Z8/Z8Plus C-Compiler User's Manual

UM002801-COR1099

ZiLOG, Inc.
910 East Hamilton Ave., Suite 110
Campbell, CA 95008
Telephone: (408) 558-8500
FAX: (408) 558-8300
Internet: http://www.zilog.com

PREFACE

ABOUT THIS MANUAL

We recommend that you read and understand everything in this manual before setting up and using the product. However, we recognize that users have different styles of learning. Therefore, we have designed this manual to be used either as a how-to procedural manual or a reference guide to important data.

The following conventions have been adopted to provide clarity and ease of use:

- Universe Medium 10-point all-caps is used to highlight to the following items:
  - commands , displayed messages
  - menu selections, pop-up lists, button, fields, or dialog boxes
  - modes
  - pins and ports
  - program or application name
  - instructions, registers, signals and subroutines
  - an action performed by the software
  - icons
- `Courier Regular 10-point` is used to highlight the following items
  - bit
  - software code
  - file names and paths
  - hexadecimal value
- Grouping of Actions Within A Procedure Step

  Actions in a procedure step are all performed on the same window or dialog box. Actions performed on different windows or dialog boxes appear in separate steps.

TABLE OF CONTENTS

Chapter Title and Subsections **Page**

Chapter Title and Subsections **Page**

LIST OF TABLES

LIST OF FIGURES

Chapter 1
Introduction

Introduction

The Z8/Z8Plus C-Compiler conforms to the ANSI's definition of a "freestanding implementation", with the exception that doubles are 32 bits. In accordance with the definition of a freestanding implementation, the compiler accepts programs which 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>. This release supports more of the standard library than is required of a freestanding implementation, as described in Chapter 4, Run Time Environment. The compiler also supports both dynamic and static frames.

There are several language extensions supported in this version, including interrupt functions and memory space accesses.

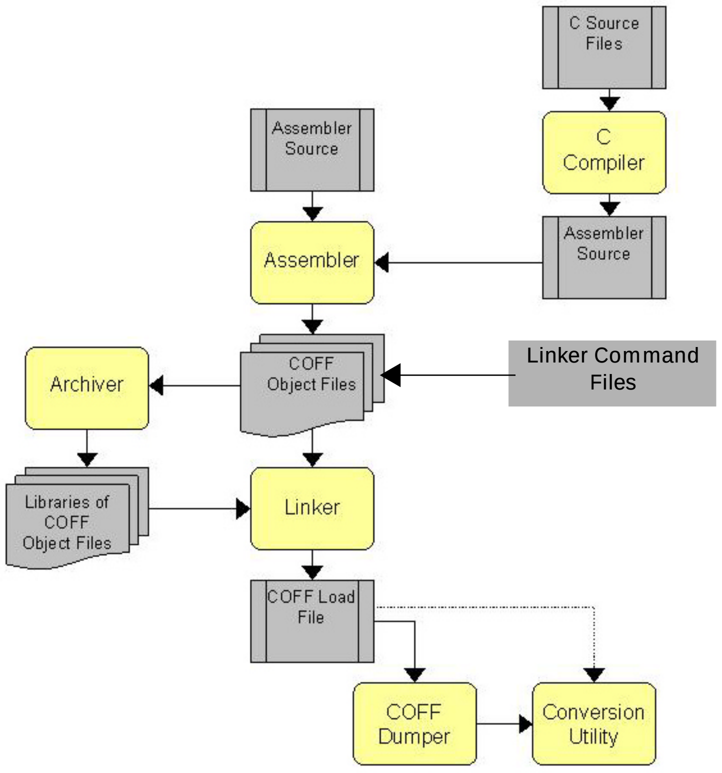


FIGURE 1-1. DEVELOPMENT FLOW

ZDS Environment

ZiLOG Developer Studio is an integrated development environment with a standard Windows 95/98/NT user interface that allows access to all of ZiLOG's development tools without having to alternate from one program to another. These development tools include a language sensitive editor, project manager, assembler, linker, and a symbolic debugger. ZDS supports the entire Z8, Z8Plus, and DSP families of ZiLOG processors.

ZDS allows the user to:

- Create project files and add or remove files to and from the project
- Create and edit a source file.
- Download, execute, debug, and analyze code
- Build and link a project file
- Compile, assemble and link files
- Prepare code for ROM release (one-time programming)

Run-time model

The compiler provides two memory models, small and large. In the small memory model the stack and data is stored in the register file. In the large model the stack and data is stored in external data memory. For the small memory model two call frames are supported; static and dynamic. Long is 32 bits, Int is 16 bits, address to register file is 8 bits and all other addresses are 16 bits. The following startup programs are included:

- For Z8, the `Z8inits.s` (small model initialization file) and the `Z8initx.s` (large model initialization file)
- For Z8Plus, the `Z8plusinit.s` (small model initialization file)

These programs clear the .bss section, set the register pointer and port mode registers, call main and copy the initialized data.

MINIMUM REQUIREMENTS

For the C-Compiler to run properly with ZDS, the host system must meet the following minimum requirements:

- The Z8/Z8Plus compiler requires Windows95 or Windows/NT. The compiler generates assembler language source, which can be assembled and linked using the UNIX, DOS or Windows versions of the ZiLOG assembler, archiver and linker.
- IBM PC (or 100-percent compatible) Pentium-based machine
- 75MHz,16MB Memory
- VGA Video Adapter
- Hard Disk Drive (12 MB free space)
- CD-ROM drive
- Mouse or Pointing Device
- Microsoft Windows 95/98/NT
- To use the ZDS debugger, an emulator is needed that corresponds to the processor required for configuration

INSTALLING THE Z8/Z8PLUS C-COMPILER

To install the Z8/Z8Plus C-Compiler, insert the Z8/Z8Plus C-Compiler CD ROM and follow the onscreen prompts

After installing the Z8/Z8Plus C-Compiler, the compiler's installation path is set in the Window's registry. When installing ZDS 3.00 or later, ZDS automatically looks for the C-Compilers installation path and loads the corresponding DLL from that path.

This is effective for the following compiler versions:

- Z380 10.00 or later
- Z3xx B0.00 or later
- Z8 C1.00 or later

NOTE: Older compiler versions require the user to copy the compiler's DLLs to the ZDS installation directory.

REGISTRY KEYS

The following keys are written to the window's registry during the C-compiler installation:

- For Z380 Installation
  - `+ HKEY_LOCAL_MACHINE\Software\ZiLOG\C Compiler\Z380`
  - `+ Z380 Key has Path value which tells where the Z380 is located`
- For Z3xx Installation
  - `+ HKEY_LOCAL_MACHINE\Software\ZiLOG\C Compiler\Z3xx`
  - `+ Z3xx Key has Path value which tells where the Z3xx is located`
- For Z8/Z8Plus Installation
  - `+ HKEY_LOCAL_MACHINE\Software\ZiLOG\C Compiler\Z8`
  - `+ Z8 Key has Path value which tells where the Z8 is located`

INSTALLING ZDS

Perform the following steps to install ZDS:

1. Insert the ZiLOG Developer Studio CD-ROM into the host CD ROM drive. The Emulator Software Setup window appears.
2. In the Select Components dialog box check ZiLOG Developer Studio.
3. Click Next. The ZiLOG Developer Studio window appears.
4. Click Next to accept the licensing agreement. Immediately after the agreement is accepted, the Choose Destination Location dialog box appears.
5. Click Next to install ZDS in the default directory. Click Browse to change the ZDS install directory.
6. After selecting the appropriate install directory, click next. The Select Program Folder dialog box appears.
7. Click Next to add the ZDS program icon to the ZiLOG Developer Studio program folder. To create a personalized folder, type the folders name in the Program Folders field.
8. Click Next. The Installing ZDS Program Files progress bar appears.
9. After installation, the Setup Complete dialog box appears. Check View README File to view the read me file containing the ZDS release notes. Check Launch ZiLOG Developer Studio to start ZDS at the end of the installation.
10. Click Finish to complete the ZDS installation.

TECHNICAL SUPPORT

Technical support for ZDS or the C-Compilers can be accessed on the web or by phone. The Zilog Internet Home Page address is http://www.zilog.com.

To get the latest software upgrades for ZDS, go to the ZDS home page and select Downloadable Software from the main menu.

To get the latest software upgrades for the Z8/Z8Plus C-Compiler, contact tools@zilog.com.

ZILOG has a worldwide customer support center located in Austin, Texas. The customer support center is open from 7 a.m. to 7 p.m. Central Time. The customer support toll-free number for the United States and Canada is 1-877-ZiLOGCS (1-877-945-6427). For calls outside of the United States and Canada dial 512-306-4169. The FAX number to the customer support center is 512-306-4072. Customers can also E-mail the support center at csupport@zilog.com

SAMPLE SESSION

The Z8/Z8Plus C-Compiler is a modular component that is part of the ZDS development environment. Users should become familiar with ZDS and configure the settings before programming or downloading files. This chapter orients the user on using ZDS and configuring the compiler for the Z8/Z8Plus family of processors. For more information on installing ZDS, consult the ZDS Quick Start Guide or the ZDS on-line help.

CREATE A PROJECT AND SELECT A PROCESSOR

The user must create a project and select a processor before creating or opening a C-file. Perform the following steps to create a new project and select a processor :

1. Open ZDS by selecting Start>Programs>Zilog Developer Studio> ZDS.
2. Choose New Project from the File menu. The New Project dialog box appears. See Figure 1-2.

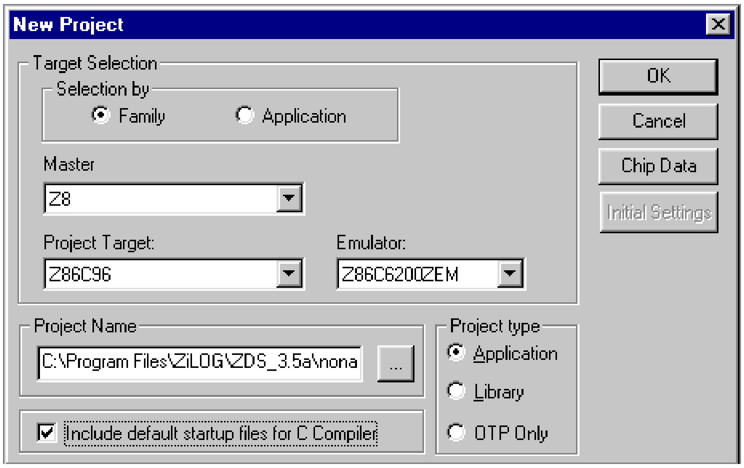

Target Selection
Selection by: (● Family) (Application)
Master: Z8
Project Target: Z86C96
Emulator: Z86C6200ZEM
Project Name: C:\Program Files\ZiLOG\ZDS_3.5a\nona
Project type: (● Application) (Library) (OTP Only)
☑ Include default startup files for C Compiler

FIGURE 1-2. NEW PROJECT DIALOG BOX

3. Select `Family` in the Selection by field.
4. For Z8, select Z8 from the Master pop-up list. For Z8Plus, select Z8Plus from the Master pop-up list.
5. Select the processor from the Project Target pop-up list.
6. Select an emulator or simulator from the Emulator pop-up list.
7. Click on the browse button (...) in the Project Name field. The New Project Browse dialog box appears.

8. Enter the file name and select a path in the New Project Browse dialog box.
9. Click Save. The file name appears in the Project Name field in the New Project dialog box.
10. Select Application from the Project type field. This selection enables the linker.
11. Check Include default startup files for C Compiler. This option must be checked to enable the Wizard. To manually add the necessary files for the C-Compiler, see Manually Configuring the Compiler on page 1-11.
12. Click on Chip Data to view specifications for the selected Project Target.

NOTE: Fields in the Chip Data page are read-only and can not be modified.

13. Click OK. The new project is saved with the file name specified in the New Project Browse dialog box.

CONFIGURING THE COMPILER USING THE WIZARD

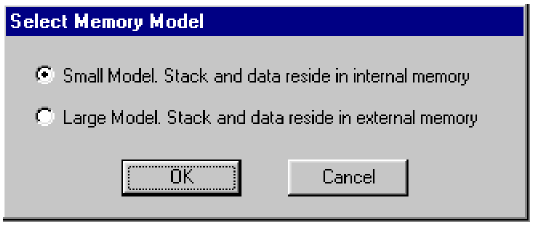
The Wizard is enabled when the Include default startup files for C Compiler option is checked in the New Project dialog box.

NOTE: The Wizard is only available for ZDS version 3.5 and later. To configure the compiler to run with static frames you must use the small model.

Perform the following steps after clicking OK in the New Project Browse dialog box:

1. Select either Small Model or Large Model in the Select Memory Model Dialog box, see Figure 1-3. To use static frames select Small Model.

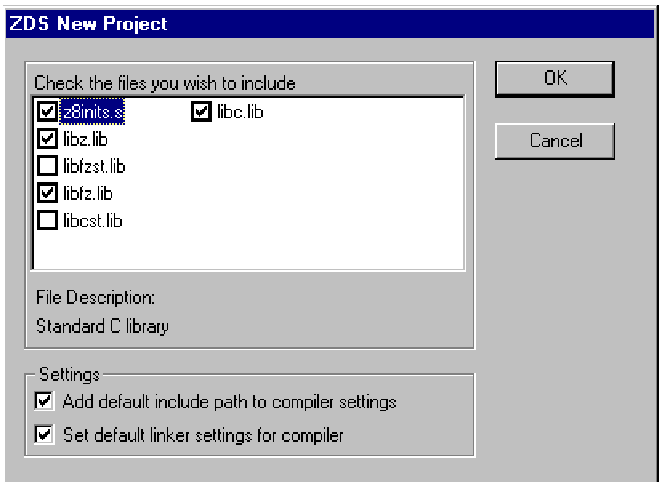
   NOTE: Only the small model is available for the Z8Plus.


Select Memory Model

[✓ Small Model. Stack and data reside in internal memory]
Large Model. Stack and data reside in external memory

FIGURE 1-3. SELECT MEMORY MODEL DIALOG BOX

2. Click OK. The ZDS New Project Dialog box appears. See Figure 1-4.


ZDS New Project

Check the files you wish to include

- [✓] z8inits.s
- [✓] libz.lib
- [ ] libfzst.lib
- [✓] libfz.lib
- [ ] libcst.lib
- [✓] libc.lib

File Description:
Standard C library

Settings
- [✓] Add default include path to compiler settings
- [✓] Set default linker settings for compiler

FIGURE 1-4. ZDS NEW PROJECT DIALOG BOX

NOTE: For Z8Plus the `.s` file is `Z8plusinit.s`.

3. Select the files ti include in the **Check the files you wish to include** window. For more information on which files to include see Installed files on page 3-4.
4. Select **Set default include path to compiler settings** in the Settings window. *Selecting this option sets the path of the include files in the* **Additional include directories** *field in the C-Compiler preprocessor page.*
5. Select **Set default linker settings for compiler**.
6. Click OK. The initialization file for the selected model appears in the project viewer window. See Figure 1-5.

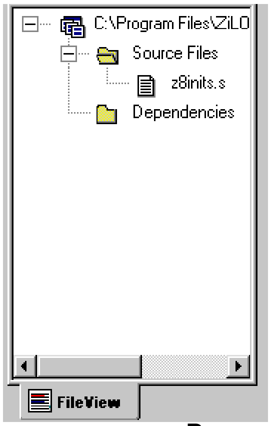

C:\Program Files\ZiLO
Source Files
z8inits.s
Dependencies

FIGURE 1-5. INITIALIZATION FILE IN PROJECT VIEWER WINDOW

Enable Static Frames

Perform the following additional steps to enable static frames:

1. To use static frames select Memory from the Category pop-up list in the C-Compiler Settings Options dialog box. The Memory page appears.
2. Select the Static Frame options. If the Static Frame option is not enabled then select a small memory model in the Memory Model Selection window, see Figure 1-12.
3. Click Apply.

Create a File

Perform the following steps to create a new C file :

1. Select Add to Project>New from the Project menu. The Insert New Files Into Project dialog box appears.
2. Select C Files from the Files of type pop-up menu.
3. Type a file name in the File Name field.
4. Click Open. The new file name appears in the Project Viewer window with a `.c` suffix, and a blank Edit window also appears.
5. Type the following code in the edit window:

```
#include <stdlib.h>
int randnum;
int main()
{
    srand(10);
   randnum=rand();
   randnum=rand();
}
```

6. Close and save the file.

NOTE: Skip the Manually Configuring the Compiler section if you configured the compiler using the wizard.

MANUALLY CONFIGURING THE COMPILER

The user can manually add files and configure the settings for the C-Compiler.

After creating a project the user must add or create new files. A previously created project has the following attributes saved with it:

- Target settings
- Assembler and Linker settings for the specified target
- Source files (including header files)

The user must first add the necessary files for the compiler to function properly. The following examples are based on using a small model.

Perform the following steps to add files:

1. Select **Open Project** from the File menu. The **Open Project** dialog box appears.
2. In the **Open Project** dialog box, select the project that was created in the previous exercise. The project appears in the Project Viewer window.
3. Select **Add to Project>Files** from the **Project** menu. The **Insert Files into Project** dialog box appears. See Figure 1-6.

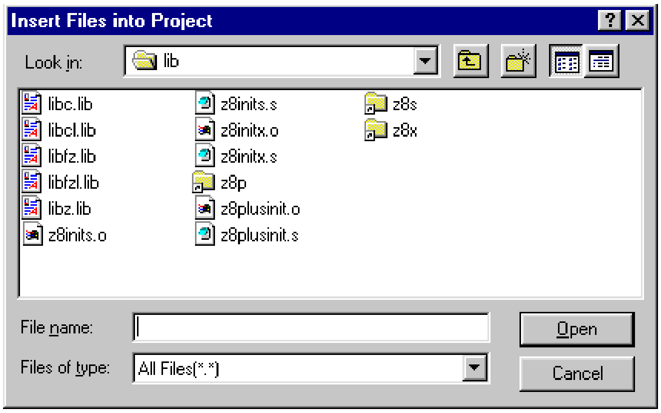


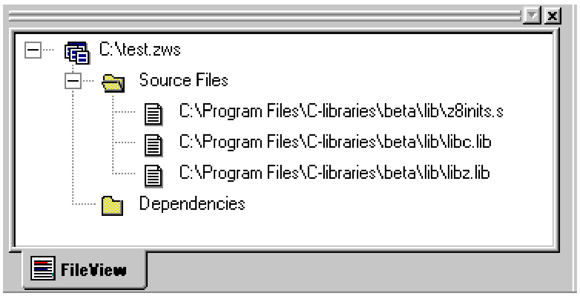
FIGURE 1-6. INSERT FILES INTO PROJECT DIALOG BOX

4. Browse to the directory where the C-Compiler was installed.
5. Select the Lib directory.
6. Select `all files` from the **Files of type** pop-up menu.

7. Hold the **Control key** and select the following files:
   - For Z8, select `Z8inits.s` *(C run time initialization file)*
   - For Z8Plus, select `Z8plusinit.s` *(C run time initialization file)*
   - To use dynamic frames for the Z8 or Z8Plus small memory model select `libz.lib` *(integer arithmetic library)*, `libc.lib` *(C library)* and `libfz.lib` *(floating point arithmetic library)*
   - To use static frames for the Z8 small memory model select `libz.lib` *(integer arithmetic library)*, `libcst.lib` *(C library)* and `libfzst.lib` *(floating point arithmetic library)*
   - To use the large memory model for the Z8 select `libz.lib` *(integer arithmetic library)*, `libcl.lib` *(C library)* and `libfzl.lib` *(floating point arithmetic library)*

NOTE: The large memory model is only available for the Z8. Static frames are only available for the Z8 small memory model.

8. Click Open. The files appear in the Project Viewer window. See Figure 1-7.



```
C:\test.zws
  Source Files
    C:\Program Files\C-libraries\beta\lib\z8inits.s
    C:\Program Files\C-libraries\beta\lib\libc.lib
    C:\Program Files\C-libraries\beta\lib\libz.lib
  Dependencies
```

FIGURE 1-7. PROJECT VIEWER WINDOW WITH FILES

NOTE: For Z8Plus, the `z8init.s` file is replaced by the `Z8plusinit.s` file.

9. Select Add to Project>New from the Project menu. The Insert New Files Into Project dialog box appears.
10. Select C Files from the Files of type pop-up menu.
11. Type a file name in the File Name field.
12. Click Open. The new file name appears in the Project Viewer window with a `.c` suffix, and a blank Edit window also appears.
13. Type the following code in the edit window:

```
#include <stdlib.h>
```

```
int randnum;
int main()
{
    srand(10);
   randnum=rand();
   randnum=rand();
}
```

14. Close and save the file.
15. Select **Update All Dependencies** from the **Build** menu. The Dependencies folder list in the Project Viewer window is updated.

Configure Settings

The Z8/Z8Plus C-Compiler can be configured through the Settings Option dialog box in ZDS. The Settings Option dialog box allows the user to configure:

- General options
- Optimization levels
- Preprocessor symbol definitions
- C-Compiler memory locations
- Section names

Perform the following steps to open the C-Compiler Settings Option dialog box:

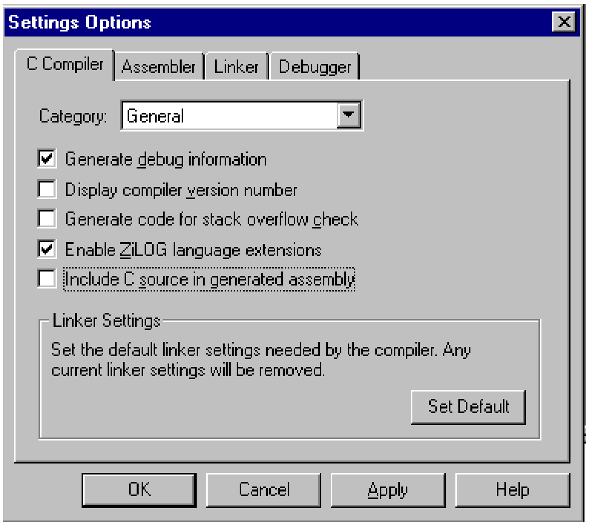
1. Open the project.
2. Select **Settings** from the **Project** menu. The Settings Options dialog box appears.
3. Click the **C-Compiler** tab. The C-Compiler Settings Option dialog box appears.

General Configuration

The C-Compiler General page allows you to enable or disable settings for the C-Compiler.

Perform the following steps to configure the General Page .

1. Select General from the Category pop-up list in the C-Compiler Settings dialog box. The C-Compiler General page appears. See Figure 1-8.
2. Click the Set Default button.
3. Click Apply.



- [✓] Generate debug information
- [ ] Display compiler version number
- [ ] Generate code for stack overflow check
- [✓] Enable ZiLOG language extensions
- [ ] Include C source in generated assembly

Linker Settings

Set the default linker settings needed by the compiler. Any current linker settings will be removed.

FIGURE 1-8. C-COMPILER GENERAL PAGE

The following options are available in the C-Compiler General page.

- The Generate debug information option generates symbolic debug information in the output object module. If a relocatable object file is being generated, symbols and other debugging information are embedded in the output relocatable object file. If this option is not checked, no symbolic debug information is generated. If this option is checked, optimizations are not performed *(default setting)*.
- The Display compiler version number option causes a two-line message to display in the Output window showing the copyright notice and version number.
- The Generate Code for stack overflow check option causes the C-Compiler to automatically write code that checks for memory stack overflows.
- The Enable ZiLOG language extensions option causes the C-Compiler to recognize // style comments *(default setting)*.
- The Include C Source in generated assembly option causes the C-source code to be included in the assembly file. This option only works when the Generate debug information option is selected.
- The Set Default button automatically configures the linker for use by the C-Compiler.

Warning Options

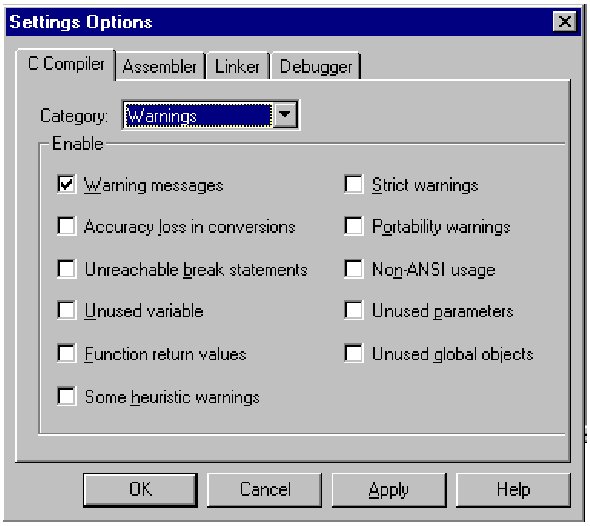
The C-Compiler Warning page allows you to select which C-Compiler warning messages are displayed in the ZDS output window.

Perform the following steps to configure the Warning Page.

1. Select **Warnings** from the **Category** pop-up list in the C-Compiler Settings dialog box. The C-Compiler Warning page appears. See Figure 1-9.
2. Select the warning messages to apply.

NOTE: The **Warning messages** option enables all the warning messages regardless of whether a warning is checked or not.

3. Click **Apply**.



Category: Warnings

Enable

- [✓] Warning messages
- [ ] Accuracy loss in conversions
- [ ] Unreachable break statements
- [ ] Unused variable
- [ ] Function return values
- [ ] Some heuristic warnings
- [ ] Strict warnings
- [ ] Portability warnings
- [ ] Non-ANSI usage
- [ ] Unused parameters
- [ ] Unused global objects

FIGURE 1-9. C-COMPILER WARNING PAGE

The following options are available in the C-compiler Warnings page.

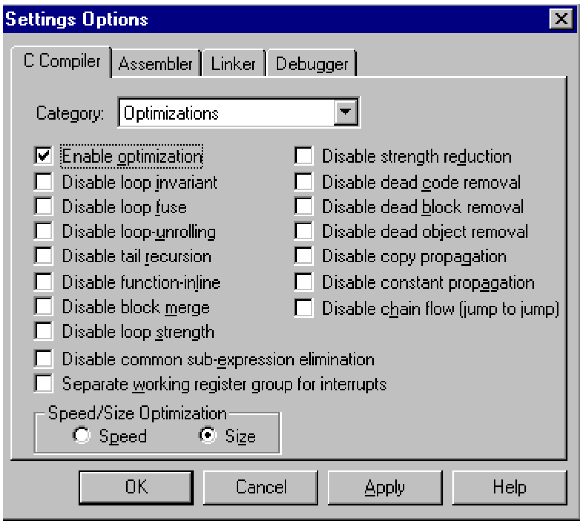
- Warning Messages - enables all warning messages
- Accuracy loss in conversions - enables portability warnings about accuracy loss in conversions
- Unreachable break statements - enables warnings about unreachable break statements
- Unused variable - enable warnings about variable usage, such as unused variable, defined but not used, etc.
- Function return values - enables warnings about function return values
- Some heuristic warnings - enables some heuristic warnings
- Strict warnings - enables strict warnings
- Portability warnings - enables portability warnings, and warnings about handling enumeration types
- Non-ANSI usage - enables warnings about non-ANSI usage
- Unused parameters - enables warnings about unused parameters (not included in –Wd)
- Unused global objects - enables warnings about unused global objects

Configuring Optimization Levels

The C-Compiler Optimizations page allows you to select an optimization level for the C-compiler. See Optimizing for Size and Speed on page 2-10 for a detailed description of the different optimizations.

Perform the following steps to configure the Optimizations page .

1. Select Optimizations from the Category pop-up list in the C-Compiler Settings Options dialog box. The Optimizations page appears. See Figure 1-10.
2. Select Enable optimization.
3. Select either Speed or Size in the Speed Size Optimization window. For a detailed description of speed or size optimizations see Optimizing for Size and Speed on page 2-10.
4. Click Apply.



Category: Optimizations

[✓] Enable optimization
[ ] Disable loop invariant
[ ] Disable loop fuse
[ ] Disable loop-unrolling
[ ] Disable tail recursion
[ ] Disable function-inline
[ ] Disable block merge
[ ] Disable loop strength
[ ] Disable strength reduction
[ ] Disable dead code removal
[ ] Disable dead block removal
[ ] Disable dead object removal
[ ] Disable copy propagation
[ ] Disable constant propagation
[ ] Disable chain flow (jump to jump)
[ ] Disable common sub-expression elimination
[ ] Separate working register group for interrupts

Speed/Size Optimization
Speed [✓ Size]

FIGURE 1-10. C-COMPILER OPTIMIZATIONS PAGE

NOTE: For more information on the optimization choices, see Optimization Descriptions on page 2-11.

Defining Preprocessor Symbols

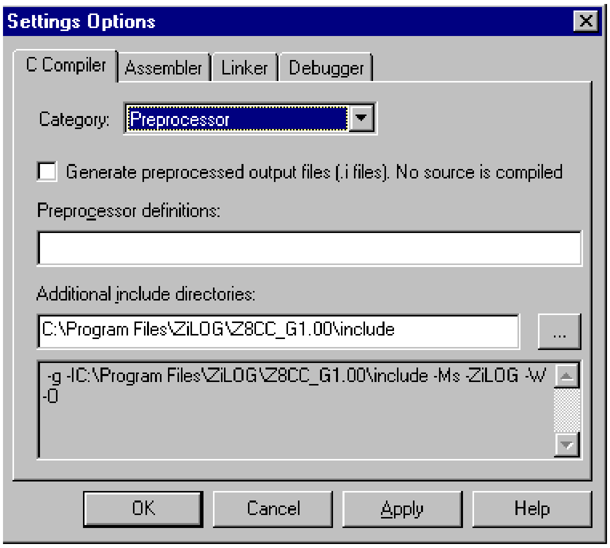
The C-Compiler Preprocessor page allows you to define preprocessor definitions, and specify additional search paths for included files.

Perform the following steps to configure the Optimizations page.

1. Select Preprocessor from the Category pop-up list in the C-Compiler Settings dialog box. The Preprocessor page appears. See Figure 1-11.
2. In the Additional Include Directories field enter the C-Compiler's installation path and `\INCLUDE`.

For example: If the compiler's installation path is `C:\PROGRAMS\Z8` enter `c:PROGRAMS\Z8\INCLUDE.`

3. Click Apply.



Category: Preprocessor

☐ Generate preprocessed output files (.i files). No source is compiled

Preprocessor definitions:

Additional include directories: C:\Program Files\ZiLOG\Z8CC_G1.00\include

```
-g -IC:\Program Files\ZiLOG\Z8CC_G1.00\include -Ms -ZiLOG -W
-O
```

FIGURE 1-11. C-COMPILER PREPROCESSOR PAGE

The Preprocessor page has the following fields and options:

- The Generate preprocessed output file option causes the compiler to only generate a viewable preprocessed file. An assembly or hex file is not generated when this option is checked.
- The Preprocessor definitions field is used to define the names of the symbols that are used by the preprocessor. Symbols may be defined with or without a value and successive symbols should be separated by a comma.

  EXAMPLE: `DEBUG,VERSION=3` defines the symbol DEBUG, but does not assign it a value. The statement also defines the symbol VERSION and assigns it a value of 3.

- The Additional Include Directories field is used to enter additional search paths the C-Compiler should use to locate included files. The search path can consist of directory names separated by semicolons.

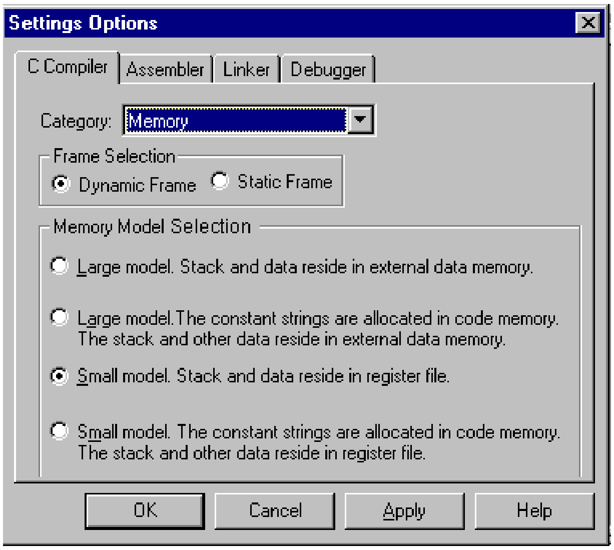
  EXAMPLE: `C:\PROGRAMS\ZDS\INCLUDE:LIB`

Memory Configuration

The C-Compiler Memory page allows the user to define where the stacks and data reside.

Perform the following steps to configure the Memory page.

1. Select Memory from the Category pop-up list in the C-Compiler Settings dialog box. The Memory page appears. See Figure 1-12.
2. Select the Small model. Stack and data reside in register file option.
3. Click Apply.



Category: Memory

Frame Selection
[✓ Dynamic Frame] Static Frame

Memory Model Selection
Large model. Stack and data reside in external data memory.
Large model.The constant strings are allocated in code memory. The stack and other data reside in external data memory.
[✓ Small model. Stack and data reside in register file.]
Small model. The constant strings are allocated in code memory. The stack and other data reside in register file.

FIGURE 1-12. C-COMPILER MEMORY PAGE

The following options are available in the C-Compiler memory page.

- Frame Selection
  - Dynamic Frame—Stack memory is allocated for frames during run-time.
  - Static Frame— Function frame are allocated in memory during the link process (for the Z8 small model only).
- Memory Model Selection window
  - Large model. Stack and data reside in external data memory.
  - Large model. The constant strings are allocated into code memory. The stack and other data reside in external data memory.
  - Small model. Stack and data reside in register file.
  - Small model. The constant strings are allocated in code memory. The stack and other data reside in register file.

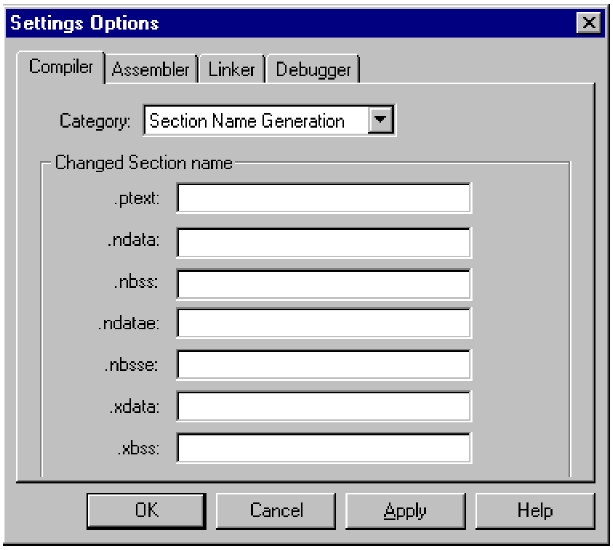
NOTE: For more information the difference between dynamic and static frames and the different memory models see Call Frames on page 4-1.

Rename a Section xName

The C-Compiler Section Name Generation page allows the user to rename section names.

Perform the following steps to rename a section name.

1. Select Section Name Generation from the Category pop-up list in the C-Compiler Settings dialog box. The Section Name Generation page appears. See Figure 1-13.
2. Enter the section's new name in the field next to the section's name. Sections can be renamed any valid segment name. For section name descriptions see Section Names on page 2-15.
3. Click Apply.



Category: Section Name Generation

.ptext:
.ndata:
.nbss:
.ndatae:
.nbsse:
.xdata:
.xbss:

FIGURE 1-13. SECTION NAME GENERATION PAGE

DOWNLOADING TO AN EMULATOR OR SIMULATOR

Before performing a debug session the user must compile the code and connect to the emulator. For more information on performing a debug session, see the ZDS Quick Start Guide or the ZDS on-line help.

Compile a Project

Perform the following steps to compile a project.

1. Open the previously project created.
2. In the Project Viewer window, double click on the C file that was created earlier in the session. The C file appears in the Edit window.
3. Select Build from the Build menu (the shortcut is F7) to compile, and link the files in the project. If an error occurs, double click on the error in the Output window .

NOTE: When building a project, ZDS only processes the files in the project that have changed since the last build. During a build, ZDS updates a dependency list for the project by adding each included filename to the project list.

CONNECT TO THE EMULATOR

Perform the following steps to connect to the emulator.

1. Select !Connect from the Project menu. The ZDS status bar shows that it’s connecting to the Emulator.
2. The message Emulator connected appears in the Output window Debug page.

NOTE: If an error message is received, ensure that both the target and emulator for the project are selected.

CHAPTER 2
C-COMPILER OVERVIEW

OVERVIEW

The Z8 C compiler is an optimizing compiler that translates standard ANSI C programs into ZiLOG assembly language source code. Key characteristics of the compiler are:

- **Supports ANSI C language** - ZiLOG's C-Compiler conforms to the ANSI C standard as defined by ANSI specifications for a free standing implementation.
- **Assembly output** - The compiler generates assembly language source files that can be viewed and modified.
- **Provides ANSI-standard run-time libraries** - A run-time library for each device is included with the compiler's tools. All library functions conform to the ANSI C library standard. These libraries include functions for string manipulation, buffer manipulation, data conversion, math, variable length argument lists.
- **COFF object files** - Common object file format (COFF) is used. This format allows the user to define the system's memory map at link time. This maximizes performance by linking C code and data objects into specific memory areas. Source-level debugging is also supported by the COFF file format.
- **Friendly assembly interface** - The compilers calling conventions are easy to use and flexible. These calling conventions allow the user to easily call assembly and C functions.
- **Preprocessor integration** - The compiler front end has a built in preprocessor for faster compilation.
- **Optimization levels** - The compiler allows the user to select optimization levels that employ advanced techniques for compacting and streamlining C code.
- **Language extensions** - Language extensions are provided to support processor specific features.
  - Memory spaces are supported through memory qualifiers.

   – Support for interrupt functions is also available.

- **Intrinsic functions** - Intrinsic functions are provided for inline assembly, setting interrupt vectors and enabling and disabling interrupts.

LANGUAGE EXTENSIONS

The Z8/Z8Plus family of processors are analog based processors that rely heavily on memory allocations. The C-language, without extensions, is only capable of placing data in one memory location. The C-Compiler memory extensions allow the user to assign data to either internal data memory (RFILE), ROM, or external data memory (XDATA)

ASSIGNING TYPES

Types are extended by adding memory qualifiers to the front of a statement. These memory qualifiers are defined with the key words

- **__R** assigns the type to the register file (RFILE) memory space. In a small memory model the default memory space for data is RFILE memory space.
- **__P** assigns the type to code memory space (ROM).
- **__M** assigns the type to XDATA. In a large model the default memory space for data is XDATA.

A derived type is not qualified by memory qualifiers (if any) of the type from where it was derived. Derived types can be structures, unions and function return types.

Example: `__R int num`

`num` is memory qualified by `__R` and is found in RFILE.

Example: `__P char arr [10]={2,4,6,8,10,12,14,16,18,20}`

`arr` is memory qualified by `__P and` all the elements of `arr` are found in ROM.

Example: `__M unsigned char val;`

`val` is memory qualified by `__M` and is found in XDATA.

Example: `__R char * ptr;`

`ptr` is a pointer to `char` in RFILE memory. The `Ptr` is not memory qualified but is a pointer to a qualified memory type.

Example: `Char *__R ptr;`

`ptr` is a pointer to char. The `ptr` is a memory qualified type and is found in RFILE.

Default Memory Qualifiers

Default memory qualifiers are applied if no memory qualifiers are specified. The default memory qualifier depends on the memory model chosen. See Table 2-1 for the default memory qualifiers for each model type.

Table 2-1. Default memory qualifiers

	Function	Globals	Locals	String	Parameters	Pointer
Small (s)	__ P	__R	__R	__R	__R	__R
Small (S)	__ P	__R	__R	__ P	__R	__R
Large (l)	__ P	__M	__M	__M	__M	__M
Large (L)	__ P	__M	__M	__ P	__M	__M

POINTERS

A pointer to a qualified memory type can not be converted to a different qualified memory type.

STRUCTURE AND UNION MEMBERS

The memory qualifier for a structure or union take precedence over an individual member's memory qualifier. No warning is issued when a member's memory qualifier does not match a structure or union's memory qualifier

Example:

```
struct{
__R char num;          /*__R memory qualifier is ignored*/
__R char * ptr;        /*ptr points to a char in RFILE.
                       Ptr itself is stored in __M memory*/
} __M mystruct;        /*all of mystruct is allocated to XDATA
                       (__M) memory.*/
```

SIZE OF POINTERS

The following is a list of pointers and their size:

- Pointer to __R is 1 byte
- Pointer to __M is 2 bytes
- Pointer to __P is 2 bytes

PRAGMAS

Three pragmas are supported by the compiler.

#pragma interrupt

Functions which are preceded with the prefix `#pragma interrupt` are assumed to be interrupt handlers. They should not take parameters or return values. The compiler saves all the registers used in the prologue and restores them in the epilogue of function codes for interrupts. When returning from interrupt functions, an `iret` instruction is generated instead of `ret,` which is used for normal functions.

NOTE: If the `-f` option is enabled, the interrupts have a separate working register group associated with them and the registers are not save or restored in the prologue or epilogue.

Example

```
#include <z8.h>
#define PORT0 (*(unsigned char *)0)
unsigned char delay;
#pragma interrupt
void timer(void)
{
     PORT0 = ~PORT0;
}
int main()
{
     PORT0 = 1;
     _setvector(IRQ1, timer);
}
```

#pragma at <address>

This pragma is used to declare variables at an address. In the generated assembly file the compiler emits directives to allocate the variable at a specified address. This pragma is applicable only for globals and statics. It is ignored if used with locals.

For example:

```
#pragma at 100

__R int var;
```

#pragma sfr <address>

This pragma is used to specify the address of a variable. The difference from `pragma at` is that the compiler does not perform allocations. This pragma is only applicable for uninitialized globals. It is ignored if used with locals or initialized globals.

For example:

```
#pragma sfr 100

__R int var;
```

USING THE DOS COMMAND LINE

The Z8 C-Compiler can be used on the DOS command line.

COMMAND LINE FORMAT

The syntax for the Z8 C-Compiler command line is as follows:

```
z8 [switches] … source …
```

COMMAND LINE SWITCHES

The following command-line switches (Table 2-2) are recognized.

TABLE 2-2. COMMAND LINE SWITCHES

Switch	Function
-c	Generate code for stack overflow check
-D <macro>	Define a preprocessor macro
-f	Separate working register group for interrupts
-g	Generate symbolic debug information
-I	Specify include path. This option may be repeated to specify multiple include paths
-Ml	Specify the model of compilation as large. In this model, stack and data reside in external data memory. This option is only for use with the Z8 .
-Ms	Specify the model of compilation as small. In this model, stack and data reside in register file. This is the default model.
-ML	Specify the model of compilation as large. The constant strings are allocated in code memory. The stack and other data reside in external data memory. This option is only for use with the Z8.
-MS	Specify the model of compilation as small. The constant strings are allocated in code memory. The stack and other data reside in register file
-Nbss <secname>	Change RFILE .bss section name to <secname> *(default is* `.nbss`*)*
-Nbsse <secname>	Change even aligned RFILE bss section name to <secname> *(default is* `.nbsse`*)*
-Ndata <secname>	Change RFILE data section name to <secname> *(default is* `.ndata`*)*

TABLE 2-2. COMMAND LINE SWITCHES

Switch	Function
-Ndatae <secname>	Change even aligned RFILE data section name to <secname> *(default is* `.ndatae`*)*
-Ntext <secname>	Change code section name to <secname> *(default is* `.ptext`*)*
-Nxdata <secname>	Change XDATA data section name to <secname> *(default is* `.xdata`*)*
-Nxbss <secname>	Change XDATA bss section name to <secname> *(default is* `.xbss`*)*
-o <name>	Specifies the output assembly file name
-O	Enable optimizations. Default is off
-Od loopinvariant	Disable loop invariant optimization
-Od tailrec	Disable tail recursion optimization
-Od loopunroll	Disable loop-unrolling optimization
-Od loopstrength	Disable loop strength optimization
-Od loopfuse	Disable loop fuse optimization
-Od inliner	Disable function-inline optimization
-Od deadobj	Disable dead object removal optimization
-Od deadblock	Disable dead block removal optimization
-Od deadcode	Disable dead code removal optimization
-Od blockmerge	Disable block merge optimization
-Od chainflow	Disable chain flow (jump to jump) optimization
-Od copyprop	Disable copy propagation optimization
-Od cse	Disable common sub-expression elimination optimization
-Od strength	Disable strength reduction optimization
-Od constprop	Disable constant propagation optimization
-P <filename>	Generate a preprocessed output to the file specified by <filename>
-speed	Optimize for speed, default is size
-src	Include C source in generated assembly when –g option is on
-static	Causse the compiler to use static call frames. This is supported for small models only and is ignored for large models. *The default is dynamic call frames*
-V	Display compiler version number

TABLE 2-2. COMMAND LINE SWITCHES

Switch	Function
-W	Enable warning messages
-Wa	Enable portability warnings about accuracy loss in conversions
-Wall	Equivalent to specifying all of the warning options
-Wansi	Enable warnings about non-ANSI usage
-Wb	Enable warnings about unreachable break statements
-Wd	Enable warnings about variable usage, such as unused variable, defined but not used, etc.
-Wf	Enable warnings about function return values
-Wh	Enable some heuristic warnings
-Wp	Enable portability warnings, and warnings about handling enumeration types
-Wstrict	Enable strict warnings
-Wv	Enable warnings about unused parameters (not included in –Wd)
-Wx	Enable warnings about unused global objects
-ZiLOG	Allow // style comments

NOTE: Other switches are for ZiLOG use only in this version.

COMMAND LINE EXAMPLES

Compiling

The command for Z8:

`Z8 test.c` generates `test.s`. By default the -Ms option is used.

Assembling

The command for Z8:

`zma -pz8 -J -otest.o test.s` generates `test.o`

The command for Z8Plus:

`zma -pz8E001 -J -otest.o test.s` generates `test.o`

Linking

The command for Z8:

`zld -mtest -otest` (compiler installation path)`\z8inits.o test.o` generates `test.ld` and `test.map`.

NOTE: z8inits.o is the run time initialization for a small model.

The command for Z8Plus:

`zld -mtest -otest` (compiler installation path)`\z8plusinit.o test.o` generates `test.ld` and `test.map`.

NOTE: z8plusinit.o is the run time initialization for a small model.

OPTIMIZING FOR SIZE AND SPEED

The Z8 C-compiler allows the user to optimize for speed and size.

When optimizing for size the following optimizations are performed:

- constant folding
- dead object removal
- simple jump optimization
- constant propagation
- copy propagation
- dead-code elimination
- common sub-expression elimination
- jump-to-jump optimization
- loop invariant code motion
- constant condition evaluation and other condition evaluation optimizations
- constant evaluation and expression simplification
- tail recursion
- loop fusion
- loop strength reduction

When optimizing for speed the following additional optimizations are performed:

- loop unroll

- function inligning

NOTE: A feature is provided to disable individual optimizations, which allows the user more control over optimizations.

DEBUGGING CODE AFTER OPTIMIZATION

Debugging of code should be complete before performing any level of optimization on the code. If the generate debug information is on no optimizations are performed, even if an optimization level is chosen. See General Configuration on page 1-14 for more information on enabling and disabling debug information.

OPTIMIZATION DESCRIPTIONS

The following is a description of the supported optimizations

Constant Folding

The compiler simplifies expressions by folding them into equivalent forms that require fewer instructions.

EXAMPLE: Before optimization: a=(b+2) +(c+3); After optimization: a=b+c+5

Dead Object Removal

Local and static variables that are declared but never used are removed

Simple Jump Optimization

Jump to next instruction is removed. Unreachable code is also removed.

Constant Propagation

Unaliased local variables are replaced by their assigned constant.

Copy Propagation

The compiler replaces references to the variable with its value. The value could be another variable, a constant, or a common sub-expression. This replacement increases the chances for constant folding, common sub-expression elimination, or total elimination of the variable.

Dead Code Elimination

Useless code is removed or changed. **For example:** assignments to local variables that are not used afterwards are removed.

Common Sub Expression Elimination

When the same value is produced by two or more expressions, the compiler computes the value once, saves it, and reuses it.

Jump to Jump Optimization

Targets in the control statement are replaced by the ultimate target.

Loop Invariant Code Motion

Expression within loops that compute the same value are identified and are replaced by a reference to a precomputed value.

Constant Condition Evaluation

The conditional expressions that are constant are computed at compile time.

Constant Evaluation and Expression Simplification

Replaces an expression by a simpler expression with the same semantics using constant folding, algebraic identities and tree transformations.

Tail Recursion

When a void result procedure ends with a call to itself, this optimization will replace the call with assignment of each actual parameter to the corresponding formal, and jump to the beginning of the procedure.

Function Inlining

This optimization replaces a call statement by a modified in-line copy of the callee. Only procedures that do not contain a call to another procedure and are not too large are selected for inlining.

Loop Fusion

Two adjacent loops are fused into one loop when they match a number of constraints. Both loops should have the same iteration count, and be data independent from each other. When the iteration counters are different objects, the optimization will tries to eliminate one.

Loop Strength

This optimization converts array indexing operations into pointer operations, for the sake of faster operation of loops.

Loop Unroll

This optimization tries to unroll a loop body, thus reducing loop control overhead, allowing more optimizations.

UNDERSTANDING ERRORS

The Z8 C-Compiler detects and reports errors in the source program. When an error is encountered, an error message is displayed in the ZDS Output window.

For example:

```
“ file.c”, line n: error message
```

ENABLING WARNING MESSAGES

Warning messages can be disabled or enabled through the command line. See Table 2-2 for more information on the various warnings that can be enabled.

INCLUDED FILES

A path to included files must be defined before the C-Compiler can recognize included files. An included files path is set in the Preprocessor page in the C-Compiler setting options dialog box. For more information on the Preprocessor page, see Defining Preprocessor Symbols on page 1-19. For command line version -I command line option can be used to specify the include path.

PREDEFINED NAMES

The Z8 C-Compiler comes with four predefined macro names. These names are:

- `_LINE_` Expands to the current line number
- `_FILE_` Expands to the current source filename
- `_DATE_` Expands to the compilation date in the form of *mm dd yy*
- `_TIME_` Expands to the compilation time in the form of *hh:mm:ss*

The following predefined names are only valid when the corresponding memory model is selected:

- `_SMALL_` is defined if the `-Ms` compilation option is selected (Small Model—Stack and data reside in register file)
- `_SMALLC_` is defined if the -MS compilation option is selected (Small Model—The constant strings are allocated in code memory, the stack and other data reside in register file)
- `_LARGE_` is defined if the `-Ml` compilation option is selected (Large model—Stack and data reside in external data memory)
- `_LARGEC_` is defined if the `-ML` compilation option is selected (Large model—The constant strings are allocated into code memory, the stack and other data reside in external data memory)

NOTE: For more information on using the command line see page 2-7. For more information on selecting a memory model using ZDS see page 1-21.

GENERATED ASSEMBLY FILE

After compiling a c-file an assembly file is generated and placed in the project directory. The assembly files are downloaded and linked and a COFF file is produced that is downloaded to the emulator. The user can modify the assembly in the ZDS Editor window.

To open and edit the assembly file:

1. Select Open File from the ZDS Edit menu. The Open file dialog box appears.
2. Select Assembler Files from the files of type pull down menu.
3. Browse to the project directory and double click on the file you want to open. The selected file appears in the ZDS edit window.

OBJECT SIZES

The following table lists basic objects and their size.

Type	Size
char	8 bits
short	16 bits
int	16 bits
long	32 bits
float	32 bits
double	32 bits
long double	32 bits

SECTION NAMES

The compiler places code and data into separate sections in the object file. Every section has a name that is used by the linker to determine which sections to combine and how sections are ultimately grouped in the executable file.

- Initialized Data Segment (.ndata) in RFILE
- Even aligned initialized data segment (.ndatae) in RFILE
- Uninitialized Data Segment (.nbss) in RFILE bss
- Even aligned uninitialized data segment (.nbsse) in RFILE bsse
- Code segment (.ptext)
- Working register group (.worksec) in RFILE
- Optional additional working register group (.worksec2) in RFILE for interrupts

Additional Z8 sections:

- Initialized external data segment (.xdata) in XDATA xdata
- Uninitialized external data segment (.xbss) in XDATA xbss

INCORPORATING ASSEMBLY WITH C

The Z8 C-Compiler allows the user to incorporate assembly code into their C code.

In the body of a function, use the `asm` statement. The syntax for the ASM statement is `_asm("<assembly line>");.`

- The contents of `<assembly line>` must be legal assembly syntax
- The only processing done on the `<assembly line>` is escape sequences
- Normal C escape sequences are translated

Example

```
#include <z8.h>

int main()

{

     _asm("\tclr\timr\n");

     return (0);

}
```

INCORPORATING C WITH ASSEMBLY

The C libraries that are provided with the compiler can also be used to add functionality to an assembly program. The user can create their own function or they can reference the library using the `ref` statement.

The following example shows an assembly file referencing the function `imul`. The `imul` function is defined in the C file.

NOTE: The C-Compiler precedes the use of globals with an underscore in the generated assembly.

Examples

Assembly file

```
     .def _main

     .def val1

     .def val2

     .def res

     .ref _imul
```

```
          define .ptext,space=ROM
          define .ndata,space=RFILE
          define .nbss,space=RFILE

          segment .ndata
     val1:dw 10
     val2:dw 20
     res:dw 0

          segment .ptext

     _main:
          ; save the registers if any in use
          ; in this example none

          push val2+1; push parameter 2
          push val2+0

          push val1+1; push parameter 1
          push val1+0

          call _imul; call the c functions
          add spl,#4

          ld res+0,r0 ; the result is in r0,r1
          ld res+1,r1 ; save the result
```

```
        ;restore the registers, if any were saved

        ld r0,#0; return 0
        ld r1,#0
        ret
```

Referenced C file

```
        typedef unsigned short uint16;
        typedef char int8;

    uint16 imul(uint16 x, uint16 y)
    {
        uint16 res;
        int8 i;

        res = 0;
        for (i=0; i < (int8)(16); i++)
        {
             if (y & 1)
             {
                     res += x;
             }
             x = x << 1;
             y = y >> 1;
        }
        return res;
    }
```

CHAPTER 3 LINKING FILES

INTRODUCTION

The purpose of the Zilog cross linker is to read relocatable object files and libraries and link them together to generate an executable load file. The file may then be loaded or written to a target system and debugged using ZDS. This chapter briefly describes the linker's inputs and outputs, and how the inputs to the linker are transformed into those outputs. See Figure 3-1.

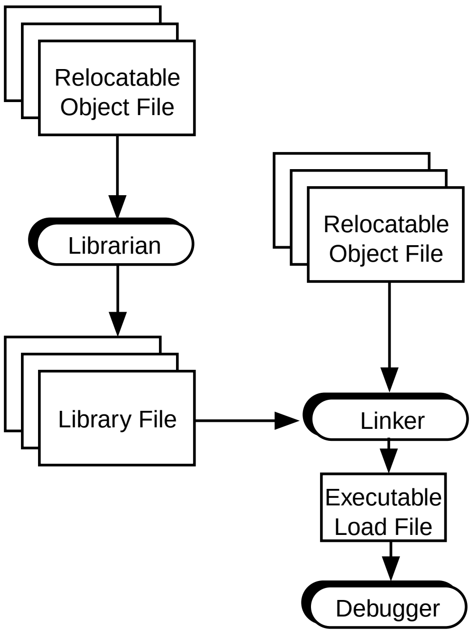


FIGURE 3-1. LINKER FUNCTIONAL RELATIONSHIP

WHAT THE LINKER DOES

The linker performs the following fundamental actions:

- Reads in Relocatable object modules and library files in Common Object File Format (COFF) or ZiLOG Object Module Format (ZOMF)
- Resolves external references
- Assigns absolute addresses to Relocatable sections
- Supports Source-Level Debugging (SLD)
- Generates a single executable module to download into the target system or burn into OTP or EPROM programmable devices
- Generates a map file
- Generates COFF files (for Libraries)

Linkage Editing

The linker creates a single executable load module from multiple relocatable objects.

Resolving External References

After reading multiple object modules, the linker searches through each of them to resolve external references to public symbols. The linker looks for the definition of public symbols corresponding to each external symbol in the object module.

Relocating Addresses

The linker allows the user to specify where the code and data are stored in the target processor system's memory at run-time. Changing relocation addresses within each section to an absolute address is handled in this phase.

Debugging Support

When the debug option is specified, the linker creates an executable file that can be loaded into the debugger at run-time. A warning message is generated if any of the object modules do not contain a special section that has debug symbols for the corresponding source module. Such a warning indicates that a source file was compiled or assembled without turning on a special switch that tells the compiler or assembler to include debug symbols information while creating a relocatable object module.

Creating Map Files

The linker can be directed to create a map file that details the location of the Relocatable sections and Public Symbols.

Outputting OMF Files

Depending upon the options specified by the user, the linker can produce two types of OMF files:

- Intel Hex Format Executable File
- COFF Format Executable File

USING THE LINKER WITH THE C-COMPILER

The linker is used to link compiled and assembled object files, C-Compiler libraries, user created libraries and C runtime initialization files. These files are linked according to the commands that are given in the linker command file. Once the files are linked an executable file in COFF (.ld) format is produced. The linker can also produce Intel hex (.hex, .dat) files, map files (.map) and symbol files (.sym) in ZiLOG symbol format.

The primary components of the linker are shown in Figure 3-2.

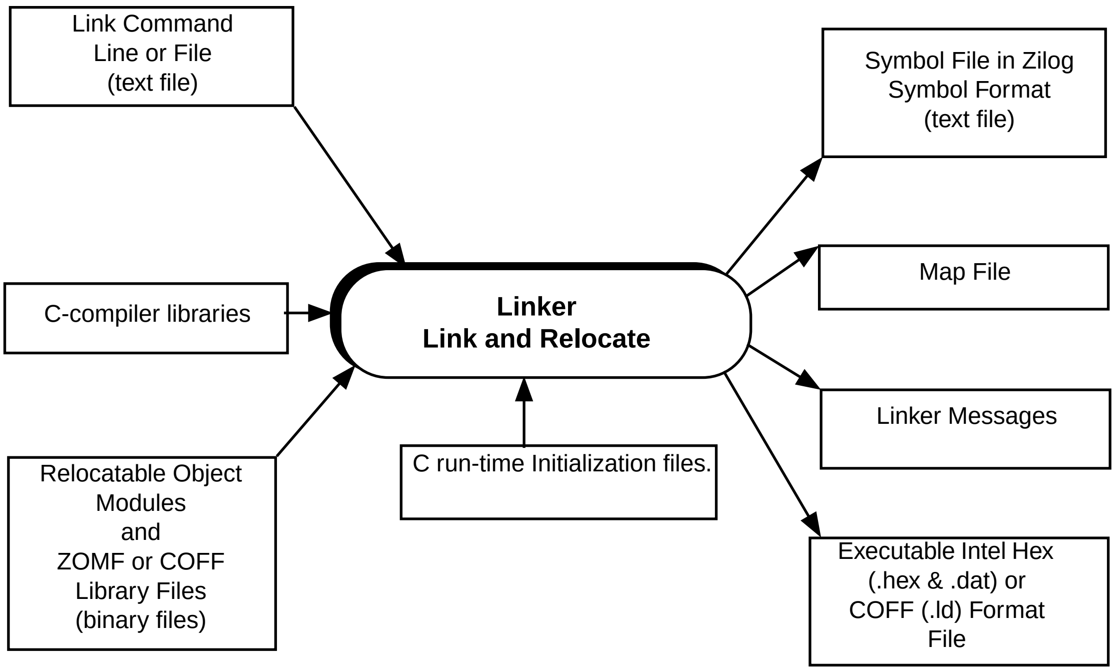

Link Command Line or File (text file)
Symbol File in Zilog Symbol Format (text file)
Map File
C-compiler libraries
Linker
Link and Relocate
Linker Messages
C run-time Initialization files.
Relocatable Object Modules and ZOMF or COFF Library Files (binary files)
Executable Intel Hex (.hex & .dat) or COFF (.ld) Format File

FIGURE 3-2. LINKER COMPONENTS

Run Time Initialization File

The C run-time initialization file is an assembly program that initializes memory before linking. This assembly program clears the .bss section, sets the register and stack pointers, and initializes the port mode resisters. Once these initializations are complete the program calls `main`, which is the C entry point.

Installed files

The following linker associated files in Table 3-1 are installed in the C-Compiler installation directory.

TABLE 3-1. LINKER REFERENCED FILES

File	Description
z8inits.s	Assembly source of example C startup module for z8 small model.
z8inits.o	Example C startup module for z8 small model.
z8initx.s	Assembly source of example C startup module for z8 large model.
z8initx.o	Example C startup module for z8 large model.
z8plusinit.s	Assembly source of example C startup module for z8 plus.
z8plusinit.o	Example C startup module for z8 plus.
z8s.lnk	Example linker command file for z8 small model.
z8x.lnk	Example linker command file for z8 large model.
z8p.lnk	Example linker command file for z8 plus.
libz.lib	Library helper functions for integer arithmetic.
libc.lib	Standard C library for small model.
libfz.lib	Library helper functions for floating point arithmetic (small model).
libcst.lib	Standard C library for small model, static frame.
libfzst.lib	Library helper functions for floating point arithmetic (small model, static frame)
libcl.lib	Standard C library for large model
libfzl.lib	Library helper functions for floating point arithmetic (large model).

NOTE: Source files for the run-time initialization files are provided in Appendix A, Initialization and Link Files.

INVOKING THE LINKER

The linker can be invoked either through ZDS or the DOS command line.

USING THE LINKER IN ZDS

The linker is automatically invoked when performing a build in ZDS. The following steps are performed when using the linker with ZDS.

1. ZDS calls the linker after compiling and assembling all the files.
2. All the object files and libraries that are include in the project are linked.
3. Error or warning messages that are generated by the linker are displayed in the ZDS output window.
4. If no errors are encountered the linker produces an executable file in either a COFF or HEX format. This executable file is placed in the project directory.

NOTE: Include the C-run time initialization file that is appropriate for the compilation model chosen in the project. See Table 3-1 for a list of initialization files that are included with the C-Compiler. For more information on adding included files see Create a File on page 1-10.

Configuring the Linker with ZDS

Perform the following steps to set the linker command file options in ZDS :

1. Open the project
2. Select Settings from the Project menu. The Settings Options dialog box appears.
3. Click the C-Compiler tab.
4. Select General from the Category pop-up list in the C-Compiler Settings dialog box. The C-Compiler General page appears.
5. Click the Set Default button.
6. Click Apply.

NOTE: The linker's settings can also be modified through the Linker Settings dialog box. Consult ZDS's on-line help for more information on configuring the linker.

Using the Linker with the Command Line

Use the syntax below to invoke the linker on the command line :

`zld -o` **output name** `-a` **init-object-files** {**object files**} **c-comp-lib-file lib-files map-file linker-command-file**

- `output-name` is the `.ld` filename. **For example**: If `test.ld` is the desired output file, then the output name should be `test`.
- `init.-object-file` is the C run time initialization file. For example: `z8inits.o` is for the Z8 small memory model, `z8initx.o` is for the Z8 large memory model and `Z8plusinit.o` is for the Z8Plus memory model. The user can specify their own initialization files to use. If the file is not in the current directory the path needs to be included in the file name.
- `{object files}` is the list of object files that are to be linked.
- `c-comp-lib-file` is the C-Compiler library files that need to be linked. See Table 3-1 for a list of library files that are include with the C-Compiler.
- `lib-files` is the library files created by the user using the ZDS archiver (ZAR).
- `map-file` is the map file's name that is to be generated by the linker.
- `linker-command-file` is the command file to be linked by the linker. Sample command files are provided in the lib directory. See Table 3-1 for a list of command files that are include with the C-Compiler.

Linker Command Line example

The following examples shows how to invoke the linker using the DOS command line.

For the Z8:

```
zld -otest -A lib-path\z8inits.o test.o lib-path\libz.lib lib-
path\libc.lib lib-path\libfz.lib\lib-path\z8s.link -mtest.map
```

For the Z8Plus:

```
zld -otest -A lib-path\z8plusinit.o test.o lib-path\libz.lib
lib-path\libc.lib lib-path\libfz.lib\lib-path\z8p.link -
mtest.map
```

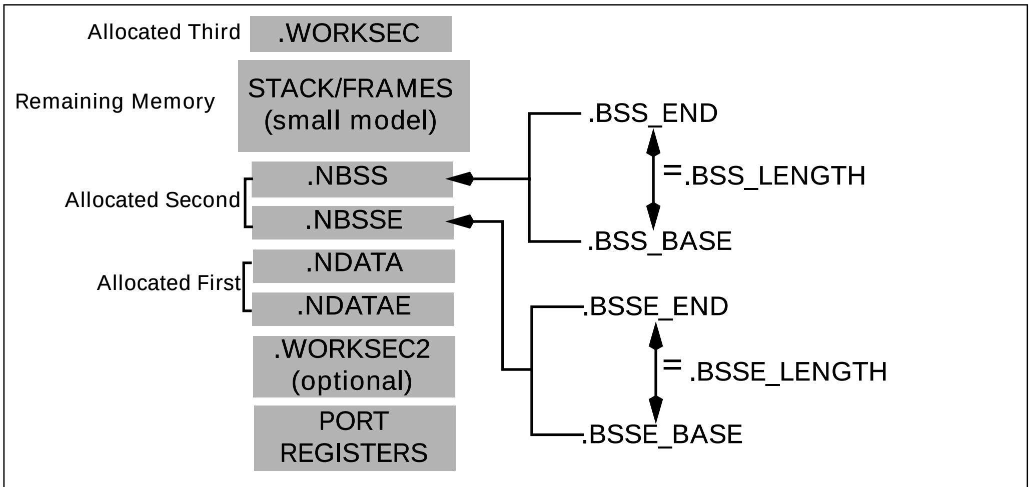
This example generates test.ld, test.hex, test.dat, test.sym and test.map as output. The lib-path is the (C-Compiler installation path)\ lib, and test.o is the object file corresponding to the C file created after compiling and assembling.

For more information on the linker command line see Linker Command Line on page 3-14.

MEMORY LAYOUT

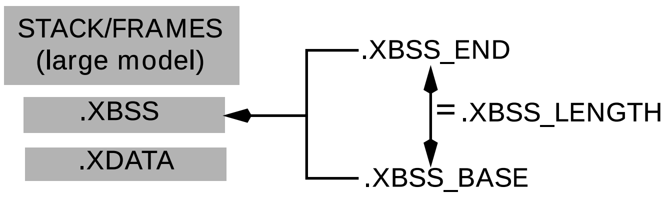
The linker allocates memory for sections per the linker command file. These sections need to be contiguous for the output section and cannot contain memory holes. The memory layout of RFILE and XDATA memory space is shown in Figure 3-3 and Figure 3-4. See Section Names on page 2-15 for a description of the sections.

NOTE: The STACK is in RFILE for the small model and is in XDATA for the large model .



NOTE: For large model the stack is moved from RFILE to the top of XDATA.

FIGURE 3-3. RFILE MEMORY LAYOUT



NOTE: This model is only for the Z8.For large model the stack is moved from RFILE to the top of XDATA.

FIGURE 3-4. XDATA MEMORY LAYOUT

LINKER SYMBOLS

The linker command file defines the symbols that are used by the C run-time initialization file to initialize the stack pointer, register pointer and clear the BSS section. Table 3-2 shows the symbols that are used by the linker.

TABLE 3-2. LINKER SYMBOLS

Symbol	Description
BSS_BASE	Base of .NBSS section
BSSE_BASE	Base of .NBSSE section
XBSS_BASE	Base of .XBSS section (*Z8 only)*
BSS_END	End of .NBSS section +1
BSSE_END	End of .NBSSE section +1
XBSS_END	End of .XBSS section +1 (*Z8 only)*
BSS_LENGTH	Length of .NBSS section
BSSE_LENGTH	Length of .NBSSE section
XBSS_LENGTH	Length of .XBSS section (*Z8 only)*
TOS	Top of stack
BOS	Bottom of stack

NOTE: Additional symbols are defined in the linker command file for copy table operations. See Appendix A, Initialization and Link Files for more information. Source files for the run-time initialization files are also provided in Appendix A

LINKER COMMAND FILE

The linker command file is text file that contains the linker command and options. The linker commands that can be used in the command file are summarized in Table 3-3. For linker options see Table 3-4.

TABLE 3-3. SUMMARY OF LINKER COMMANDS

Command	Description
Assign	Assigns a control section to an address space
Copy	Makes a copy of a control section
Define	Creates a public symbol at link-time; helps resolve an external symbol referenced at assembly time
Group	Creates a group of control sections that can be defined using the range command
Order	Specifies the ordering of specified control sections
Range	Sets a lower bound and an upper bound for an address space or a control section

NOTE: The linker commands are listed alphabetically in the table, for convenience; however, it is not required that commands be specified alphabetically in the command file. Command words and parameters other than those shown in the table are not legal. If any other word or parameter is used, an error message is written to the messages file, and the linker terminates without linking anything.

Linker Command ASSIGN

The ASSIGN command assigns a control section to an address space. This command is designed to be used in conjunction with the assembler's .SECT instruction.

Syntax: ASSIGN <*section*> <*address-space*>

The <*section*> must be a control section name, and the <*address-space*> must be an address space name.

Example: ASSIGN DSEG DATA

Linker Command COPY

This command makes a copy of a control section. The control section is loaded at the specified location, rather than at its linker-determined location. This command is designed to make a copy of an initialized RAM data section in a ROM address space, so that the RAM may be initialized from the ROM at run time.

Syntax: COPY <*section*> <*address-space*> [AT <*expression*>]

The <*section*> must be a control section name, and the <*address-space*> must be an address space name. The optional AT <*expression*> is used to copy the control section to a specific address in the target address space.

Example: `COPY bank1_data ROM or COPY bank1_data ROM at %1000`

Linker Command GROUP

This command allows the user to group control sections together and define the size of the grouped sections using the RANGE command.

Syntax: GROUP <*group name*> = <*section1*> <*section2*>]

The *group name* is the name of the grouped sections. The group name can not be the same name as an existing address space. *Section1* and *section2* are the sections assigned to the group. Sections within a group are allocated in the specified order.

NOTE: The new group's lower address location and size must be defined uisng the linker's RANGE command.

Example:

```
GROUP RAM =.data,.bss
RANGE RAM = 1000h:1FFFh
```

This example defines RAM as a block of memory in the range of `1000h` to `1FFFh`. The `.data` and `.bss` control sections are allocated to this block. The `.data` section is allocated at address `1000h` and the `.bss` section is allocated at the end of the `.data` section.

Linker Command ORDER

This command determines a sequence of linking.

Syntax: ORDER <*name1*> [,<*name2*> ...]

<*name*n> must be a control section name.

Example: `ORDER CODE1, CODE2`

Linker Command RANGE

This command sets the lower and upper limits of a control section or an address space. The linker issues a warning message if an address falls beyond the range declared with this command.

The linker provides multiple ways for the user to apply this command for a link session. Each separate case of the possible syntax is described below.

CASE 1

Syntax : RANGE <name> <expression> , <length> [, ...]

<*name*> may be a control section, or an address space. The first <*expression*> indicates the lower bound for the given address RANGE. The <*length*> is the length, in words, of the object.

Example: `RANGE ROM %700 , %100`

CASE 2

Syntax :RANGE <*name*> <*expression*> : <*expression*> [, ...]

<*name*> may be a control section or an address space. The first <*expression*> indicates the lower bound for the given address RANGE. The second <*expression*> is the upper bound for it.

Example: `RANGE ROM %17ff : %2000`

NOTE: Refer to the Expression Formats for the format of writing an expression.

Linker Command DEFINE

This command is used for a link-time creation of a user defined public symbol. It helps in resolving any external references (EXTERN) used in assembly time.

Syntax: DEFINE <*symbol name*> = <*expression*>

<*symbol name*> is the name of the public symbol. <*expression*> is the value of the public symbol.

Example: `DEFINE copy_size = copy top of usr_seg - copy base of usr_seg`

The “Expression Formats” section, which follows, explains different types of expressions that can be used.

LINKER COMMAND LINE

The syntax for the linker command line is:

`ZLD [<options>]<filename1> ...<filenamen>`.

- The "[ ]" enclosing the string "*options*" denotes that the options are not mandatory. In this document this convention will be continued for further discussion on linker's syntax and operations.
- The items enclosed in "`< >`" indicate the non-literal items.
- The "`. . .`" (ellipses) indicate that multiple tokens can be specified. These tokens are of the type of the non-literal specified in the syntax just prior to the ellipses.
- The syntax uses "`%`" prefix to a number to specify a hexadecimal numeric representation.
- The linker links the files listed in <*filename*> list. Each <*filename*> is the name of a COFF object file or library file, or the name of a text file containing linker commands and options.

COMMAND LINE SPECIFICATIONS

The following rules govern the command line specification:

- ZLD examines the named files' content to determine the file type (object, library, or command).
- The file names of the input files specified on the command line must be separated by spaces or tabs.
- The commands are not case sensitive; however, command line options and symbol names are case sensitive.
- The order of specifying options does not matter.
- The options must appear before the filenames.
- Specifying that input files use both command line and list creates a union of the two sets of inputs that is treated as input object and library files. The linker links the file twice, if the file names appear twice.
- During linking, the linker combines all object files in the order specified and resolves the external references. linker searches through the library files when it is unable to resolve references.
- A command file is a text file containing linker commands and options. Comments can be specified by use of the ";" character.
- If the linker is unable to open a named object file, library file, or a link command file, an error message is written to the standard error device, and the linker terminates without linking anything.
- If an unsupported OMF type of object file is included in the <*filename*> list, the linker displays an error message and terminates without linking anything.

LINKER COMMAND LINE OPTIONS

Linker options are specified by prefixing an option word with a minus (-). The linker options are summarized in Table 3-4 .

TABLE 3-4. SUMMARY OF LINKER OPTIONS

Options	Description
-?	Displays product logo, version number, and brief description of command line format and options.
-a	Generates an absolute object file in Intel Hex Format or Zilog Symbol Format.
-e <*entry*>	Specifies the program entry point. <entry> is any Public symbol.
-g	Generates symbolic debug information.
-m <*mapfile*>	Generates the map file.
-o <*objectfile*>	Generates the output file.
-q	Disables display of linker's copyright notice.
-r	Disables address range checking on relocatable expression. *This option should be used when linking compiler generated code*.
-W	Treats warnings as errors.
-w	Disables the generation of warning messages.

1. It is not required that options be specified alphabetically on the command line.
2. If any other option word is used, an error message is written to the messages file, and the linker terminates without linking anything.
3. All options must be preceded by a dash (-).

NOTE: For more information on the linker options refer to the ZDS On-line help.

Symbol File In Zilog Symbol Format

A symbol file in the Zilog symbol format is generated when the user specifies the absolute link mode (-a linker option). It is in the standard Zilog symbol format as shown inFigure 3-5, which follows. In each row, the first column lists the symbol name, second column lists the attribute of the symbol (“I” stands for internal symbol, “N” stands for local symbol, and “X” stands for public symbol), and the third column provides the value of the symbol expressed as four hexadecimal bytes.

```
_dgt_outbfr          I 0000800d

_digit_cntr          I 00008011

_dgt_inbfr           I 00008012

_led_refresh         I 000000b5

hex_reg              N 00008009

_bcd_hex_conv
                     I fffff7f5

_7conv_reg_4         N 00008009

_8conv_reg_3         N 0000800a
```

FIGURE 3-5. SAMPLE SYMBOL FILE

USING THE LIBRARIAN

The librarian allows the user to modify libraries and view the contents of individual library files.

The syntax for the librarian command line is as follows:

```
Zar [options] library [ member1 … membern ]
```

The librarian performs the operation specified in the options on the named library using the named member files. Libraries conventionally have an extension of `.lib` and library members have an extension of `.o`.

Command Line Options

Command line options are specified by prefixing an option letter with a minus (-). The command line options are summarized in Table 3-5.

.

Table 3-5. Summary of Library Options

Options	Description
-?	Requests a usage display.
-a	Appends the specified members to the library. This command does not replace an existing member that has the same name as an added member; it simply appends new members to the end of the library.
-d	Deletes the specified members from the library.
-q	Quiet mode: suppress display of the librarian copyright notice.
-r	Replaces the specified members in the library. If a specified member is not found in the library, the librarian adds it instead of replacing it.
-t	Prints a table of contents of the library. If you don't specify any member names, the librarian lists the names of all members of the library. If you specify any member names, only those members are listed.
-x	Extracts the specified members from the library. The librarian does not remove from the library those members that it extracts.

CHAPTER 4
RUN TIME ENVIRONMENT

FUNCTION CALLS

The C-compiler imposes a strict set of rules on function calls. Except for special runtime-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 program to fail.

FUNCTION CALL STEPS

A function performs the following tasks when it calls another function:

1. The caller saves the registers that are in use.
2. The caller pushes the arguments 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.
3. The caller calls the function.
4. When the called function is complete, the caller pops the arguments off the stack.

CALL FRAMES

The Z8/Z8plus C-Compiler supports both dynamic and static call frames.

By default the C-compiler uses dynamic call frames to allocate memory on the stack. To use static call frames, you need to specify the static call frame switch on the command line (see Command line switches on page 2-7) or select the Static Call frames option in ZDS.

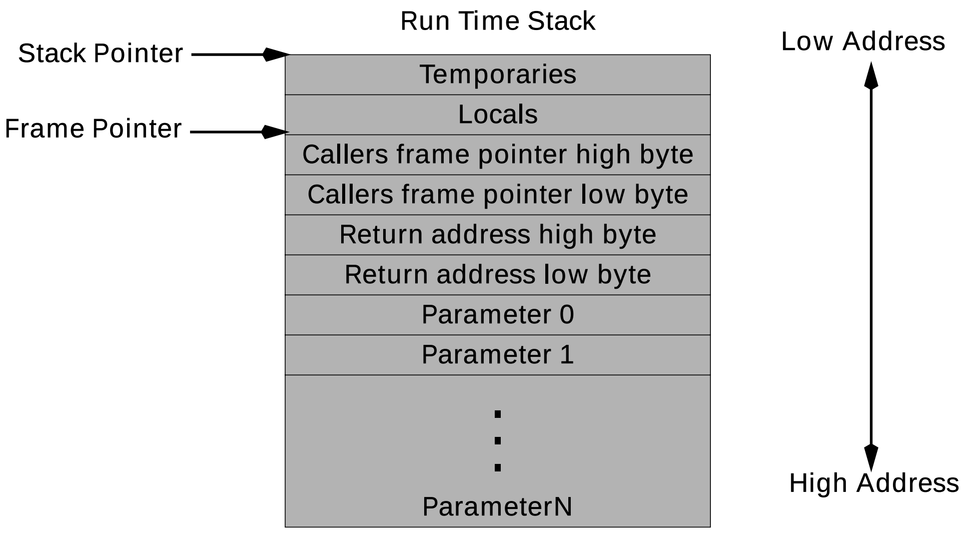
The following sections explain the difference between dynamic call frames and static call frames.

NOTE: Static call frames are only available for the Z8small model.

DYNAMIC CALL FRAMES

A dynamic call frame allocates memory during the run-time cycle of the program. As memory is requested it is allocated onto the stack. Once the function has returned, the memory that it was using is freed on the stack.

Dynamic call frames are perfect for programs that use recursion, or that call functions indirectly through pointers. For most processors the use of dynamic memory allocation is the most efficient way to partition and map call frames. However, processors that do not have a dedicated frame pointer register or have small amounts of memory, a static frame layout is ideal. Table 4-1 shows a diagram of a Z8 dynamic call frame layout.



NOTE: The frame layout for the small memory model uses only one byte for the frame pointer.

FIGURE 4-1. FRAME LAYOUT FOR THE LARGE MODEL

RESPONSIBILITIES OF A CALLED FUNCTION

A called function must perform the following tasks.

1. Push the frame pointer (r14, r15) for large model, and r15 for small model onto the stack.
2. Allocate the local frame.
3. Execute the code for the function.
4. If the function returns a scalar value, place it in the working registers r0-r3. A `char` value is returned in r0, int in r0,r1, long in r0,r1,r2,r3.
5. Deallocate the local frame.
6. Restore the caller's frame pointer.
7. Return.

SPECIAL CASES FOR A CALLED FUNCTION

The following exceptions apply to special cases for called functions.

Returning a structure

If the function returns a structure, the caller allocates the space for the structure on top of the stack. The size of the space allocated is the size of structure plus two additional bytes. To return a structure, the called function then copies the structure to the space allocated by the caller.

Not allocating a local frame

If there are no local variables, no arguments, no use of temporary locations, the code is not being compiled to run under the debugger and the function does not return a structure, there is no need to allocate a stack frame.

STATIC CALL FRAMES

A static frame layout should be used when the physical memory size of the processor is limited or the chip does not have a built in frame-pointer register. Function frames used in a static frame layout are allocated in memory during link time. A memory map of these functions is created and is used by the compiler to assign a frame's address during runtime. Instead of assigning a dynamic address to the function frame a static address is used that was decided by the linker during the link process.

The linker maps frame addresses based on a function call graph that identifies memory resources that are shared at the same time. Disadvantage of static frames is that a function cannot be called through a pointer due to the static assignment of function addresses.

Static call frames are supported in small model only. When the compiler is invoked with `-static` option static call frames are generated. The structure of a static call frame is as given below:

For the non static function `fun`:

Local are allocated in a frame labeled `.L_fun,` see Figure 4-2. All the locals are at a positive offset from this label.

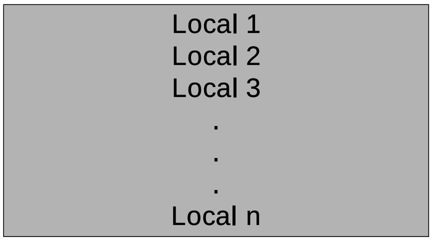


FIGURE 4-2. .L_FUN FRAME LAYOUT

Parameters are allocated in a frame labeled `.P_fun`. See Figure 4-3. All the parameters are at positive offset from this label.

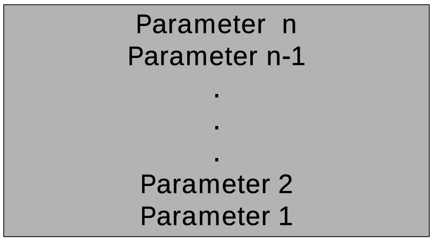


FIGURE 4-3. .P_FUN FRAME LAYOUT

For static function the frame labels also include the base of filename. For example for static function sfun in `prog.c`, the local frame label will be `.Lprog_sfun`, and the parameter frame label will be `.Pprog_sfun`.

USING THE RUN-TIME LIBRARY

The C-Compiler provides a collection of run-time libraries that can be easily referenced and incorporated into your code. The following sections describe the use and format of run-time libraries. Each library function is declared in a supplied header file. These header files can be included in C programs using the `#include` preprocessor directive. See Defining Preprocessor Symbols on page 1-19 for more information on including header files.

Each header file contains declarations for a set of related functions plus any necessary types and additional macros. See Table 4-1 for a description of each header file that is include with the C-Compiler.

The header files are installed in the include directory of the compiler installation path. The library files are installed in the lib directory of the compiler installation path.

NOTE: Two sets of libraries are included, one for small memory model (libc.lib) and one for large memory model (libcl.lib).

INSTALLED FILES

The following header files are installed in the C-Compiler installation directory.

TABLE 4-1. INSTALLED LIBRARY FILES

File	Description
asset.h	Asserts
ctype.h	Character handling functions
errno.h	Errors

TABLE 4-1. INSTALLED LIBRARY FILES (CONTINUED)

File	Description
float.h	Floating point limits
limits.h	Interger limits
math.h	Math functions
stdarg.h	Variable argument macros
stddef.h	Standard defines
stdio.h	Standard types and defines
stdlib.h	General utility functions
string.h	String handling functions
z8.h	Z8 specific function and defines
z8plus.h	Z8plus specific function and defines

LIBRARY FUNCTIONS

Run-time library routines are provided for the following:

- Buffer Manipulation
- Character Classification and Conversion
- Data Conversion
- Math
- Searching and Sorting
- String Manipulation
- Variable-Length Argument Lists
- Intrinsic functions

_asm FUNCTION

Header file statement: #include <z8.h>

Syntax: `_asm` ("assembly language instruction")

The _asm pseudo-function emits the specified assembly language instruction to the compiler-generated assembly file. The _asm pseudo-function accepts a single parameter, which must be a string literal.

Return Value

There is no return value.

abs FUNCTION

Header file statement: #include<stdlib.h>

Syntax: `int abs(` int n);

Parameter	Description
n	Integer Value

The abs function returns the absolute value of its integer parameter n.

Return Value

The abs function returns the absolute value of its parameter. There is no error return.

atof, atoi, atol FUNCTIONS

Header file statement: #include <stdlib.h>

Syntax: `double atof` (const char *string);

`int atoi`(const char *string);

`long atol`(const char *string);

Parameter	**Description**
string	String to be converted

These functions convert a character string to a double-precision floating-point value (atof), an integer value (atoi), or a long integer value (atol). The input string is a sequence of characters that can be interpreted as a numerical value of the specified type.

The function stops reading the input string at the first character that it cannot recognize as part of a number. This character may be the null character ('\0') terminating the string.

The atof function expects string to have the following form:

[whitespace] [sign] [digits] [.digits] [{d | D | e | E }[sign]digits]

A whitespace consists of space and/or tab characters, which are ignored; sign is either plus (+) or minus (-); and digits are one or more decimal digits. If no digits appear before the decimal point, at least one must appear after the decimal point. The decimal digits may be followed by an exponent, which consists of an introductory letter (d, D, e, or E) and an optionally signed decimal integer.

The atoi and atol functions do not recognize decimal points or exponents. The string argument for these functions has the form

[whitespace] [sign]digits

where whitespace, sign, and digits are exactly as described above for atof.

Return Value

Each function returns the double, int, or long value produced by interpreting the input characters as a number. The return value is 0 (for atoi), 0L (for atol), and 0.0 (for atof) if the input cannot be converted to a value of that type.

- The return value is undefined in case of overflow.

_di FUNCTION

Header file statement: #include <z8.h>

Syntax: `void _di`(void);

The _di routine disables interrupts by executing a z8 di machine instruction. This is an intrinsic function.

Return Value

There is no return value.

div FUNCTION

Header file statement: #include <stdlib.h>

Syntax: `div_t div`(int num, int denom);

Parameter	Description
numer	Numerator
denom	Denominator

The div function divides numer by denom, computing the quotient and the remainder. The div_t structure contains the following elements:

Element	Description
int quot	Quotient
int rem	Remainder

The sign of the quotient is the same as that of the mathematical quotient. Its absolute value is the largest integer that is less than the absolute value of the mathematical quotient. If the denominator is 0, the behavior is undefined.

Return Value

The div function returns a structure of type div_t, comprising both the quotient and the remainder. The structure is defined in the `stdlib.h` header file.

_ei FUNCTION

Header file statement: #include <z8.h>

Syntax: `void _ei( void );`

The _ei routine enables interrupts by executing a Z8 ei machine instruction. This is an intrinsic function.

Return Value

There is no return value.

is FUNCTIONS

Header file statement: #include <ctype.h>

Syntax:

```
int isalnum( int c );
int isalpha( int c );
int iscntrl( int c );
int isdigit( int c );
int isgraph( int c );
int islower( int c );
int isprint( int c );
int ispunct( int c );
int isspace( int c );
int isupper( int c );
int isxdigit( int c );
```

Parameter	Description
`c`	Interger to be tested

Each function in the `is` family tests a given integer value, returning a nonzero value if the integer satisfies the test condition and 0 if it does not. The ASCII character set is assumed.

The `is` functions and their test conditions are listed below:

Function	Test Condition
isalnum	Alphanumeric (‘A’-‘Z’, ‘a’-‘z’, or ‘0’-‘9’)
isalpha	Letter (‘A’-‘Z’ or ‘a’-‘z’)
iscntrl	Control character (0x00 - 0x1F or 0x7F)
isdigit	Digit (‘0’-‘9’)
isgraph	Printable character except space (‘ ‘)
islower	Lowercase letter (‘a’-‘z’)
isprint	Printable character (0x20 - 0x7E)
ispunct	Punctuation character
isspace	White-space character (0x09 - 0x0D or 0x20)
isupper	Uppercase letter (‘A’-‘Z’)
isxdigit	Hexadecimal digit (‘A’-‘F’,’a’-‘f’, or ‘0’-‘9’)

Return Value

These routines return a nonzero value if the integer satisfies the test condition and 0 if it does not.

labs FUNCTION

Header file statement: #include <stdlib.h>

Syntax: `long labs`(long n);

Parameter	Description
n	Long-integer value

The labs function produces the absolute value of its long-integer argument n.

Return Value

The labs function returns the absolute value of its argument. There is no error returned.

memchr FUNCTION

Header file statement: #include <string.h>

Syntax: `void *memchr(` const void *buf, int c, size_t count)

Parameter	Description
buf	Pointer to buffer
c	Character to look for
count	Number of characters

The memchr function looks for the first occurrence of c in the first count bytes of `buf`. It stops when it finds `c` or when it has checked the first count bytes.

Return Value

If successful, memchr returns a pointer to the first location of c in buf. Otherwise, it returns NULL.

memcmp FUNCTION

Header file statement: #include <string.h>

Syntax: `int memcmp` (const void *buf1, const void *buf2, size_t count)

Parameter	Description
buf1	First buffer
buf2	Second buffer
count	Number of characters

The memcmp function compares the first count bytes of buf1 and buf2 and returns a value indicating their relationship, as follows:

Value	Meaning
< 0	buf1 less than buf2
= 0	buf1 identical to buf2
> 0	buf1 greater than buf2

Return Value

The memcmp function returns an integer value, as described above.

memcpy FUNCTION

Header file statement: #include <string.h>

Syntax: `void *memcpy` (void *dest, const void *src, size_t count)

Parameter	Description
dest	New buffer
src	Buffer to copy from
count	Number of characters to copy

The memcpy function copies count bytes of src to dest. If the source and destination overlap, these functions do not ensure that the original source bytes in the overlapping region are copied before being overwritten. Use `memmove` to handle overlapping regions.

Return Value

The `memcpy` function returns the value of `dest`.

memmove FUNCTION

Header file statement: #include <string.h>

Syntax: `void *memmove` (void *dest, const void *src, size_t count)

Parameter	Description
dest	Destination object
src	Source object
count	Number of characters to copy

The `memmove` function copies count characters from the source (src) to the destination (dest). If some regions of the source area and the destination overlap, the `memmove` function ensures that the original source bytes in the overlapping region are copied before being overwritten.

Return Value

The `memmove` function returns the value of `dest`.

memset FUNCTION

Header file statement: #include <string.h>

Syntax: `void *memset` (void *dest, int c, size_t count)

Parameter	Description
dest	Pointer to destination
c	Character to set
count	Number of characters

The memset function sets the first count bytes of dest to the character `c`.

Return Value

The `memset` function returns the value of `dest`.

rand FUNCTION

Header file statement: #include <stdlib.h>

Syntax: `int rand` (void);

The rand function returns a pseudorandom integer in the range 0 to RAND_MAX. The srand routine can be used to seed the pseudorandom-number generator before calling rand.

Return Value

The rand function returns a pseudorandom number, as described above. There is no error returned.

_setvector FUNCTION

Header file statement: #include <z8.h>

Syntax: `void _setvector` (int idVector, void (*handler)(void))

Parameter	Description
idVector	Vector number to set
handler	Interrupt handler function

The _setvector sets the interrupt vector of the CPU to the address of the specified handler. This is an intrinsic function. The following manifest constants are defined by <z8.h> and should be used to specify the vector whose value is to be set.

Constant	Description
IRQ0	Interrupt 0 vector.
IRQ1	Interrupt 1 vector.
IRQ2	Interrupt 2 vector.
IRQ3	Interrupt 3 vector.
IRQ4	Interrupt 4 vector.
IRQ5	Interrupt 5 vector.
RESET	Reset vector.

Return Value

There is no return value.

srand FUNCTION

Header file statement: #include <stdlib.h>

Syntax: `void srand(` unsigned int seed);

Parameter	Description
seed	Seed for random-number generation

The srand function sets the starting point for generating a series of pseudorandom integers. To reinitialize the generator, use 1 as the seed argument. Any other value for seed sets the generator to a random starting point.

The rand function is used to retrieve the pseudorandom numbers that are generated. Calling rand before any call to srand generates the same sequence as calling srand with seed passed as 1.

Return Value

There is no return value.

strcat FUNCTION

Header file statement: #include <string.h>

Syntax: `char *strcat` (char *string1, const char *string2);

Parameter	Description
string1	Destination string
string2	Source string

The strcat function appends string2 to string1, terminates the resulting string with a null character, and returns a pointer to the concatenated string (string1).

The strcat function operates on null-terminated strings. The string arguments to this function are expected to contain a null character (‘\0’) marking the end of the string. No overflow checking is performed when strings are copied or appended.

Return Value

The return values for this function are described above.

strchr FUNCTION

Header file statement: #include <string.h>

Syntax: `char *strchr` (const char *string, int c);

Parameter	Description
string	Source string
c	Character to be located

The strchr function returns a pointer to the first occurrence of c (converted to char) in string. The converted character c may be the null character ('\0'); the terminating null character of string is included in the search. The function returns NULL if the character is not found.

The strchr function operates on null-terminated strings. The string arguments to these functions are expected to contain a null character ('\0') marking the end of the string.

Return Value

The return values for this function are described above.

strcmp FUNCTION

Header file statement: #include <string.h>

Syntax: `int strcmp` (const char *string1, const char *string2);

Parameter	Description
`string1`	String to compare
`string2`	String to compare

The strcmp function compares string1 and string2 lexicographically and returns a value indicating their relationship, as follows:

Value	Meaning
< 0	string1 less than string2
= 0	string1 identical to string2
> 0	string1 greater than string2

The strcmp function operates on null-terminated strings. The string arguments to these functions are expected to contain a null character (‘\0’) marking the end of the string.

Note that two strings containing characters located between ‘Z’ and ‘a’ in the ASCII table (‘[’, ‘\’_’]’, ‘^’, ‘_’, and ‘‘’) compare differently depending on their case. For example, the two strings, "ABCDE" and "ABCD^", compare one way if the comparison is lowercase ("abcde" > "abcd^") and compare the other way ("ABCDE" < "ABCD^") if it is uppercase.

Return Value

The return values for this functions are described above.

strcpy FUNCTION

Header file statement: #include <string.h>

Syntax: `char *strcpy` (char *string1, const char *string2);

Parameter	Description
string1	Destination string
string2	Source string

The strcpy function copies string2, including the terminating null character, to the location specified by string1, and returns string1.

The strcpy function operates on null-terminated strings. The string arguments to this function are expected to contain a null character ('\0') marking the end of the string. No overflow checking is performed when strings are copied or appended.

Return Value

The return values for this function are described above.

strcspn FUNCTION

Header file statement: #include <string.h>

Syntax: `size_t strcspn` (const char *string1, const char *string2);

Parameter	Description
string1	Source string
string2	Character set

The strcspn functions return the index of the first character in string1 belonging to the set of characters specified by string2. This value is equivalent to the length of the initial substring of string1 consisting entirely of characters not in string2. Terminating null characters are not considered in the search. If string1 begins with a character from string2, strcspn returns 0.

The strcspn function operates on null-terminated strings. The string arguments to these functions are expected to contain a null character ('\0') marking the end of the string.

Return Value

The return values for this function are described above.

strlen FUNCTION

Header file statement: #include <string.h>

Syntax: `size_t strlen` (const char *string);

Parameter	Description
string	Null-terminated string

The strlen function returns the length, in bytes, of string, not including the terminating null character ('\0').

Return Value

This function returns the string length. There is no error returned.

strncat FUNCTION

Header file statement: #include <string.h>

Syntax: `char *strncat` (char *string1, const char *string2, size_t count);

Parameter	Description
string1	Destination string
string2	Source string
count	Number of characters appended

The strncat function appends, at most, the first count characters of string2 to string1, terminate the resulting string with a null character ('\0'), and return a pointer to the concatenated string (string1). If count is greater than the length of string2, the length of string2 is used in place of count.

Return Value

The return values for these functions are described above.

strncmp FUNCTION

Header file statement: #include <string.h>

Syntax: `int strncmp` (const char *string1, const char *string2, size_t count);

Parameter	Description
string1	String to compare
string2	String to compare
count	Number of characters compared

The strncmp function lexicographically compares, at most, the first count characters of string1 and string2 and return a value indicating the relationship between the substrings, as listed below:

Value	Meaning
< 0	string1 less than string2
= 0	string1 identical to string2
> 0	string1 greater than string2

Return Value

The return values for this function are described above.

strncpy FUNCTION

Header file statement: #include <string.h>

Syntax: `char *strncpy` (char *string1, const char *string2, size_t count);

Parameter	Description
string1	Destination string
string2	Source string
count	Number of characters copied

The strncpy function copies count characters of string2 to string1 and return string1. If count is less than the length of string2, a null character ('\0') is not appended automatically to the copied string. If count is greater than the length of string2, the string1 result is padded with null characters ('\0') up to length count.

Note that the behavior of strncpy is undefined if the address ranges of the source and destination strings overlap.

Return Value

The return values for this function are described above.

strpbrk FUNCTION

Header file statement: #include <string.h>

Syntax: `char *strpbrk` (const char *string1, const char *string2);

Parameter	Description
string1	Source string
string2	Character set

The strpbrk function finds the first occurrence in string1 of any character from string2. The terminating null character ('\0') is not included in the search.

Return Value

This function returns a pointer to the first occurrence of any character from string2 in string1. A NULL return value indicates that the two string arguments have no characters in common.

strrchr FUNCTION

Header file statement: #include <string.h>

Syntax: `char *strrchr` (const char *string, int c);

Parameter	Description
string	Searched string
c	Character to be located

The strrchr function finds the last occurrence of c (converted to char) in string. The string's terminating null character ('\0') is included in the search. (Use strchr to find the first occurrence of c in string.)

Return Value

This function returns a pointer to the last occurrence of the character in the string. A NULL pointer is returned if the given character is not found.

strspn FUNCTION

Header file statement: #include <string.h>

Syntax: `size_t strspn`(const char *string1, const char *string2);

Parameter	Description
string1	Searched string
string2	Character set

The strspn function returns the index of the first character in string1 that does not belong to the set of characters specified by string2. This value is equivalent to the length of the initial substring of string1 that consists entirely of characters from string2 . The null character ('\0') terminating string2 is not considered in the matching process. If string1 begins with a character not in string2, strspn returns 0.

Return Value

This function returns an integer value specifying the length of the segment in string1 consisting entirely of characters in string2.

strstr FUNCTION

Header file statement: #include <string.h>

Syntax: `char *strstr(` const char *string1, const char *string2)

Parameter	Description
string1	Searched string
string2	String to search for

The strstr function returns a pointer to the first occurrence of string2 in string1.

Return Value

This function returns either a pointer to the first occurrence of string2 in string1, or NULL if it does not find the string.

strtok FUNCTION

Header file statement: #include <string.h>

Syntax: `char *strtok` (char *string1, const char *string2)

Parameter	Description
string1	String containing token(s)
string2	Set of delimiter characters

The strtok function reads string1 as a series of zero or more tokens and string2 as the set of characters serving as delimiter of the tokens in string1. The tokens in string1 may be separated by one or more of the delimiters from string2.

The tokens can be broken out of string1 by a series of calls to strtok. In the first call to strtok for string1, strtok searches for the first token in string1, skipping leading delimiters. A pointer to the first token is returned. To read the next token from string1, call strtok with a NULL value for the string1 argument. The NULL string1 argument causes strtok to search for the next token in the previous token string. The set of delimiters may vary from call to call, so string2 can take any value.

Note that calls to this function will modify string1, because each time strtok is called it inserts a null character (‘\0’) after the token in string1.

Return Value

The first time strtok is called, it returns a pointer to the first token in string1. In later calls with the same token string, strtok returns a pointer to the next token in the string. A NULL pointer is returned when there are no more tokens. All tokens are null-terminated.

strtod, strtol, strtoul FUNCTIONS

Header file statement: #include <stdlib.h>

Syntax: `double strtod`(const char *nptr, char **endptr);

`long strtol`(const char *nptr, char **endptr, int base);

`unsigned long strtoul`(const char *nptr, char **endptr, int base)

Parameter	**Description**
nptr	String to convert
endptr	Pointer to character that stops scan
base	Number base to use

The strtod, strtol, and strtoul functions convert a character string to a double-precision value, a long-integer value, or an unsigned long-integer value, respectively. The input string is a sequence of characters that can be interpreted as a numerical value of the specified type.

These functions stop reading the string at the first character they cannot recognize as part of a number. This may be the null character (‘\0’) at the end of the string. With strtol or strtoul, this terminating character can also be the first numeric character greater than or equal to base. If endptr is not NULL, a pointer to the character that stopped the scan is stored at the location pointed to by endptr. If no conversion could be performed (no valid digits were found or an invalid base was specified), the value of nptr is stored at the location pointed to by endptr.

The strtod function expects nptr to point to a string with the following form:

[whitespace] [sign] [digits] [.digits] [{d | D | e | E}[sign]digits]

A whitespace consists of space and tab characters, which are ignored; sign is either plus (+) or minus (-); and digits are one or more decimal digits. If no digits appear before the decimal point, at least one must appear after the decimal point. The decimal digits can be followed by an exponent, consisting of an introductory letter (b, D, e, or E) and an optionally signed decimal integer.

The first character that does not fit this form stops the scan.

The strtol and strtoul functions expect nptr to point to a string with the following form:

[whitespace] [{ + | -}] [0 [{ x | X }]] [digits]

If base is between 2 and 36, then it is used as the base of the number. If base is 0, the initial characters of the string pointed to by nptr are used to determine the base. If the first character is 0 and the second character is not ‘x’ or ‘X’, then the string is interpreted as an octal integer; otherwise, it is interpreted as a decimal number. If the first character is ‘0’ and the second character is ‘x’ or ‘X’, then the string is interpreted as a hexadecimal integer. If the first character is ‘1’ through ‘9’, then the string is interpreted as a decimal integer. The letters ‘a’ through ‘z’ (or ‘A’ through ‘Z’) are assigned the values 10 through 35; only letters whose assigned values are less than base are permitted.

The strtoul function allows a plus (+) or minus (-) sign prefix; a leading minus sign indicates that the return value is negated.

Return Value

The strtod function returns the value of the floating-point number, except when the representation would cause an overflow, in which case they return +/-HUGE_VAL. The functions return 0 if no conversion could be performed or an underflow occurred.

The strtol function returns the value represented in the string, except when the representation would cause an overflow, in which case it returns LONG_MAX or LONG_MIN. The function returns 0 if no conversion could be performed.

The strtoul function returns the converted value, if any. If no conversion can be performed, the function returns 0. The function returns ULONG_MAX on overflow.

In all these functions, errno is set to ERANGE if overflow or underflow occurs.

tolower, toupper FUNCTIONS

Header file statement: #include <ctype.h>

Syntax: `int tolower(` int c);

`int toupper(` int c);

Parameter	**Description**
`c`	Character to be converted

The tolower and toupper routines macros convert a single character, as described below:

Function	**MacroDescription**
tolower	tolower Converts c to lowercase if appropriate
toupper	toupper Converts c to uppercase if appropriate

The tolower routine converts c to lowercase if c represents an uppercase letter. Otherwise, c is unchanged.

The `toupper` routine converts `c` to uppercase if `c` represents an lowercase letter. Otherwise, c is unchanged.

Return Value

The tolower and toupper routines return the converted character c. There is no error returned.

va_arg, va_end, va_start FUNCTIONS

Header file statement: #include <stdarg.h>

Syntax: `type va_arg`(va_list arg_ptr, type);

`void va_end`(va_list arg_ptr);

`void va_start`(va_list arg_ptr, prev_param)

Parameter	**Description**
arg_ptr	Pointer to list of arguments
prev_param	Pointer preceding first optional argument
type	Type of argument to be retrieved

The va_arg, va_end, and va_start macros provide a portable way to access the arguments to a function when the function takes a variable number of arguments. The macros are listed below:

Macro	**Description**
va_arg	Macro to retrieve current argument
va_end	Macro to reset arg_ptr
va_list	The typedef for the pointer to list of arguments
va_start	Macro to set arg_ptr to beginning of list of optional arguments

The macros assume that the function takes a fixed number of required arguments, followed by a variable number of optional arguments. The required arguments are declared as ordinary parameters to the function and can be accessed through the parameter names. The optional arguments are accessed through the macros in STDARG.H, which set a pointer to the first optional argument in the argument list, retrieve arguments from the list, and reset the pointer when argument processing is completed.

The ANSI C standard macros, defined in STDARG.H, are used as follows:

- All required arguments to the function are declared as parameters in the usual way.
- The va_start macro sets arg_ptr to the first optional argument in the list of arguments passed to the function. The argument arg_ptr must have va_list type. The argument prev_param is the name of the required parameter immediately preceding the first optional argument in the argument list. If prev_param is declared with the register

storage class, the macro’s behavior is undefined. The va_start macro must be used before va_arg is used for the first time.

- The va_arg macro does the following:
  - Retrieves a value of type from the location given by arg_ptr
  - Increments arg_ptr to point to the next argument in the list, using the size of type to determine where the next argument starts
  - The va_arg macro can be used any number of times within the function to retrieve arguments from the list.
  - After all arguments have been retrieved, va_end resets the pointer to NULL.

Return Value

The va_arg macro returns the current argument, va_start and va_end do not return values.

APPENDIX A
INITIALIZATION AND LINK FILES

;---

Z8 LARGE MODEL

The following are examples of an initialization file and link file for a Z8 large model.

INTIALIZATION FILE

```
;-------------------------------------------------------------
;           z8initx: C Runtime Startup for z8 large model
;           Copyright (c) ZiLOG, 1999
;-------------------------------------------------------------

            define      .xdata,     space=XDATA
            define      .xbss,      space=XDATA
            define      .ndata,     space=RFILE
            define      .nbss,      space=RFILE
            define      .ndatae, align=2, space=RFILE
            define      .nbsse, align=2, space=RFILE
            define      .const,     space=ROM
            define      .worksec,   ALIGN=16,   space=RFILE
            define      .worksec2,  ALIGN=16,   space=RFILE

            define      .startup,   space=ROM



            segment     .worksec                ; working registers
.WORKREG:   ds 16


            segment     .nbss

            segment     .nbsse
            segment     .xbss

            segment     .ndata

            segment     .ndatae

            segment     .xdata

            segment     .startup

            .def        _c_int0
            .def        __exit


            vector      reset=_c_int0

            .ref        _main
            .ref        .BSS_BASE, .BSS_LENGTH
            .ref        .XBSS_BASE, .XBSS_LENGTH
            .ref        .BSSE_BASE, .BSSE_LENGTH
            .ref        .TOS
            .ref        .XBSS_END
            .ref        .DATA_COPY, .DATAE_COPY
            .ref        .XDATA_COPY, .XDATAE_COPY
            .ref        .DATA_BASE, .DATA_LENGTH
            .ref        .DATAE_BASE, .DATAE_LENGTH
            .ref        .XDATA_BASE, .XDATA_LENGTH


.INITBSS    .equ        1
.INITDATA   .equ        1
.INITSIO    .equ        1

;---------------------------------------------------------------
;           Program Entry Point
;---------------------------------------------------------------

_c_int0:
            ld          rp,# .WORKREG; Set Register Pointer
            and         rp,#%F0
            ld          p01m,#%92   ; Stack in external memory, enable
ADR8-15
            ld          p2m,#0
            ld          p3m,#%01    ; Enable XDATA,push-pull
            ld          sph, #HIGH .TOS; Load Stack Pointer High
            ld          spl, #LOW .TOS; Load Stack Pointer Low

            .if         .INITSIO
            ld          p3m,#%41    ; P3M set for bit rate generation
            ld          pre0,#%d    ; Prescalar = 3, T0 = 2 for 9600
            ld          t0,#%2                  ; T0 = 2
            ld          tmr,#%3                 ; Start bit rate
generator
            clr         imr                     ; Clear IMR
            ei                                  ; Enable Interrupt
            di                                  ; Disable Interrupt
            or          irq,#%10    ; Enable IRQ4
            ei
            .endif

            .if         .INITBSS    ; Clear the bss area

            ld          r1,#.BSS_LENGTH
            or          r1,r1
            jr          z,_c_bss_done
            ld          r0,#.BSS_BASE

_c_bss_loop:
            clr         @r0
            inc         r0
            djnz        r1, _c_bss_loop

_c_bss_done:
            ld          r2,#HIGH .XBSS_LENGTH
            ld          r3,#LOW .XBSS_LENGTH
            ld          r5,r2
            or          r5,r3
            jr          z,_c_xbss_done
            ld          r0,#HIGH .XBSS_BASE
            ld          r1,#LOW .XBSS_BASE
            clr         r4

_c_xbss_loop:
            lde         @rr0,r4
            incw        rr0
            decw        rr2
            jr          nz,_c_xbss_loop
```

```
_c_xbss_done:
            ld          r1,#.BSSE_LENGTH
            or          r1,r1
            jr          z,_c_bsse_done
            ld          r0,#.BSSE_BASE

_c_bsse_loop:
            clr         @r0
            inc         r0
            djnz        r1, _c_bsse_loop

_c_bsse_done:

            .endif

            .if         .INITDATA   ; Copy the data tables from ROM to
RAM

            ld          r1,#.DATA_LENGTH
            or          r1,r1
            jr          z,_c_data_done
            ld          r0,#.DATA_BASE
            ld          r2,#HIGH .DATA_COPY
            ld          r3,#LOW .DATA_COPY

_c_data_loop:
            ldci        @r0,@rr2
            djnz        r1, _c_data_loop

_c_data_done:

            ld          r2,#HIGH .XDATA_LENGTH
            ld          r3,#LOW .XDATA_LENGTH
            ld          r6,r2
            or          r6,r3
            jr          z,_c_xdata_done
            ld          r0,#HIGH .XDATA_BASE
            ld          r1,#LOW .XDATA_BASE
            ld          r4,#HIGH .XDATA_COPY
            ld          r5,#LOW .XDATA_COPY

_c_xdata_loop:
            ldc         r6,@rr4
            lde         @rr0,r6
            incw        rr0
            incw        rr4
            decw        rr2
            jr          nz,_c_xdata_loop

_c_xdata_done:
            ld          r1,#.DATAE_LENGTH
            or          r1,r1
            jr          z,_c_datae_done
            ld          r0,#.DATAE_BASE
            ld          r2,#HIGH .DATAE_COPY
            ld          r3,#LOW .DATAE_COPY

_c_datae_loop:
            ldci        @r0,@rr2
            djnz        r1, _c_datae_loop

_c_datae_done:

            .endif
;-------call main

            clr         r0          ; r0 = 0
            push        r0          ; argv = NULL
            push        r0

            push        r0          ; argc = 0
            push        r0
            call        _main

            add         spl,#4
            adc         sph,#0

            clr         r0          ; r0 = 0
            push        r0          ;
            push        r0
            call        __exit
            add         spl,#2
            adc         sph,#0
            ret

__exit:
            push        r15
            push        r14
            ld          r15,spl
            ld          r14,sph
            add         r15,#4
            adc         r14,#0
            lde         r0, @rr14
            incw        rr14
            lde         r1, @rr14
            jr          $
            pop         r14
            pop         r15
            ret
```

LINK FILE

```
order       .startup
order       .worksec2, .ndatae, .ndata, .nbsse, .nbss
order       .xdata, .xbss
range       rfile 4:239
range       .worksec 224:239
range       xdata 32768:65535
define      .BSS_BASE=base of .nbss
define      .BSS_LENGTH=length of .nbss
define      .BSS_END=base of .nbss + length of .nbss
define      .XBSS_BASE=base of .xbss
define      .XBSS_LENGTH=length of .xbss
define      .XBSS_END=base of .xbss + length of .xbss
define      .BSSE_BASE=base of .nbsse
define      .BSSE_LENGTH=length of .nbsse


define      .DATA_COPY=copy base of .ndata
define      .DATAE_COPY=copy base of .ndatae
define      .XDATA_COPY=copy base of .xdata

copy        .ndata ROM
copy        .ndatae ROM
copy        .xdata      ROM

define      .DATA_BASE=base of .ndata
define      .DATA_LENGTH=length of .ndata
define      .DATAE_BASE=base of .ndatae
define      .DATAE_LENGTH=length of .ndatae
define      .XDATA_BASE=base of .xdata
```

```
            define      .XDATA_LENGTH=length of .xdata

            define      .TOS=highaddr of XDATA
            define      .BOS=freemem of XDATA
```

Z8 SMALL MODEL

The following are examples of an initialization file and link file for a Z8 small model.

INITIALIZATION FILE

```
;---------------------------------------------------------------
;           z8inits: C Runtime Startup for z8 small model
;           Copyright (c) ZiLOG, 1999
;---------------------------------------------------------------

            define      .xdata, space=XDATA
            define      .xbss, space=XDATA
            define      .ndata, space=RFILE
            define      .nbss, space=RFILE
            define      .ndatae, align=2, space=RFILE
            define      .nbsse, align=2, space=RFILE
            define      .const,                 space=ROM

            define      .startup,space=ROM
            define      .worksec,ALIGN=16,space=RFILE
            define      .worksec2,ALIGN=16,space=RFILE

            segment     .worksec                ; working registers
.WORKREG:   ds 16

            segment     .nbss

            segment     .nbsse

            segment     .xbss
            segment     .ndata

            segment     .ndatae

            segment     .xdata

            segment     .startup

            .def        _c_int0
            .def        __exit


            vector      reset=_c_int0

            .ref        _main
            .ref        .BSS_BASE, .BSS_LENGTH
            .ref        .XBSS_BASE, .XBSS_LENGTH
            .ref        .BSSE_BASE, .BSSE_LENGTH
            .ref        .BSS_END
            .ref        .TOS
            .ref        .DATA_COPY, .DATAE_COPY
            .ref        .XDATA_COPY, .XDATAE_COPY
            .ref        .DATA_BASE, .DATA_LENGTH
            .ref        .DATAE_BASE, .DATAE_LENGTH
            .ref        .XDATA_BASE, .XDATA_LENGTH


.INITBSS    .equ        1
.INITDATA   .equ        1
.INITSIO    .equ        1


;---------------------------------------------------------------
;           Program Entry Point
;---------------------------------------------------------------

_c_int0:
            ld          rp,# .WORKREG; Set Register Pointer
            and         rp,#%F0
            ld          p01m,#%96   ; Stack in internal memory, enable
ADR8-15
            ld          p2m,#%0
            ld          p3m,#%01    ; Enable XDATA
            ld          spl,#LOW  .TOS; Load Stack Pointer Low
            .if         .INITSIO
            ld          p3m,#%41     ; P3M set for bit rate generation
            ld          pre0,#%d     ; Prescalar = 3, T0 = 2 for 9600
            ld          t0,#%2                   ; T0 = 2
            ld          tmr,#%3                  ; Start bit rate
generator
            clr         imr                      ; Clear IMR
            ei                                   ; Enable Interrupt
            di                                   ; Disable Interrupt
            or          irq,#%10     ; Enable IRQ4
            ei
            .endif

            .if         .INITBSS     ; Clear the bss area


            ld          r1,#.BSS_LENGTH
            or          r1,r1
            jr          z,_c_bss_done
            ld          r0,#.BSS_BASE

_c_bss_loop:
            clr         @r0
            inc         r0
            djnz        r1, _c_bss_loop

_c_bss_done:

            ld          r2,#HIGH .XBSS_LENGTH
            ld          r3,#LOW .XBSS_LENGTH
            ld          r5,r2
            or          r5,r3
            jr          z,_c_xbss_done
            ld          r0,#HIGH .XBSS_BASE
            ld          r1,#LOW .XBSS_BASE
            clr         r4

_c_xbss_loop:
            lde         @rr0,r4
            incw        rr0
            decw        rr2
            jr          nz,_c_xbss_loop
```

```
_c_xbss_done:

            ld          r1,#.BSSE_LENGTH
            or          r1,r1
            jr          z,_c_bsse_done
            ld          r0,#.BSSE_BASE

_c_bsse_loop:
            clr         @r0
            inc         r0
            djnz        r1, _c_bsse_loop

_c_bsse_done:

            .endif

            .if         .INITDATA   ; Copy the data tables from ROM to
RAM

            ld          r1,#.DATA_LENGTH
            or          r1,r1
            jr          z,_c_data_done
            ld          r0,#.DATA_BASE
            ld          r2,#HIGH .DATA_COPY
            ld          r3,#LOW .DATA_COPY

_c_data_loop:
            ldci        @r0,@rr2
            djnz        r1, _c_data_loop

_c_data_done:

            ld          r2,#HIGH .XDATA_LENGTH
            ld          r3,#LOW .XDATA_LENGTH
            ld          r6,r2
            or          r6,r3
            jr          z,_c_xdata_done
            ld          r0,#HIGH .XDATA_BASE
            ld          r1,#LOW .XDATA_BASE
            ld          r4,#HIGH .XDATA_COPY
            ld          r5,#LOW .XDATA_COPY

_c_xdata_loop:
            ldc         r6,@rr4
            lde         @rr0,r6
            incw        rr0
            incw        rr4
            decw        rr2
            jr          nz,_c_xdata_loop

_c_xdata_done:
            ld          r1,#.DATAE_LENGTH
            or          r1,r1
            jr          z,_c_datae_done
            ld          r0,#.DATAE_BASE
            ld          r2,#HIGH .DATAE_COPY
            ld          r3,#LOW .DATAE_COPY

_c_datae_loop:
            ldci        @r0,@rr2
            djnz        r1, _c_datae_loop

_c_datae_done:

            .endif

;-------call main

            clr         r0          ; r0 = 0
            push        r0          ; argv = NULL
            push        r0

            push        r0          ; argc = 0
            push        r0
            call        _main

            add         spl,#4

            clr         r0          ; r0 = 0
            push        r0          ;
            push        r0
            call        __exit
            add         spl,#2
            ret

__exit:
            push        r15
            ld          r15,spl
            ld          r0,3(r15)
            ld          r1,4(r15)
            jr          $
            pop         r15
            ret
```

LINK FILE

```
order       .startup
order       .worksec2, .ndatae, .ndata, .nbsse, .nbss
order       .xdata, .xbss
range       rfile 4:239
range       .worksec 224:239
range       xdata 32768:65535
define      .BSS_BASE=base of .nbss
define      .BSS_LENGTH=length of .nbss
define      .BSS_END=base of .nbss + length of .nbss
define      .XBSS_BASE=base of .xbss
define      .XBSS_LENGTH=length of .xbss
define      .XBSS_END=base of .xbss + length of .xbss
define      .BSSE_BASE=base of .nbsse
define      .BSSE_LENGTH=length of .nbsse


define      .DATA_COPY=copy base of .ndata
define      .DATAE_COPY=copy base of .ndatae
define      .XDATA_COPY=copy base of .xdata

copy        .ndata ROM
copy        .ndatae ROM
copy        .xdata      ROM


define      .DATA_BASE=base of .ndata
define      .DATA_LENGTH=length of .ndata
define      .DATAE_BASE=base of .ndatae
define      .DATAE_LENGTH=length of .ndatae
define      .XDATA_BASE=base of .xdata
define      .XDATA_LENGTH=length of .xdata

define      .TOS=base of .worksec - 1
define      .BOS=freemem of RFILE
```

Z8 PLUS

The following are examples of an initialization file and link file for a Z8 Plus model.

INITIALIZATION FILE

```
;--------------------------------------------------------------
;           z8plusinit: C Runtime Startup for z8 plus
;           Copyright (c) ZiLOG, 1999
;--------------------------------------------------------------


            define      .ndata,     space=RFILE
            define      .nbss,      space=RFILE
            define      .ndatae, align=2, space=RFILE
            define      .nbsse, align=2, space=RFILE
            define      .const,     space=ROM

            define      .startup,   space=ROM, org=20h
            define      .worksec,ALIGN=16,space=RFILE
            define      .worksec2,ALIGN=16,space=RFILE


            segment     .worksec                ; working registers
            .align 16
.WORKREG:   ds 16

            segment     .nbss

            segment     .nbsse

            segment     .ndata

            segment     .ndatae

            segment     .startup

            .def        _c_int0
            .def        __exit


            .ref        _main
            .ref        .BSS_BASE, .BSS_LENGTH
            .ref        .BSSE_BASE, .BSSE_LENGTH
            .ref        .BSS_END
            .ref        .TOS
            .ref        .DATA_COPY, .DATAE_COPY
            .ref        .DATA_BASE, .DATA_LENGTH
            .ref        .DATAE_BASE, .DATAE_LENGTH


.INITBSS    .equ        1
.INITDATA   .equ        1
.DWATCHDOG  .equ        0

;---------------------------------------------------------------
;           Program Entry Point
;---------------------------------------------------------------

_c_int0:
            .if         .DWATCHDOG
            ld          tctlhi,#%08 ; Disable watchdog timer
            .endif
            ld          rp,# .WORKREG; Set Register Pointer
            and         rp,#%F0
            ld          spl,#LOW  .TOS; Load Stack Pointer Low

            .if         .INITBSS    ; Clear the bss area

            ld          r0,#.BSS_BASE
            ld          r1,#.BSS_LENGTH
            or          r1,r1
            jr          z,_c_bss_done

_c_bss_loop:
            clr         @r0
            inc         r0
            djnz        r1, _c_bss_loop

_c_bss_done:
            ld          r0,#.BSSE_BASE
            ld          r1,#.BSSE_LENGTH
            or          r1,r1
            jr          z,_c_bsse_done

_c_bsse_loop:
            clr         @r0
            inc         r0
            djnz        r1, _c_bsse_loop

_c_bsse_done:
            .endif

            .if         .INITDATA   ; Copy the data tables from ROM to
RAM

            ld          r1,#.DATA_LENGTH
            or          r1,r1
            jr          z,_c_data_done
            ld          r0,#.DATA_BASE
            ld          r2,#HIGH .DATA_COPY
            ld          r3,#LOW .DATA_COPY

_c_data_loop:
            ldci        @r0,@rr2
            djnz        r1, _c_data_loop

_c_data_done:
            ld          r1,#.DATAE_LENGTH
            or          r1,r1
            jr          z,_c_datae_done
            ld          r0,#.DATAE_BASE
            ld          r2,#HIGH .DATAE_COPY
            ld          r3,#LOW .DATAE_COPY

_c_datae_loop:
            ldci        @r0,@rr2
            djnz        r1, _c_datae_loop

_c_datae_done:

            .endif

;-------call main

            call        _main
            push        r0
            push        r1
            call        __exit
            add         spl,#2
            ret
```

```
__exit:
            push        r15
            ld          r15,spl
            ld          r0,3(r15)
            ld          r1,4(r15)
            jr          $
            pop         r15
            ret
```

LINK FILE

```
            order       .startup
            order       .worksec2, .ndatae, ndata, .nbsse, .nbss
            range       rfile 4:63
            define      .BSS_BASE=base of .nbss
            define      .BSS_LENGTH=length of .nbss
            define      .BSS_END=base of .nbss + length of .nbss
            define      .BSSE_BASE=base of .nbsse
            define      .BSSE_LENGTH=length of .nbsse

            define      .DATA_COPY=copy base of .ndata
            define      .DATAE_COPY=copy base of .ndatae

            copy        .ndata ROM
            copy        .ndatae ROM

            define      .DATA_BASE=base of .ndata
            define      .DATA_LENGTH=length of .ndata
            define      .DATAE_BASE=base of .ndatae
            define      .DATAE_LENGTH=length of .ndatae

            define      .TOS=highaddr of RFILE
            define      .BOS=freemem of RFILE
```

APPENDIX B
ASCII CHARACTER SET

TABLE B-1. ASCII CHARACTER SET

Graphic	Decimal	Hexadecimal	Comments
	0	0	Null
	1	1	Start Of Heading
	2	2	Start Of Text
	3	3	End Of Text
	4	4	End Or Transmission
	5	5	Enquiry
	6	6	Acknowledge
	7	7	Bell
	8	8	Backspace
	9	9	Horizontal Tabulation
	10	A	Line Feed
	11	B	Vertical Tabulation
	12	C	Form Feed
	13	D	Carriage Return
	14	E	Shift Out
	15	F	Shift In

TABLE B-1. ASCII CHARACTER SET (CONTINUED)

Graphic	Decimal	Hexadecimal	Comments
	16	10	Data Link Escape
	17	11	Device Control 1
	18	12	Device Control 2
	19	13	Device Control 3
	20	14	Device Control 4
	21	15	Negative Acknowledge
	22	16	Synchronous Idle
	23	17	End Of Block
	24	18	Cancel
	25	19	End Of Medium
	26	1A	Substitute
	27	1B	Escape
	28	1C	File Separator
	29	1D	Group Separator
	30	1E	Record Separator
	31	1F	Unit Separator
	32	20	Space
!	33	21	Exclamation Point
"	34	22	Quotation Mark
#	35	23	Number Sign
$	36	24	Dollar Sign
%	37	25	Percent Sign
&	38	26	Ampersand
'	39	27	Apostrophe

TABLE B-1. ASCII CHARACTER SET (CONTINUED)

Graphic	Decimal	Hexadecimal	Comments
(	40	28	Opening (Left) Parenthesis
)	41	29	Closing (Right) Parenthesis
*	42	2A	Asterisk
+	43	2B	Plus
,	44	2C	Comma
-	45	2D	Hyphen (Minus)
.	46	2E	Period
/	47	2F	Slant
0	48	30	Zero
1	49	31	One
2	50	32	Two
3	51	33	Three
4	52	34	Four
5	53	35	Five
6	54	36	Six
7	55	37	Seven
8	56	38	Eight
9	57	39	Nine
:	58	3A	Colon
;	59	3B	Semicolon
<	60	3C	Less Than
=	61	3D	Equals
>	62	3E	Greater Than
?	63	3F	Question Mark

TABLE B-1. ASCII CHARACTER SET (CONTINUED)

Graphic	**Decimal**	**Hexadecimal**	Comments
@	64	40	Commercial At
A	65	41	Uppercase A
B	66	42	Uppercase B
C	67	43	Uppercase C
D	68	44	Uppercase D
E	69	45	Uppercase E
F	70	46	Uppercase F
G	71	47	Uppercase G
H	72	48	Uppercase H
I	73	49	Uppercase I
J	74	4A	Uppercase J
K	75	4B	Uppercase K
L	76	4C	Uppercase L
M	77	4D	Uppercase M
N	78	4E	Uppercase N
0	79	4F	Uppercase 0
P	80	50	Uppercase P
Q	81	51	Uppercase Q
R	82	52	Uppercase R
S	83	53	Uppercase S
T	84	54	Uppercase T
U	85	55	Uppercase U
V	86	56	Uppercase V
W	87	57	Uppercase W

TABLE B-1. ASCII CHARACTER SET (CONTINUED)

Graphic	Decimal	Hexadecimal	Comments
X	88	58	Uppercase X
Y	89	59	Uppercase Y
Z	90	5A	Uppercase Z
[	91	5B	Opening (Left) Bracket
\	92	5C	Reverse Slant
]	93	5D	Closing (Right) Bracket
^	94	5E	Circumflex
_	95	SF	Underscore
`	96	60	Grave Accent
a	97	61	Lowercase A
b	98	62	Lowercase B
c	99	63	Lowercase C
d	100	64	Lowercase D
e	101	65	Lowercase E
f	102	66	Lowercase F
g	103	67	Lowercase G
h	104	68	Lowercase H
i	105	69	Lowercase I
j	106	6A	Lowercase J
k	107	6B	Lowercase K
1	108	6C	Lowercase L
m	109	6D	Lowercase M
n	110	6E	Lowercase N
o	111	6F	Lowercase O

TABLE B-1. ASCII CHARACTER SET (CONTINUED)

Graphic	Decimal	Hexadecimal	Comments
p	112	70	Lowercase P
q	113	71	Lowercase Q
r	114	72	Lowercase R
s	115	73	Lowercase S
t	116	74	Lowercase T
u	117	75	Lowercase U
v	118	76	Lowercase V
w	119	77	Lowercase W
x	120	78	Lowercase X
y	121	79	Lowercase Y
z	122	7A	Lowercase Z
{	123	7B	Opening (Left) Brace
|	124	7C	Vertical Line
}	125	7D	Closing (Right) Brace
~	126	7E	Tilde
	127	7F	Delete

APPENDIX C
PROBLEM/SUGGESTION REPORT FORM

If you experience any problems while using this product, or if you note any inaccuracies while reading the User's Manual, please copy this form, fill it out, then mail or fax it to ZiLOG. We also welcome your suggestions!

Customer Information

Name		Country	
Company		Telephone	
Address		Fax Number	
City/State/ZIP		E-Mail Address	

Product Information and Return Information

Serial # or Board Fab #/Rev. #	**ZiLOG**, Inc.
Software Version	Wordlwide Customer Support Center
Manual Number	4201 Bee Caves Road Suite C-100
Host Computer Description/Type	Austin, Tx. 78746
	Fax Number: (512) 306-4072
	Email: csupport@zilog.com

Problem Description or Suggestion

Provide a complete description of the problem or your suggestion. If you are reporting a specific problem, include all steps leading up to the occurrence of the problem. Attach additional pages as necessary.

GLOSSARY

AABS	Absolute Value
Address Space	Physical or logical area of the target system's Memory Map. The memory map could be physically partitioned into ROM to store code, and RAM for data. The memory can also be divided logically to form separate areas for code and data storage.
ANSI	American National Standards Institute.
ASCII	American Standard Code of Information Interchange.
ASM	Assembler File.
B	Binary.
Binary	Number system based on 2. A binary digit is a bit.
Bisynchronous Communications	A protocol for communications data transfer used extensive in mainframe computer networks. The sending and receiving computers synchronize their clocks before data transfer may begin.
Bit	A digit of a binary system. It has only two possible values: 0 or 1.

BPS	Bits Per Second. Number of binary digits transmitted every second during a data-transfer procedure.
Buffer	Storage Area in Memory.
Bug	A defect or unexpected characteristic or event.
Bus	In Electronics, a parallel interconnection of the internal units of a system that enables data transfer and control Information.
Byte	A collection of four sequential bits of memory. Two sequential bytes (8 bits) comprise one word.
CALL	This command invokes a subroutine
Checksum	A field of one or more bytes appended to a block of *n* words which contains a truncated binary sum formed from the contents of that block. The sum is used to verify the integrity of data in a ROM or on a tape.
COM	Device name used to designate a communication port.

Control Section	A continuous logical area containing code or user data. Each control section has a name. The linker puts all those control sections with the same name in one entity. The linker provides address spaces to the control sections. There are either absolute control sections or relocatable ones.
CPU	Central Processing Unit.
Cross-Linkage Editor	A linkage editor that executes on a processor that is not the same as the target processor.
DSP	Digital Signal Processing. A specialized micropro-cessor that is tailored to perform high repetition math processing and improve signal quality.
Emulator	An emulation device. For example, an In-Circuit Emulator (ICE) module duplicates the behavior of the chip it emulates in the circuit being tested.
External Symbol	A symbol that is referenced in the current program file but is defined in another program file.
GUI	Graphical User Interface. The windows and text that a user sees on their computer screen when they are using a program.
H	Hexadecimal, Half-Carry Flag.
Hex	Hexadecimal.
Hexadecimal	A Base-16 Number System. Hex values are often substituted for harder to read binary numbers.
ICE	In-Circuit Emulator. A ZiLOG product which supports the application design process.
IE	Interrupt Enable.

IM	Immediate Data Addressing Mode.
IMASK	Interrupt Mask Register.
IMR	Interrupt Mask Register.
INC	Increment.
INCW	Increment Word.
Initialize	To establish start-up parameters, typically involving clearing all of some part of the device's memory space.
Instruction	Command.
INT	Interrupt.
Internal Symbol	A symbol that is defined in a program file. This symbol could be visible to multiple functions within the same program file.
I/O	Input/Output. In computers, the part of the system that deals with interfacing to external devices for input or output, such as keyboards or printers.
IPR	Interrupt Priority Register.
Ir	Indirect Working-Register Pair Only.
IR	Infrared. A light frequency range just below that of visible light.
IRQ	Interrupt Request.
ISDN	Integrated Services Digital Network.
ISO	International Standards Organization.

JP	Jump.
JR	Jump Range.
Library	A File Created by a Librarian. This file contains a collection of object modules that were created by an assembler or directly by a C compiler.
Local Symbol	Symbol visible only to a particular function within a program file.
LSB	Least Significant Bit.
MCU	Microcontroller or Microcomputer Unit.
MI	Minus.
MLD	Multiply and Load.
MPYA	Multiply and ADD.
MPYS	Multiply and Subtract.
MSB	Most Significant Bit.
Nibble	A Group of 4 Bits.
NMI	Non-Maskable Interrupt.
NOP	No Operation.
Object Module	Programming code created by assembling a file with an assembler or compiling a file with a compiler. These are relocatable object modules and are input to the linker in order to produce an executable file.
OMF	Object Module Format.

OPC	Operation Code.
Op Code	Operation Code.
OTP	One-Time Programmable.
PCON	Port configuration register.
PER	Peripheral. A device which supports the import or output of information.
POP	Retrieve a Value from the Stack.
POR	Power-On Reset.
Port	The point at which a communications circuit terminates at a Network, Serial, or Parallel Interface card.
PRE	Prescaler.
PROM	Programmable Read-Only Memory.
Protocol	Formal set of communications procedures governing the format and control between two communications devices. A protocol determines the type of error checking to be used, the data compression method, if any, how the sending device will indicate that it has finished sending a message, and how the receiving device will indicate that it has received a message.
PRT	Programmable Reload Timer or Print.
PTR	Pointer.
PTT	Post, Telephone, and Telegraph. Agency in many countries that is responsible for providing telecommunication approvals.

Public/Global Symbol	A programming variable that is available to more than one program file.
PUSH	Store a Value In the Stack.
r	Working Register Address.
R	Register or Working-Register Address, Rising Edge.
RA	Relative Address.
RAM	Random-Access Memory. A memory that can be written to or read at random. The device is usually volatile, which means the data is lost without power.
RC	Resistance/Capacitance.
RD	Read.
RES	Reset.
Resolution	In a digital image, the total number of pixels in the horizontal and vertical directions.
RFSH	Refresh.
ROM	Read-Only Memory. Nonvolatile memory that stores permanent programs. ROM usually consists of solid-state chips.
ROMCS	ROM Chip Select.
RP	Register Pointer.
RR	Read Register or Rotate Right.
SCF	Set C Flag.

SIO	Serial Input/Output.
SL	Shift Left or Special Lot.
SLL	Shift Left Logical.
SMR	Stop Mode Recovery.
SN	Serial Number.
SOIC	Small Outline IC.
SP	Stack Pointer.
SPH	Stack Pointer High.
SPI	Serial Peripheral Interface.
SPL	Stack Pointer Low.
SRAM	Static Random Access Memory.
SR	Shift Right.
SRA	Shift Right Arithmetic.
SRC	Source.
SSI	Small Scale Integration. Chip that contains 5 to 50 gates or transistors.
Static	Characteristic of Random Access Memory that enables It to operate without clocking signals.
ST	Status.
STKPTR	Stack Pointer.

SUB	Subtract.
SVGA	Super Video Graphics Adapter.
S/W	Software.
SWI	Software Interrupt.
Symbol Definition	Symbol defined when the symbol name is associated with a certain amount of memory space, depending on the type of the symbol and the size of Its dimension.
Symbol Reference	Symbol referenced within a program flow, whenever It is accessed for a read, write, or execute operation.
SYNC	Synchronous Communication Protocol. An event or device is synchronized with the CPU or other process timing.
TC	Time Constant.
TCM	Trellis Coded Modulation.
TCR	Timer Control Register.
TMR	Timer Mode Register.
UART	Universal Asynchronous Receiver Transmitter. Component or functional block that handles asynchronous communications. Converts the data from the parallel format in which it is stored, to the serial format for transmission.
UGE	Unsigned Greater Than or Equal.
UGT	Unsigned Greater Than.

ULE	Unsigned Less Than or Equal.
ULT	Unsigned Less Than.
UM	User's Manual.
USART	Universal Synchronous/Asynchronous Receiver/Transmitter. Can handle synchronous as well as asynchronous transmissions.
USB	Universal Serial Bus.
USC	Universal Serial Controller.
UTB	Use Test Box. A board or system to test a particular chip in an end-use application.
V	Volt, Overflow Flag.
V_{CC}	Supply Voltage.
V_{DD}	Voltage from the Digital Power Supply.
V_{PP}	Programmed Voltage.
VRAM	Video Random-Access Memory. A special form of RAM chip that has a separate serial-output port for display refresh operations. This architecture speeds up video adaptor performance.
V_{REF}	Analog Reference Voltage.
WDT	Watch-Dog Timer. A timer that, when enabled under normal operating conditions, must be reset within the time period set within the application (WDTMR (1,0)). If the timer is not reset, a Power-on Reset occurs. Some earlier manuals refer to this timer as the WDTMR.

WDTOUT	Watch-Dog Timer Output.
Word	Amount of data a processor can hold in its registers and process at one time. A DSP word is often 16 bits. Given the same clock rate, a 16-bit controller processes four bytes in the same time it takes an 8-bit controller to process two.
WR	Write.
WS	Wafer Sort.
X	Indexed Address, Undefined.
XOR	Bitwise Exclusive OR.
XTAL	Crystal.
Z	Zero, Zero Flag.
ZASM	ZiLOG Assembler. ZiLOG’s program development environment for DOS.
ZDS	ZiLOG Developer Studio. ZiLOG’s program development environment for Windows 95/98/NT.
ZEM	ZiLOG Emulator.
ZiLOG Symbol Format	Three fields per symbol including a string containing the Symbol Name, a Symbol Attribute, and an Absolute Value in Hexadecimal.
ZLD	ZiLOG Linkage Editor. Cross linkage editor for ZiLOG’s microcontrollers.
ZLIB	ZiLOG Librarian. Librarian for creating library files from locatable object modules for the ZiLOG family of microcontrollers.

ZMASM	ZiLOG Macro Cross Assembler. ZiLOG's program development environment for Windows 3.1.
ZOMF	ZiLOG's Object Module Format. The object module format used by the linkage editor.

INDEX