ZDS II by using the appropriate project file in the source directory or by running the batch files provided in the source directories. Refer to the *ZiLOG Standard Library API Reference Manual* (RM0037) for more details.

HOW CAN I USE STANDARD I/O CALLS LIKE PRINTF() AND GETCH()?

The standard I/O calls—such as `printf()`, `getch()`, and `putch()`—are routed to UART0 by default. You can route them to UART1 by setting the UART1 as the default device.

To do so, open the `uart.h` file and replace the DEFAULT_UART0 macro with DEFAULT_UART1 and rebuild the library. The `uart.h` file is in the following directory:

ZILOGINSTALL`\ZDSII_`*product_version*`\include\zilog`

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 `Z8Encore!`, `ZNEO`, `eZ80Acclaim!`, `Crimzon`, or `Z8GP`.
- *version* is the ZDS II version number. For example, *version* might be `4.11.0` or `5.0.0`.

Refer to the *ZiLOG Standard Library API Reference Manual* (RM0037) for more details.

WHAT IS THE DIFFERENCE BETWEEN THE INTERRUPT MODE AND THE POLL MODE IN THE UARTS?

The INTERRUPT mode uses UART interrupts to transmit and receive characters to and from the UARTs; whereas, POLL mode just polls on the UART device for the transmission and reception of data. Also, the INTERRUPT mode uses software FIFO for data buffering; whereas, POLL mode does not.

WHAT ARE THE DEFAULT SETTINGS FOR THE UART DEVICE?

UART devices are initialized with 57600 baud, 8 data bits, 2 stop bits and no parity. Also, no flow control mechanism is supported in version 1.0 of the library.

HOW CAN I CHANGE THE DEFAULT UART SETTINGS FOR MY APPLICATION?

UARTs can be initialized to the required settings by the passing appropriate parameter in the `open_UARTx()` API during build time or by using the appropriate APIs at run time. Refer to the *ZiLOG Standard Library API Reference Manual* (RM0037) for more details.

I AM USING THE UART IN THE INTERRUPT MODE. WHY DO I SEEM TO LOSE SOME OF THE DATA WHEN I TRY TO PRINT OR TRY TO RECEIVE A LARGE AMOUNT OF DATA?

One of the reasons could be that the software FIFO buffer size is small. Try increasing the size to a bigger value. The default size of the software FIFO is 64. The software FIFO size is defined in the `zsldevinit.asm` file as the BUFF_SIZE macro.

WHEN I CALL OPEN_UARTX() FUNCTION BY CONFIGURING IT IN INTERRUPT MODE, THE CONTROL NEVER COMES BACK TO MY PROGRAM AND MY PROGRAM BEHAVES INDIFFERENTLY. WHY IS THIS?

The `open_UARTx()` function calls the `control_UARTx()` function, which enables the UART interrupt. As a result of this, the UARTx transmit empty interrupt is generated immediately. If the ISR for UART is not installed, the control on the program might be lost. So install the ISR before calling `open_UARTx()` in the INTERRUPT mode. This is not a problem when the standard boot module is used.

WHERE CAN I FIND SAMPLE APPLICATIONS THAT DEMONSTRATE THE USE OF ZSL?

The following applications have been built and tested using the ZiLOG Standard Library:

- eZ80190 Flash Loader application under the following directory:

  ZILOGINSTALL\ZDSII_*product_version*\Applications\eZ80\eZ80190\Flash_Loader\

  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 `Z8Encore!`, `eZ80Acclaim!`, `Crimzon`, or `Z8GP`.

  – *version* is the ZDS II version number. For example, *version* might be `4.11.0` or `5.0.0`.

  This application demonstrates the use of ZSL without involving the standard startup module and zsldevinit.asm file.

- ez80Acclaim! Flash Loader under the following directory:

  ZILOGINSTALL\ZDSII_*product_version*\applications\eZ80Acclaim!\FLashApp\FlashLoaderApp

  This application demonstrates the use of ZSL by using the standard startup module and `zsldevinit.asm` file.

I HAVE USED INIT_UART() AND OTHER FUNCTIONS PROVIDED IN THE RTL. DO I NEED TO CHANGE MY SOURCE CODE BECAUSE OF ZSL?

No. The `sio.c` file of RTL has been modified to call ZSL APIs, so you can continue to use the run-time library (RTL) without changing your source code. But ZiLOG advises you to change your source code to make direct calls to ZSL. This is recommended for the following reasons:

- The calls in RTL support only one UART (UART0 or UART1) at any given time in the library. You cannot switch between the UARTs dynamically.
- There is a small code size increase in the RTL due to the additional overhead of calling ZSL APIs from `sio.c`.
- Future releases of RTL might or might not continue to support this method of indirectly accessing the UARTs via ZSL.

C Standard Library

As described in “Run-Time Library” on page 154, the eZ80Acclaim! 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 eZ80Acclaim! 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. The deviations from the ANSI Standard in these files are summarized in “Library Files Not Required for Freestanding Implementation” on page 166. The standard header files provided with the compiler are listed in the following table and described in detail in “Standard Header Files” on page 319. The following sections describe the use and format of the standard portions of the run-time libraries:

- “Standard Header Files” on page 319
- “Standard Functions” on page 332

Table 12. Standard Headers

Header	Description	Location
`<assert.h>`	Diagnostics	page 320
`<ctype.h>`	Character-handling functions	page 321
`<errno.h>`	Error numbers	page 320
`<float.h>`	Floating-point limits	page 322
`<limits.h>`	Integer limits	page 322
`<math.h>`	Math functions	page 324
`<setjmp.h>`	Nonlocal jump functions	page 326
`<stdarg.h>`	Variable arguments functions	page 326
`<stddef.h>`	Standard defines	page 320
`<stdio.h>`	Standard input/output functions	page 327
`<stdlib.h>`	General utilities functions	page 328
`<string.h>`	String-handling functions	page 330

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_eZ80Acclaim!_<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 needed 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 need to 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>” on page 320
- “Standard Definitions <stddef.h>” on page 320
- “Diagnostics <assert.h>” on page 320
- “Character Handling <ctype.h>” on page 321
- “Limits <limits.h>” on page 322
- “Floating Point <float.h>” on page 322
- “Mathematics <math.h>” on page 324
- “Nonlocal Jumps <setjmp.h>” on page 326
- “Variable Arguments <stdarg.h>” on page 326
- “Input/Output <stdio.h>” on page 327
- “General Utilities <stdlib.h>” on page 328
- “String Handling <string.h>” on page 330

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 defines the `assert()` macro. It refers to the NDEBUG macro, which 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)
```

Macro

assert(expression);	Tests the expression and, if false, prints the diagnostics including the expression, file name, and line number. Also calls exit with nonzero 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: These 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`.

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 eZ80Acclaim! 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 eZ80Acclaim! 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

The following sections describe the mathematical functions:

- "Trigonometric" on page 325
- "Hyperbolic" on page 325
- "Exponential" on page 325
- "Logarithmic" on page 326
- "Power" on page 326
- "Nearest Integer" on page 326

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.

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.

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.

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.

Power

double pow(double x, double y);	Calculates x to the y.
double sqrt(double 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.

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 needed 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. "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 needed 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

The following sections describe the general utilities functions:

- "String Conversion" on page 329
- "Pseudorandom Sequence Generation" on page 329
- "Memory Management" on page 329
- "Searching and Sorting Utilities" on page 329

- “Integer Arithmetic” on page 330

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.

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

The following sections describe the string-handling functions:

- "Copying" on page 331
- "Concatenation" on page 331
- "Comparison" on page 331
- "Search" on page 331
- "Miscellaneous" on page 332

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	asin	assert	atan
atan2	atof	atoi	atol	bsearch
calloc	ceil	cos	cosh	div
exp	fabs	floor	fmod	free
frexp	getchar	gets	isalnum	isalpha
iscntrl	isdigit	isgraph	islower	isprint
ispunct	isspace	isupper	isxdigit	labs
ldexp	ldiv	log	log10	longjmp
malloc	memchr	memcmp	memcpy	memmove
memset	modf	pow	printf	putchar
puts	qsort	rand	realloc	scanf
setjmp	sin	sinh	sprintf	sqrt
srand	sscanf	strcat	strchr	strcmp
strcpy	strcspn	strlen	strncat	strncmp
strncpy	strpbrk	strrchr	strspn	strstr
strtod	strtok	strtol	tan	tanh
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

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);
```

Returns

The arc cosine in the range [0, pi].

Example

```
double y=0.5635;
double x;
x=acos(y);
```

asin

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);
```

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

Computes the principal value of the arc tangent of *x*.

Synopsis

```
#include <math.h>
double atan(double x);
```

Returns

The arc tangent in the range (–pi/2, +pi/2).

Example

```
double y=.1234;
double x;
x=atan(y);
```

atan2

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);
```

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

Converts the string pointed to by nptr to double representation. Except for the behavior on error, `atof` is equivalent to `strtod (nptr, (char **)NULL)`.

Synopsis

```
#include <stdlib.h>
double atof(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(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(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(void *key, void *base, size_t nmemb, size_t size,  int
(*compar)(void *, 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 (void * x, void * y);
int main(void)
{
            int *result;
            result = bsearch(&k, list, 4, sizeof(int), compare);
}

int compare (void * x, void * y)
{
            int a = *(int *) x;
            int b = *(int *) y;
            if (a < b) return -1;
            if (a == b)return 0;
            return 1;
}
```

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

Computes the smallest integer not less than x.

Synopsis

```
#include <math.h>
double ceil(double x);
```

Returns

The smallest integer not less than x, expressed as a `double`.

Example

```
double y=1.45;
double x;
x=ceil(y);
```

cos

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);
```

Returns

The cosine value.

Example

```
double y=.1234;
double x;
x=cos(y);
```

cosh

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);
```

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

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);
```

Returns

The exponential value.

Example

```
double y=.1234;
double x;
x=exp(y);
```

fabs

Computes the absolute value of a floating-point number `x`.

Synopsis

```
#include <math.h>
double fabs(double x);
```

Returns

The absolute value of x.

Example

```
double y=6.23;
double x;
x=fabs(y);
```

floor

Computes the largest integer not greater than x.

Synopsis

```
#include <math.h>
double floor(double x);
```

Returns

The largest integer not greater than x, expressed as a `double`.

Example

```
double y=6.23;
double x;
x=floor(y);
```

fmod

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);
```

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

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);
```

Returns

The value x, such that x is a `double` 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: This function makes a call to hardware-specific functions to get data from the UART. You must either link to ZSL to provide these hardware-specific functions or provide your own equivalent functions. See "Run-Time Library" on page 154.

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: This function makes a call to hardware-specific functions to get data from the UART. You must either link to ZSL to provide these hardware-specific functions or provide your own equivalent functions. See "Run-Time Library" on page 154.

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

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);
```

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

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);
```

Returns

The natural logarithm.

Example

```
double x=2.56;
double y;
y=log(x);
```

log10

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);
```

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 return 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(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(void *s1, 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, 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, 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

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` in the object pointed to by iptr.

Synopsis

```
#include <math.h>
double modf(double value, double *iptr);
```

Returns

The signed fractional part of value.

Example

```
double x=1.235;
double f;
double i;
i=modf(x, &f);
```

pow

Computes the value of 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);
```

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 the %, the following appear in sequence:

- Zero or more flags that modify the meaning of the conversion specification.
- 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.
- 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.
- 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.
- A character that specifies the type of conversion to be applied.

- 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 on the flag characters and conversion characters for the `printf` function, see "printf Flag Characters" on page 354.

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: This function makes a call to hardware-specific functions to get data from the UART. You must either link to ZSL to provide these hardware-specific functions or provide your own equivalent functions. See "Run-Time Library" on page 154.

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: This function makes a call to hardware-specific functions to send data to the UART. You must either link to ZSL to provide these hardware-specific functions or provide your own equivalent functions. See “Run-Time Library” on page 154.

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: This function makes a call to hardware-specific functions to send data to the UART. You must either link to ZSL to provide these hardware-specific functions or provide your own equivalent functions. See “Run-Time Library” on page 154.

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)(void *, void *));
```

Example

```
int lst[]={5,8,2,9};
int compare (void * x, void * y);
qsort (lst, sizeof(int), 4, compare);

int compare (void * x, void * y)
{
            int a = *(int *) x;
            int b = *(int *) y;
            if (a < b) return -1;
            if (a == b)return 0;
            return 1;
}
```

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 like 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 the %, the following appear in sequence:

- An optional assignment-suppressing character *.

- An optional decimal integer that specifies the maximum field width.
- 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 characters 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.
- 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:

- Input white-space characters (as specified by the `isspace` function) are skipped, unless the specification includes a ’[’, ’c,’ or ’n’ character.
- 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.
- 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 “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

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);
```

Returns

The sine value.

Example

```
double x=1.24;
double y;
y=sin(x);
```

sinh

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);
```

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, 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

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);
```

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(char *s, 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, 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(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(char *s1, 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, 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(char *s1, 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(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, 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(char *s1, 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, 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(char *s1, 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(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(char *s1, 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(char *s1, 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

Converts the string pointed to by nptr to `double` 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);
```

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, 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(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

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);
```

Returns

The tangent value.

Example

```
double x=2.22;
double y;
y=tan(x);
```

tanh

Computes the hyperbolic tangent of x.

Synopsis

```
#include <math.h>
double tanh(double 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(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, 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);
```

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" on page 379
- "Running the Compiler from the Command Line" on page 380
- "Running the Assembler from the Command Line" on page 380
- "Running the Linker from the Command Line" on page 380
- "Assembler Command Line Options" on page 381
- "Compiler Command Line Options" on page 383
- "Librarian Command Line Options" on page 385
- "Linker Command Line Options" on page 385

BUILDING A PROJECT FROM THE COMMAND LINE

To build a project from the command line, use the following procedure:

1. Add the ZDS II bin directory (for example, `C:\Program Files\ZiLOG\ZDSII_eZ80Acclaim!_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.

   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, type 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_eZ80Acclaim!_4.11.0\bin\eZ80cc -alias -asm -const:RAM
-debug -define:_EZ80F91 -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_eZ80Acclaim!_4.11.0\include"
-usrinc:"..\include" -cpu:EZ80F91 -bitfieldsize:24 -charsize:8 -doublesize:32
-floatsize:32 -intsize:24 -longsize:32 -shortsize:16 -asmsw:"-cpu:EZ80F91" test.c
```

NOTE: If you use DOS, use double quotation marks for the `-stdinc` and `-usrinc` commands for the C-Compiler. For example:

```
-stdinc:"C:\eZ80\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:\eZ80\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, type 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_eZ80Acclaim!_4.11.0\bin\eZ80asm -debug -genobj
-NOigcase -include:"..\include" -list -NOlistmac -name -pagelen:56
-pagewidth:80 -quiet -warn -NOzmasm -cpu:EZ80F91 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, type the path to the linker and your linker file. For example:

```
C:\PROGRA~1\ZiLOG\ZDSII_eZ80Acclaim!_4.11.0\bin\eZ80lnk @e:\ez80\rtl\testfiles\test\test.linkcmd
```

ASSEMBLER COMMAND LINE OPTIONS

The following table describes the assembler command line options.

NOTE: If you use DOS, use double quotation marks for the `-stdinc` and `-usrinc` commands for the C compiler. For example:

```
-stdinc:"C:\eZ80\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:\eZ80\include}'
```

Table 13. 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`.
`-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.
`-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 13. 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

The following table describes the compiler command line options.

NOTE: If you use DOS, use double quotation marks for the `-stdinc` and `-usrinc` commands for the C compiler. For example:

```
-stdinc:"C:\eZ80include"
```

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:\eZ80\include}'
```

Table 14. 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.
`-cpu:`*cpu*	Sets the CPU. For example: `-cpu:EZ80F91`
`-debug`	Generates debug information for the symbolic debugger.
`-define:`*def*	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`.
`-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. The default is `nokeepasm`.
`-keeplst`	Keeps the assembly listing file (`.lst`). The default is `nokeeplst`.
`-list`	Generates a `.lis` source listing file. The default is `nolist`.
`-listinc`	Displays included files in the compiler listing file.
`-model:`*model*	This option has no effect on the eZ80Acclaim! compiler.
`–modsect`	Generate distinct code segment name for each module.
`-noasm`	Does not assemble the compiler-generated assembly file. This default is `asm`.
`-nodebug`	Does not generate symbol debug information.
`-nogenprint`	A call to `printf()` or `sprintf()` parses the format string at run time to generate the required output. The default is `genprintf`.

Table 14. Compiler Command Line Options (Continued)

Option Name	Description
`-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.
`-nomodsect`	Does not generate distinct code segment name for each module. The code segment is named as "code" for every module. This is the default.
`-nopromote`	Turns off ANSI promotions (deprecated).
`-noquiet`	Displays the title information.
`-noregvar`	This option has no effect on eZ80Acclaim!.
`-promote`	Turns on ANSI promotions.
`-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.
`-regvar`	This option has no effect on eZ80Acclaim!.
`-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 default 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.
`-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 default 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

The following table describes the librarian command line options.

NOTE: If you use DOS, use double quotation marks for the `-stdinc` and `-usrinc` commands for the C compiler. For example:

```
-stdinc:"C:\eZ80\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:\eZ80\include}'
```

Table 15. 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.
`-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.
`-warn`	Displays warnings.

LINKER COMMAND LINE OPTIONS

The following table describes the linker command line options.

NOTE: If you use DOS, use double quotation marks for the `-stdinc` and `-usrinc` commands for the C compiler. For example:

```
-stdinc:"C:\eZ80\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:\eZ80\include}'
```

Table 16. 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:[intel32|omf695]`	Sets the format of the hex file output of the linker to `intel32` (Intel Hex records) or `omf695` (IEEE695 format).
`-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.

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 integrated development environment (IDE). This section covers the following topics:

- "Sample Command Script File" on page 391
- "Supported Script File Commands" on page 392
- "Running the Flash Loader from the Command Processor" on page 412

You can run commands in one of the following ways:

- Using the Command Processor toolbar in the IDE.

  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" on page 22.

- 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 need to precede the script file with an at symbol (@) when passing the path and name of the command 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 "log" on page 400), 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. For example, the following examples are legal:

```
cd "..\path\to\change\to"
cd "..\\path\\to\\change\\to"
cd "../path/to/change/to"
```

The following examples are illegal:

```
cd "..\path/to/change/to"
cd "..\\path\to\change\to"
```

The following table lists ZDS II menu commands and dialog box options that have corresponding script file commands.

Table 17. Script File Commands

ZDS II Menus	ZDS II Commands	Dialog Box Options	Script File Commands	Location
File	New Project		new project	page 400
	Open Project		open project	page 401
	Exit		exit	page 398
Edit	Manage Breakpoints		list bp	page 399
		Go to Code		
		Enable All		
		Disable All		
		Remove	cancel bp	page 393
		Remove All	cancel all	page 393
Project	Add Files		add file	page 392
	Settings (General page)	CPU Family		Table 21
		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	Table 19
		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	
		Jump Optimization	option assembler sdiopt	
	Settings (Code Generation page)	Optimize For	option compiler optspeed	Table 20
		Limit Optimizations for Easier Debugging	option compiler reduceopt	
	Settings (Listing Files page)	Generate C Listing Files (.lis)	option compiler list	Table 20
		With Include Files	option compiler listinc	
		Generate Assembly Source Code	option compiler keepasm	
		Generate Assembly Listing Files (.lst)	option compiler keeplst	
	Settings (Preprocessor page)	Preprocessor Definitions	option compiler define	Table 20
		Standard Include Path	option compiler stdinc	
		User Include Path	option compiler usrinc	

Table 17. Script File Commands (Continued)

ZDS II Menus	ZDS II Commands	Dialog Box Options	Script File Commands	Location
	Settings (Advanced page)	Generate Printfs Inline Distinct Code Segment for Each Module	option compiler genprintf option compiler modsect	Table 20
	Settings (Deprecated page)	Disable ANSI Promotions	option compiler promote	Table 20
	Settings (Librarian page)	Output File Name	option librarian outfile	Table 22
	Settings (ZSL page)	Include ZiLOG Standard Library (Peripheral Support) Ports Uarts	option middleware usezsl option middleware zslports option middleware zsluarts	Table 24
	Settings (Commands page)	Link Configuration 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	Table 23
	Settings (Objects and Libraries page)	Additional Object/Library Modules Standard Included in Project Use Standard Startup Linker Commands Use C Runtime Library Floating Point Library ZiLOG Standard Library (Peripheral Support)	option linker objlibmods option linker startuptype option linker startuptype option linker startuplnkcmds option linker usecrun option linker fplib	Table 23
	Settings (Address Spaces page)	ROM RAM ExtIO IntIO FlashInfo	option linker rom option linker ram option linker extio option linker intio option linker flashinfo	Table 23
	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	Table 23

Table 17. Script File Commands (Continued)

ZDS II Menus	ZDS II Commands	Dialog Box Options	Script File Commands	Location
	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	Table 23
	Settings (Debugger page)	Use Page Erase Before Flashing Target Setup Add Copy Delete Debug Tool Setup	 target set target options target create target copy debugtool set debugtool set	 page 411 page 411 page 410 page 409 page 396 page 396
	Export Makefile		makfile makefile	page 400 page 400
Build	Build		build	page 393
	Rebuild All		rebuild	page 408
	Stop Build		stop	page 409
	Set Active Configuration		set config	page 408
	Manage Configurations		set config delete config	page 408
Debug	Stop Debugging		quit	page 407
	Reset		reset	page 408
	Go		go	page 399
	Break		stop	page 409
	Step Into		stepin	page 409
	Step Over		step	page 409
	Step Out		stepout	page 409
Tools	Flash Loader			page 412
	Calculate File Checksum		checksum	page 394

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 is a sample command script file:

```
; change to correct default directory
cd "m:\eZ80Acclaim!\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 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
- batch
- bp
- build
- cancel all
- cancel bp
- cd
- checksum
- debugtool copy
- debugtool create
- debugtool get
- debugtool help
- debugtool list
- debugtool save
- debugtool set
- debugtool setup
- defines
- delete config
- examine (?) for Expressions
- examine (?) for Variables
- exit
- fillmem
- go
- list bp
- loadmem
- log
- makfile or makefile
- new project
- open project
- option
- print
- pwd
- quit
- rebuild
- reset
- savemem
- set config
- step
- stepin
- stepout
- stop
- target copy
- target create
- target get
- target help
- target list
- target options
- target save
- target set
- target setup
- wait
- wait bp

In the following syntax descriptions, items enclosed in angle brackets (< >) need to be replaced with actual values, items enclosed in square brackets ([ ]) are optional, double quotes (") indicate where double quotes must exist, and all other text needs to be included as is.

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 is 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 is the syntax of the `batch` command:

```
batch [wait] "<[path\]<filename>"
```

`wait`	blocks processing of the current script until the invoked batch file completes—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 in the active 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
```

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 is the syntax of the `build` command:

```
build
```

cancel all

The `cancel all` command clears all breakpoints in the active file. The following is the syntax of the `cancel all` command:

```
cancel all
```

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 is the syntax of the `cancel bp` command:

```
cancel bp <index>
```

For example:

```
cancel bp 3
```

cd

The `cd` command changes the working directory to *dir*. The following is 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 is 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
```

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="eZ80190"
```

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 is 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 is 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 is 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 is 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 is 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 examine command 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 are 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 is the syntax:

? <*expression*>

The following are 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 is the syntax of the `exit` command:

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 "Fill Memory" on page 297). The following is the syntax of the `fillmem` command:

```
fillmem SPACE="<displayed spacename>" FILLVALUE="<hexcadecimal value>"
   [STARTADDRESS="<hexadecimal address>"] [ENDADDRESS="<hexadecimal address>"]
```

If `STARTADDRESS` and `ENDADDRESS` are not specified, all 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 are 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 is 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" on page 299). The following is 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 eZ80Acclaim! IDE can only create eZ80Acclaim!-based projects).
- <`exe|lib`> The type parameter 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" "eZ80Acclaim!" "exe"
new project "test1.zdsproj" "eZ80Acclaim!" "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 is 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 is 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=ez80F91
```

NOTE: Many of these script file options are also available from the command line. For more details, see “Running ZDS II from the Command Line” on page 379.

The following table lists some command line examples and the corresponding script file commands.

Table 18. Command Line Examples

Script File Command Examples	Corresponding Command Line Examples
`option compiler keepasm = TRUE`	`eZ80cc -keepasm`
`option compiler keepasm = FALSE`	`eZ80cc -nokeepasm`
`option compiler const = RAM`	`eZ80cc -const:RAM`
`option assembler debug = TRUE`	`eZ80asm -debug`
`option linker igcase = "FALSE"`	`eZ80link -NOigcase`
`option librarian warn = FALSE`	`eZ80lib -nowarn`

The following script file options are available:

- “Assembler Options” on page 403
- “Compiler Options” on page 403
- “Debugger Options” on page 404
- “General Options” on page 404
- “Librarian Options” on page 405
- “Linker Options” on page 405
- “ZSL Options” on page 406

Assembler Options

Table 19. 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) check box	TRUE, FALSE
`listmac`	Assembler page, Expand Macros check box	TRUE, FALSE
`pagelen`	Assembler page, Page Length field	integer
`pagewidth`	Assembler page, Page Width field	integer
`quiet`	Toggles quiet assemble.	TRUE, FALSE
`sdiopt`	Toggles Jump Optimization.	TRUE, FALSE

Compiler Options

Table 20. Compiler Options

Option Name	Description or Corresponding Option in Project Settings Dialog Box	Acceptable Values
`define`	Preprocessor page, Preprocessor Definitions field	string (separate multiple defines with semicolons)
`genprintf`	Advanced page, Generate Printfs Inline check box	TRUE, FALSE
`keepasm`	Listing Files page, Preprocessor Definitions field	
`keeplst`	Listing Files page, Generate Assembly Listing Files (.lst) check box	TRUE, FALSE
`list`	Listing Files page, Generate C Listing Files (.lis) check box	TRUE, FALSE
`listinc`	Listing Files page, With Include Files check box Only applies if `list` option is currently true.	TRUE, FALSE
`modsect`	Advanced page, Distinct Code Segment for Each Module check box	TRUE, FALSE

Table 20. Compiler Options (Continued)

Option Name	Description or Corresponding Option in Project Settings Dialog Box	Acceptable Values
`optspeed`	Toggles optimizing for speed.	TRUE (optimize for speed), FALSE (optimize for size)
`promote`	Deprecated page, Disable ANSI Promotions check box **NOTE:** This option is deprecated.	TRUE, FALSE (FALSE disables the ANSI promotions)
`reduceopt`	Code Generation page, Limit Optimizations for Easier Debug check box	TRUE, FALSE
`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 21. 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 check box	TRUE, FALSE
`igcase`	General page, Ignore Case of Symbols check box	TRUE, FALSE
`outputdir`	General page, Intermediate Files Directory field	string (path)
`warn`	General page, Show Warnings check box	TRUE, FALSE

Librarian Options

Table 22. 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 23. 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
`exeform`	Output page, Executable Formats area	string (one or more of “IEEE 695” or “Intel Hex32”)
`extio`	Address Spaces page, ExtIO field	string (address range in the format <*low*>-<*high*>)
`flashinfo`	Address Spaces page, FlashInfo field	string (min-max, for example, "00-FF")
`fplib`	Objects and Libraries page, Floating Point Library drop-down list box	string (“real”, “dummy”, or “none”)
`intio`	Address Spaces page, IntIO 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 check box	TRUE, FALSE
`maxhexlen`	Output page, Maximum Bytes per Hex File Line drop-down list box	integer (16, 32, 64, or 128)

Table 23. Linker Options (Continued)

Option Name	Description or Corresponding Option in Project Settings Dialog Box	Acceptable Values
`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 check box	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 check box	TRUE, FALSE
`rom`	Address Spaces page, Constant Data (ROM) field	string (address range in the format "<*low*>-<*high*>")
`sort`	Output page, Sort Symbols By buttons	string
`startuplnkcmds`	Objects and Libraries page, Use Standard Startup Linker Commands check box	TRUE, FALSE
`startuptype`	Objects and Libraries page, C Startup Module area	string ("standard" or "included")
`useadddirectives`	Commands page, Additional Directives check box	TRUE, FALSE
`usecrun`	Objects and Libraries page, Use C Runtime Library check box	TRUE, FALSE
`undefisfatal`	Warnings page, Treat Undefined Symbols as Fatal check box	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

ZSL Options

For ZSL options, the *tool_name* is `middleware`. For example:

```
option middleware usezsl = TRUE
```

Table 24. ZSL Options

Option Name	Corresponding Option in Project Settings Dialog Box	Acceptable Values
`usezsl`	ZSL page, Include ZiLOG Standard Library (Peripheral Support) check box or Objects and Libraries page, ZiLOG Standard Library (Peripheral Support) check box	TRUE, FALSE
`zslports`	ZSL page, Ports area	comma-delimited string (“Port A,Port D”)
`zsluarts`	ZSL page, Uarts area	comma-delimited string (“UART0,UART1”)

Next: “print” on page 407

Previous: “Linker Options” on page 405

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 is 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 is the syntax of the pwd command:

```
pwd
```

quit

The `quit` command ends the current debug session. The following is 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 is 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 is 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) check box on the Debugger tab of the Options dialog box (see "Options—Debugger Tab" on page 127), 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" on page 298). The following is 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 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 is the syntax of the `set config` command:

```
set config "config_name" ["copy_from_config_name"]
```

The `set config` command does the following:

- 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 like `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 is 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 is 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 is the syntax of the `stepout` command:

```
stepout
```

stop

The `stop` command stops the execution of program code. The following is 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 is the syntax of the `target create` command:

```
target create NAME="<target name>" CPU="<cpu name>"
```

For example:

```
target create NAME="mytarget" CPU="eZ80190"
```

target get

The `target get` command displays the current value for the given data item for the active target. The following is 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 is 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="eZ80"
```

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 would be `C:\Program Files\ZiLOG\ZDSII_eZ80Acclaim!_<version>`, where *<version>* might be `4.10.0` 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 is 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 is 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 is 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.

Use the following procedure to run the Flash Loader:

1. Create a project or open a project with EZ80L92, EZ80190, EZ80F91, EZ80F92, or EZ80F93 selected in the CPU field on the General page of the Project Settings dialog box (see “Project Settings—General Page” on page 57).
2. The Flash Loader requires RAM memory for execution and also requires the correct external Flash memory chip select parameters and memory limits. If there is internal RAM memory, the Flash Loader uses that memory for execution. The parameters are entered in the Configure Target dialog box (see “Setup” on page 96). The ZPAK II Ethernet address is also needed by the Flash Loader and can be entered in the Setup TCP/IP Communication dialog box (see “Debug Tool” on page 102).

3. In the Command field (in the Command Processor toolbar), type in the command sequences in the following sections to use the Flash Loader:
   – “Displaying Flash Help” on page 413
   – “Setting Up Flash Options” on page 413
   – “Executing Flash Commands” on page 414
   – “Examples” on page 414

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 FLASHBASE = "<*address*>"	Start location for external Flash memory
Flash Options NUMFLASH = <1-8>	Number of stack Flash devices
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 internal and external memory
Flash Options MANUF="<*manufacture name*>" DEVICE="<*Name of Device*>"	Set the Flash device
Flash Options NEBF	Do not erase before flash
Flash Options EBF	Erase before flash
Flash Options NEIP	Do not erase info page
Flash Options EIP	Erase info page
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
Flash Options NIP	No info page
Flash Options IP	Info page

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 for a target with an eZ80F91 and external Flash memory:

```
FLASH Options INTMEM
FLASH Options "c:\testing\test.hex"
FLASH Options OFFSET="0x1000"
FLASH Options EBF
FLASH BURN REPEAT
```

or

```
flash options intmem
flash options "c:\testing\test.hex"
flash options offset="0x1000"
flash options ebf
flash burn repeat
```

The file `test.hex` is loaded into internal Flash memory with a value of `0x1000` added to all addresses. The Flash memory is erased before flashing. 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 with a offset value of `0x1000`.

```
target set "flashManufacturer" "Micron"
target set "flashDevice" "MT28F008B3xx-xxB"
FLASH VERIFY
```

The file `test.hex` (from the first example) is verified against the external specific Flash device.

```
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.

Compatibility Issues

The following sections describe assembler and compiler compatibility issues:

- "Asssembler Compatibility Issues" on page 416
- "Compiler Compatibility Issues" on page 419

ASSSEMBLER COMPATIBILITY ISSUES

The following table shows the equivalences between eZ80Acclaim! directives and those of other assemblers that are also supported by the eZ80Acclaim! assembler. ZMASM directives that are compatible with eZ80Acclaim! directives are also listed. The eZ80Acclaim! assembler directives in the left column are the preferred directives, but the assembler also accepts any of the directives in the right column.

Table 25. eZ80Acclaim! Directive Compatibility

eZ80Acclaim! Assembler	Compatible With
`ALIGN`	`.align`
`ASCII`	`.ascii`
`ASCIZ`	`.asciz`
`ASECT`	`.ASECT`
`ASG`	`.ASG`
`ASSUME`	`.ASSUME`
`BES`	`.bes`
`BREAK`	`.$BREAK,.$break`
`BSS`	`.bss`
`CHIP`	`chip, .cpu`
`CONTINUE`	`.$CONTINUE, .$continue`
`DATA`	`.data`
`DB`	`.byte, .ascii, DEFB, FCB, STRING, .STRING, byte, .asciz`
`DD`	`.double`
`DEFINE`	`.define`
`DF`	`.float`
`DL`	`.long, long`
`DR`	`<none>`

Table 25. eZ80Acclaim! Directive Compatibility (Continued)

eZ80Acclaim! Assembler	Compatible With
DS	.block
DW	.word, word, .int
DW24	.word24, .trio, .DW24
ELIF	.ELIF, .ELSEIF, ELSEIF, .$ELSEIF, .$elseif
ELSE	.ELSE, .$ELSE, .$else
END	.end
ENDIF	.endif, .ENDIF, ENDC, .$ENDIF, .$endif
ENDMAC	.endm, ENDMACRO, .ENDMACRO, .ENDMAC, ENDM, .ENDM, MACEND, .MACEND
ENDSTRUCT	.ENDSTRUCT
ERROR	.emsg
EQU	.equ, .EQU, EQUAL, .equal
EVAL	.EVAL
FCALL	.FCALL
FILE	.file
FRAME	.FRAME
GREG	GREGISTER
IF	.if, .IF, IFN, IFNZ, COND, IFTRUE, IFNFALSE, .$IF, .$if, .IFTRUE
INCLUDE	.include, .copy
LIST	.list <on> or <off>, .LIST
MACCNTR	<none>
MACEXIT	<none>
MACLIST	<none>
MACNOTE	.mmsg
MACRO	.macro, .MACRO
MLIST	<none>
MNOLIST	<none>
NEWBLOCK	.NEWBLOCK
NEWPAGE	.page [<len>] [<width>], PAGE

Table 25. eZ80Acclaim! Directive Compatibility (Continued)

eZ80Acclaim! Assembler	Compatible With
NIF	IFZ, IFE, IFFALSE, IFNTRUE, .IFNTRUE
NOLIST	.NOLIST
NOSAME	IFDIFF, IFNSAME
ORG	.org, ORIGIN
PE	V
P0	NV
POPSEG	<none>
PRINT	<none>
PT	<none>
PUSHSEG	<none>
REPEAT	.$REPEAT, .$repeat
SAME	IFNDIFF, IFSAME
SBLOCK	.SBLOCK
SCOPE	<none>
SEGMENT	.section, SECTION
STRUCT	.STRUCT
TAG	.TAG
TEXT	.text
TITLE	.title
UNTIL	.$UNTIL, .until
VAR	.VAR, SET, .SET
VECTOR	<none>
WARNING	.wmsg, MESSAGE
WEND	.$WEND, .$wend
WHILE	.$WHILE, .$while
XDEF	.global, GLOBAL, .GLOBAL, .public, .def, public
XREF	EXTERN, EXTERNAL, .extern, .ref
ZIGNORE	<none>

Table 25. eZ80Acclaim! Directive Compatibility (Continued)

eZ80Acclaim! Assembler	Compatible With
ZSECT	.sect
ZUSECT	.USECT

COMPILER COMPATIBILITY ISSUES

NOTE: Use of the #pragmas documented in this section should not be necessary in ZDS II release 4.10 and later. ZiLOG does not recommend their use, especially in new projects because it is extremely difficult to validate that they continue to work correctly as the compiler is updated and in all circumstances. They continue to be supported as they have been in older releases and will be accepted by the compiler.

Compiler options can be set in the Project Settings dialog box (on the C pages) or by using the following `#pragma` directives:

#pragma alias	#pragma nobss	#pragma nooptlink
#pragma noalias	#pragma jumpopt	#pragma optsize
#pragma cpu <cpu name>	#pragma nojumpopt	#pragma optspeed
#pragma globalcopy	#pragma localcopy	#pragma peephole
#pragma noglobalcopy	#pragma nolocalcopy	#pragma nopeephole
#pragma globalcse	#pragma localcse	#pragma promote
#pragma noglobalcse	#pragma nolocalcse	#pragma nopromote
#pragma globaldeadvar	#pragma localfold	#pragma sdiopt
#pragma noglobaldeadvar	#pragma nolocalfold	#pragma nosdiopt
#pragma globalfold	#pragma localopt	#pragma stkck
#pragma noglobalfold	#pragma nolocalopt	#pragma nostkck
#pragma intrinsics: <state>	#pragma noopt	#pragma strict
#pragma nointrinsics	#pragma optlink	#pragma nostrict

If the `#pragma` directive is inserted in your code, it overrides the selections you made in the Project Settings dialog box.

#pragma alias

Enables alias checking. The compiler assumes that program variables can be aliased. This pragma is the default.

#pragma noalias

Disables alias checking. Before using this pragma, be sure that the program does not use aliases, either directly or indirectly. An alias occurs when two variables can reference the same memory location. The following example illustrates an alias:

```
func()
{
   int x,*p;
   p = &x;   /* both “x” and “*p” refer to same location */
  .
  .
  .
}
```

If both *p and x are used below the assignment, then malignant aliases exist and the NOALIAS switch must not be used. Otherwise, alias is benign, and the NOALIAS switch can be used.

#pragma cpu <cpu name>

Defines the target processor to the compiler.

#pragma globalcopy

Enables global copy propagation.

#pragma noglobalcopy

Disables global copy propagation.

#pragma globalcse

Enables global common elimination unless local common subexpressions are disabled.

#pragma noglobalcse

Disables global copy subexpression elimination.

#pragma globaldeadvar

Enables global dead variable removal.

#pragma noglobaldeadvar

Disables global dead variable removal.

#pragma globalfold

Enables global constant folding.

#pragma noglobalfold

Disables global constant folding.

#pragma intrinsics: <state>

Defines whether the compiler-defined intrinsic functions are to be expanded to inline code.

NOTE: The intrinsic property is only available for compiler-defined intrinsic functions; user-defined intrinsics are not supported.

<*state*> can be ON or OFF. This pragma, with <*state*> ON, is the default.

#pragma nointrinsics

Disables the INTRINSICS switch.

#pragma nobss

Does not put uninitialized static data in bss segment, instead it puts it in data segment and initializes it at link time.

#pragma jumpopt

Enables jump optimizations.

#pragma nojumpopt

Disables jump optimizations.

#pragma localcopy

Enables local copy propagation.

#pragma nolocalcopy

Disables local copy propagation.

#pragma localcse

Enables local common subexpression elimination.

#pragma nolocalcse

Disables local and global common subexpression elimination.

#pragma localfold

Enables local constant folding.

#pragma nolocalfold

Disables local constant folding.

#pragma localopt

Enables all local optimizations.

#pragma nolocalopt

Disables all local optimizations.

#pragma noopt

Disables all optimizations.

#pragma optlink

Enables optimized linkage calling conventions.

#pragma nooptlink

Disables optimized linkage calling conventions.

#pragma optsize

Optimizes code to minimize size.

#pragma optspeed

Optimizes code to minimize execution time.

#pragma peephole

Enables peephole optimizations.

#pragma nopeephole

Disables peephole optimizations.

#pragma promote

Enables ANSI integer promotions.

#pragma nopromote

Disables ANSI integer promotions.

#pragma sdiopt

Performs span-dependent instruction optimization. This optimization results in branches generated by the compiler taking the shortest form possible. This pragma is the default.

#pragma nosdiopt

Disables span-dependent instruction optimizations.

#pragma stkck

Performs stack checking.

#pragma nostkck

Does not perform stack checking.

#pragma strict

Checks for conformance to the ANSI standard and its obsolescent features. These include old-style parameter type declarations, empty formal parameter lists, and calling functions with no prototype in scope. When any of these features are used a warning is flagged. The compiler requires this switch for proper code generation because it makes use of a static frame.

#pragma nostrict

Does not flag warnings for obsolete features.

Index