eZ80ZDS0100ZCC

eZ80 C-Compiler Version 1.03

User Manual

UM005504-0402

ZiLOG Worldwide Headquarters • 532 Race Street • San Jose, CA 95126-3432
Telephone: 408.558.8500 • Fax: 408.558.8300 • www.ZiLOG.com

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

ZiLOG Worldwide Headquarters
532 Race Street
San Jose, CA 95126-3432
Telephone: 408.558.8500
Fax: 408.558.8300
www.ZiLOG.com

Document Disclaimer

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.

Manual Conventions

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

- Arial Medium 10-point ALL-CAPS highlights 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
  - Action(s) performed by the software
  - Icons
- `Courier Regular 10-point` highlights the following items
  - Bit
  - Software code
  - File names and paths
  - Hexadecimal value

Table of Contents

Run Time Environment

is functions 73

List of Figures

List of Tables

Introduction

The eZ80 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 Run Time Environment on page 57. Figure 1 depicts the development flow.

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

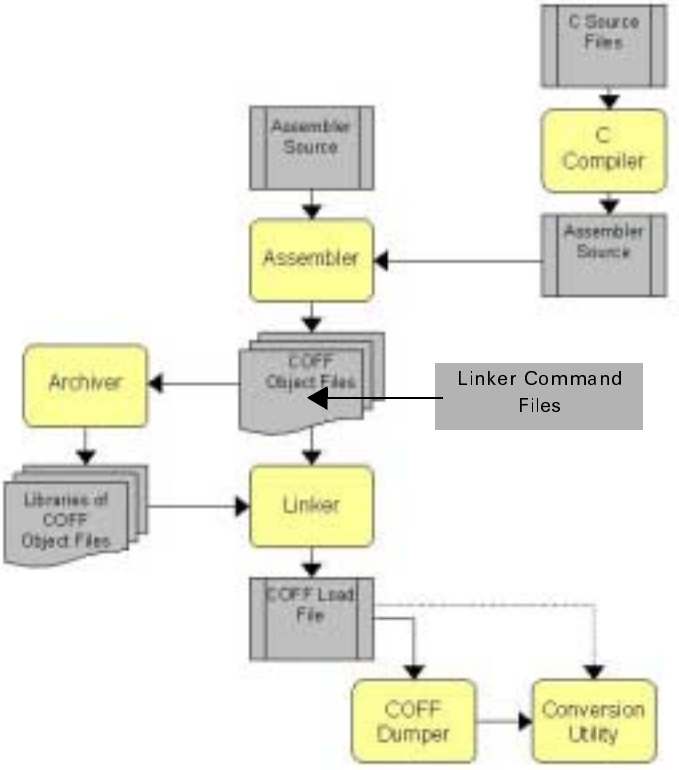

Figure 1. Development Flow

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 eZ80 line 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

Inergers of type Ints and Pointers are 16 bits. A startup program named `eZ80boot.s` is included on the installation diskette. This program clears the `.bss` section, sets the processor mode, and calls the main function.

Note: The startup program does not copy initialized data.

MINIMUM REQUIREMENTS

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

- The eZ80 C-Compiler requires Windows 95, Windows 98, 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,16 MB 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 EZ80 C-COMPILER

To install the eZ80 C-Compiler, insert the eZ80 C-Compiler CD ROM and follow the onscreen prompts

After installing the eZ80 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:

- eZ80 1.00 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
- For eZ80 installation
  - + HKEY_LOCAL_MACHINE\Software\ZiLOG\C Compiler\eZ80
  - + eZ80 Key has Path value which tells where the eZ80 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.

SAMPLE SESSION

The eZ80 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 theeZ80 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, as shown in Figure 2.

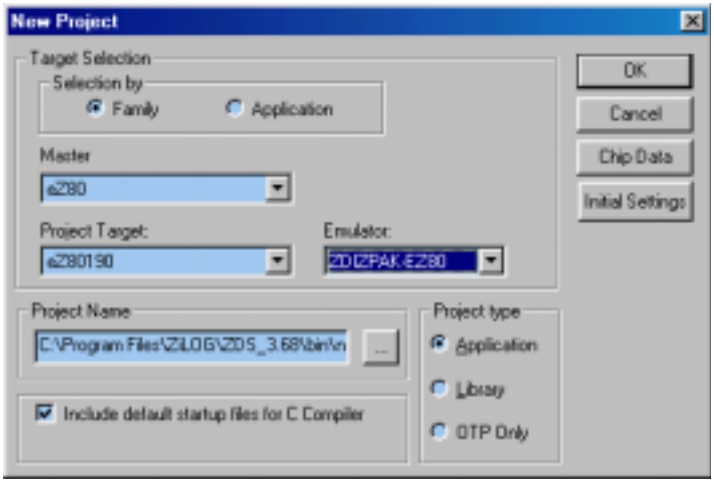

Figure 2. New Project Dialog Box (Example Only)

3. Select `Family` in the Selection by field.
4. Select `eZ80` from the Master pop-up list.
5. Select the processor from the Project Target pop-up list.
6. Select an emulator 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 Adding Included Files on page 9.
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.

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

1. The ZDS New Project Dialog box appears, as shown in Figure 3.

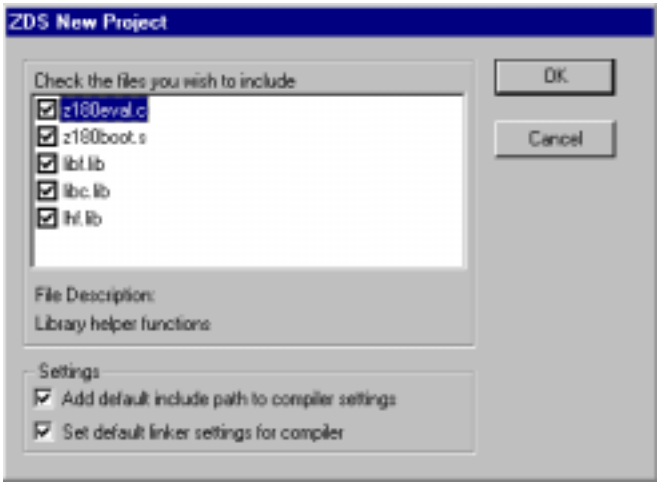

Figure 3. ZDS New Project Dialog Box

2. Select all the files in the Check the files you wish to include window.
3. 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.*
4. Select Set default linker settings for compiler.
5. Click OK. The initialization file for the selected model appears in the project viewer window.
6. Select Optimizations from the Category pop-up list in the C-Compiler Settings Options dialog box. The Optimizations page appears.
7. Select Level 4 optimization.
8. 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 Adding Included Files section if you configured the compiler using the wizard.

ADDING INCLUDED FILES

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.
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 all the files in the lib directory.
8. Click Open. The files appear in the Project Viewer window.

CONFIGURING THE COMPILER

The following section explains how to configure the compiler using ZDS.

Configure Settings

The eZ80 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
- Warnings
- Optimization levels
- Preprocessor symbol definitions
- Code generation configuration

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, see Figure 4.

General Configuration

The C-Compiler General page allows the user 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.
2. Click the Set Default button.
3. Click Apply.

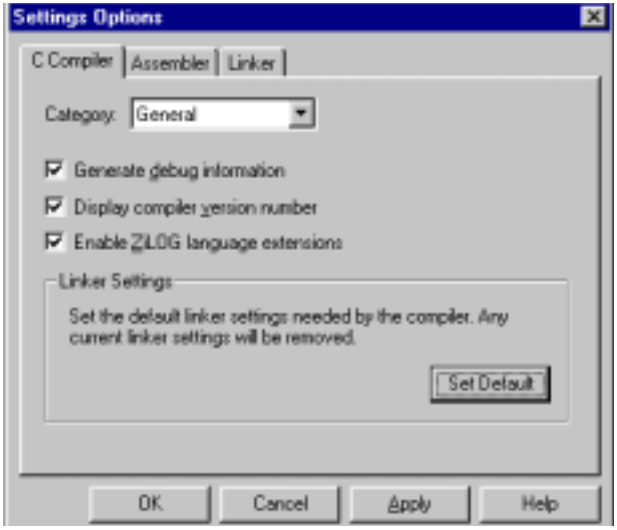

Figure 4. C-Compiler General Page

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.
- The Display compiler version number option causes a two-line message to display in the Output window that shows the C-Compiler copyright notice and version number.
- The Enable ZiLOG extensions causes the C-Compiler to automatically recognize language extensions for the target device. These language extensions allow the microcontroller to communicate with external devices.
- The Set Default button automatically configures the linker for use by the C-Compiler.

Configuring Warnings

The C-Compiler Optimizations page allows the user to control the informational and warning messages that are generated in the ZDS output window.

Perform the following steps to configure the Warnings page, see Figure 5.

1. Select Warnings from the Category pop-up list in the C-Compiler Settings Options dialog box. The Optimizations page appears.
2. Select the warnings to apply.
3. Click Apply.

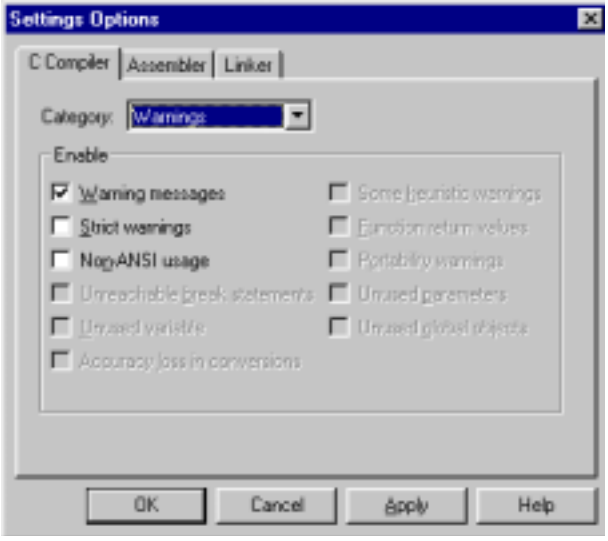

Figure 5. C-Compiler Warnings Page

Configuring Optimization Levels

The C-Compiler Optimizations page allows the user to select an optimization level for the C-Compiler. See Optimization Levels on page 28 for a detailed description of the different optimization levels.

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 as shown in Figure 6.
2. Select Level 4 optimization.
3. Click Apply.

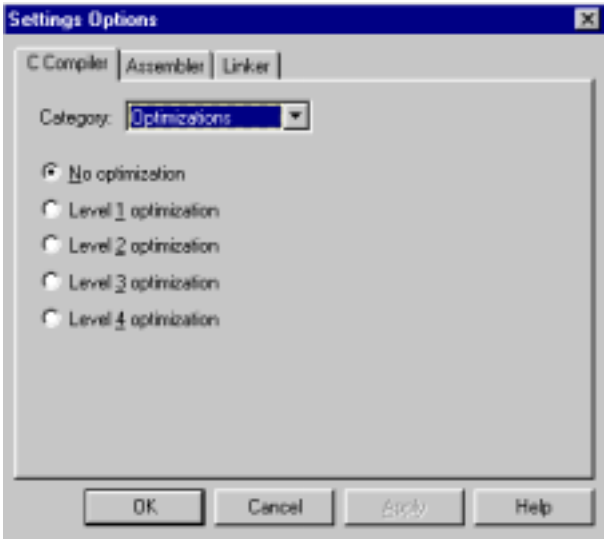

Figure 6. C-Compiler Optimizations Page

The following optimization levels are available in the C-Compiler Optimizations page.

- The No optimization option disables all optimizations.
- The Level 1 optimization option performs:
  - constant folding
  - dead object removal
  - simple jump optimization
- The Level 2 optimization option performs:
  - 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
  - all the optimizations in level 1

- The Level 3 optimization option performs Level 2 optimizations twice, and replaces any redirection of read-only nonvolatile global or static data with a copy of the initial expression.
- The Level 4 optimization option performs Level 2 optimizations three times, and eliminates common sub-expressions by transforming expression trees.

Defining Preprocessor Symbols

The C-Compiler Preprocessor page enables 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 as shown in Figure 7.
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\eZ80` enter `c:PROGRAMS\eZ80\INCLUDE`.

3. Click Apply.

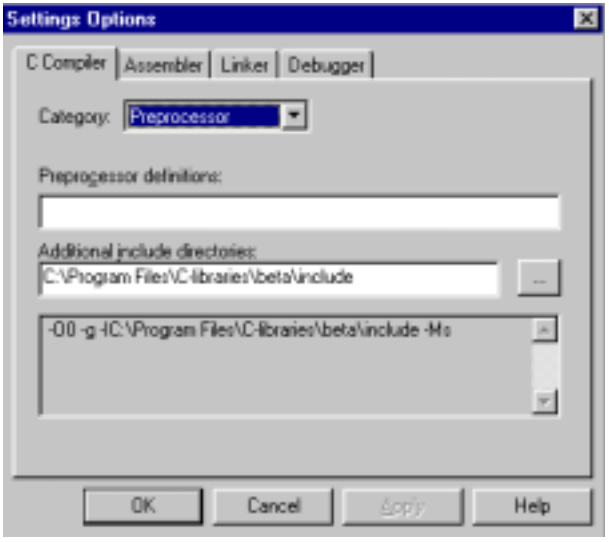

Figure 7. C-Compiler Preprocessor Page

The Preprocessor page defines the following:

- 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 that 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`

Code Generation Configuration

The Code Generation page (Figure 8) allows the user to define the name of memory sections.

Perform the following steps to configure the Memory page.

1. Select Code Generation from the Category pop-up list in the C-Compiler Settings dialog box. The Memory page appears.
2. Enter the names that you have selected to rename the memory sections to.
3. Click Apply.

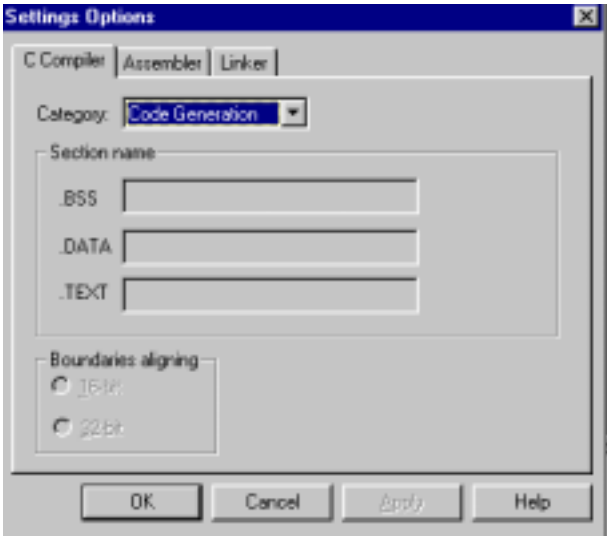

Figure 8. Code Generation Memory Page

Compiling and Connecting to the Emulator

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.

CONTACTING ZILOG CUSTOMER SUPPORT

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 access customer support via the website at:

- For customer service:
  - http://register.zilog.com/login.asp?login=servicelogin
- For technical support:
  - http://register.zilog.com/login.asp?login=supportlogin

For valuable information about hardware and software development tools go to ZiLOG home page at http://www.zilog.com. The latest released version of the ZDS can be downloaded from this site.

C-Compiler Overview

The eZ80 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 a for 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 and I/O address spaces are supported through memory qualifiers
– Support for interrupt functions
– Intrinsic functions are provided for in-line assembly
– Programs containing up to 1 megabyte of code are supported using the eZ80 memory management unit.

LANGUAGE EXTENSIONS

The eZ80 family of processors supports various address spaces that correspond to the different types of addressed locations and the method logical addresses are formed. The C-language, without extensions, is only capable of accessing data in one memory address space. The eZ80 C-Compiler memory extensions allow the user to access data in the eZ80 memory address space, the external I/O address space, or the on-chip I/O address space.

Assigning Types

Types are extended by adding memory qualifiers to the front of a statement. These memory qualifiers are defined with the following keywords:

- **__MEMORY** assigns the type to the standard eZ80 main memory address space.
- **__EXTIO** assigns the type to the external I/O port address space, through which peripheral devices are accessed. There may be no allocations in this space, but pointers to it may be defined and used.
- **__INTIO** assigns the type to the internal (on-chip) I/O port address space, through which peripheral devices and system control registers are accessed. There may be no allocations in this space, but pointers to it may be defined and used.

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:
```
__INTIO char * ptr;
```

In the above example `ptr` is a pointer to `char` in internal I/O address space. The `ptr` is not memory qualified but is a pointer to a qualified memory type.)

DEFAULT MEMORY QUALIFIERS

Default memory qualifiers are applied if no memory qualifiers are specified. In all cases the default memory qualifier is __MEMORY.

POINTERS

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

Size of Pointers

Pointers are always 16-bits in size for the eZ80 C-Compiler.

I/O ADDRESS SPACE

The compiler automatically generates the appropriate I/O instructions for accessing data in the `__INTIO` and `__EXTIO` memory spaces. The machine instructions are described in Table 1.

Table 1. I/O Machine Instructions

Action	__EXTIO	__INTIO
Load	IN	IN0
Store	OUT	OUT0

Accessing I/O Address Space

The eZ80 instruction set does not allow indirect access of the internal I/O address space through a register.

To access the I/O address space, use the on-chip peripheral-addresses as operands to the IN0/OUT0 machine instructions. Variable pointers can not be used to access the internal I/O address space and address constants must be used.

The recommended method to access the I/O address space is shown in the below example.

```
typedef volatile unsigned char __INTIO *PBINTIO;

#defineIO_ADDR((PBINTIO)0x0002)

// …

unsigned char ch;

// …

IO_ADDR[0] = ch;// store to I/O address 2

// …

ch = IO_ADDR[0];// load from I/O address 2

// …
```

INTERRUPT FUNCTIONS

Interrupt functions are declared by preceding their definition with `#pragma` interrupt. Such functions should not take parameters or return a value. For example:

```
#include <zilog.h>
#include <eZ80.h>
volatile int gprtCount;
#pragma interrupt
void timer(void)
{
char cDummy;
cDummy = tcr;
cDummy = tmdr0l;
cDummy = tmdr0h;
gprtCount++;
}
```

Note: The compiler generates the following prologue and epilogue code for interrupt functions:

```
push af
.
.
.
pop af
ei
reti
```

Using the DOS Command Line

The eZ80 C-Compiler can be invoked from the DOS command line.

Command Line Format

The syntax for the eZ80 C-Compiler command line is as follows:

```
eZ80 [switches] ... source ...
```

Command Line Switches

The following command-line switches are recognized.

Table 2. Command Line Switches

Switch	Function
-D <macro>	Define a preprocessor macro
-g	Generate symbolic debug information
-gw	Generate symbolic debug information and facilitate ZDS watch window functionality.
-I <path>	Specify include path. This option may be repeated to specify multiple include paths
-Nbss<name>	Names the uninitialized data section. Default is `.bss`
-Ndata<name>	Names the initialized data section. Default is `.data`
-Ntext<name>	Names the text section. Default is `.text`
-o<name>	Specifies the output assembly file name
-00	No optimization
-01	Level 1 optimization—Basic optimizations: Constant folding, dead object removal and simple jump optimization
-02	Level 2 optimization—Constant propagation, copy propagation, dead code elimination, common sub expression elimination, jump to jump optimization, tail recursion elimination, loop invariant code motion, constant condition evaluation and other condition evaluation optimizations, constant evaluation and expression simplification and all the optimizations in level 1

Table 2. Command Line Switches

Switch	Function
-03	Level 3 optimization—All the optimizations in level 2 are performed twice. Also any redirection on a read only non-volatile global or static data is replaced by a copy of its initial expression.
-04	Level 4 optimization—All the optimizations in level 2 are performed three times. Also common sub expression elimination is attempted through transformation of expression trees (*This is the default optimization level)*
-P <path>	Specify the path where the pre-processor's output should be written
-V	Display compiler version number
-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, and so on.
-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 eZ80:

`eZ80 test.c` generates `test.s`. By default the -04 options is used.

Assembling

The command for eZ80:

`zma -peZ80 -j -otest.o test.s` generates `test.o`

Linking

The command for eZ80:

`zld -mtest -otest` (compiler installation path)`\eZ80inits.o test.o` generates `test.ld` and `test.map`.

OPTIMIZATION LEVELS

The eZ80 C-Compiler allows the user to manually specify the level of optimization to be performed on their code. The optimization levels are controlled through the C-Compiler options dialog box. See Configuring Optimization Levels on page 13 for more information on the C-Compiler Settings Option dialog box.

The eZ80 C-Compiler allows you to specify four levels of optimizations. The optimizations are:

- Level 1 optimization
  - constant folding
  - dead object removal
  - simple jump optimization
- Level 2 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
- all the optimizations in level 1

- Check Level 3 optimization to perform Level 2 optimizations twice, and replace any redirection of read-only nonvolatile global or static data with a copy of its initial expression.
- Check Level 4 optimization to perform Level 2 optimizations three times, and eliminate common sub-expressions by transforming of expression trees.

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.

Note: To enable or disable debug information in ZDS, select Settings from the Options menu. Click the Linker tab and select Output from the Category pop-up list.

Level 1 Optimizations

The following is a description of the optimizations that are performed during a level 1 optimization.

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.

Level 2 Optimizations

Level 2 optimization performs all the optimizations in Level 1 plus the following new optimizations.

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.

Level 3 Optimizations

Level 3 optimization perform all the Level 2 optimizations twice, and replaces any redirection of read-only nonvolatile global or static data with a copy of its initial expression.

Level 4 Optimizations

Level 4 optimization performs Level 2 optimizations three times, and eliminates common sub-expressions by transforming of expression trees.

UNDERSTANDING ERRORS

The eZ80 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 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 15. For command line version the `-I` command line option can be used to specify the include path.

PREDEFINED NAMES

The eZ80 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*

Note: For more information on using the command line see page -26.

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 to be opened. 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.

- Code Section (`.text`)
- Initialized Data Section (`.data`)
- Uninitialized Data Section (`.bss`)
- Constant Data Section (`.const`)

Note: All sections are allocated in the __MEMORY address space. The default sections `.text`, `.data`, and `.bss` can be renamed using the –Ntext, –Ndata, and –Nbss command line options.

INCORPORATING ASSEMBLY WITH C

The eZ80- 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 <zilog.h>
int main()
{
_asm("\tnop\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.

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

Example

The following example shows the C function `imul()` being called from assembly language. The assembly routine pushes two words onto the stack (parameters x and y for the function `imul`), calls the function `imul`, then cleans the arguments off the stack.

Assembly file.

```
.ref _imul; Compiler prepends '_' to names
.text
_main:
pushhl; parameter <y>
pushde; parameter <x>
call_imul
popaf; clean the stack
popaf; *
ret; result in hl:de
```

Referenced C file.

```
typedef unsigned long uint32;
typedef unsigned short uint16;
typedef char int8;
uint32
imul(uint16 x, uint16 y)
{
uint32 res;
int8 i;

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

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 9.

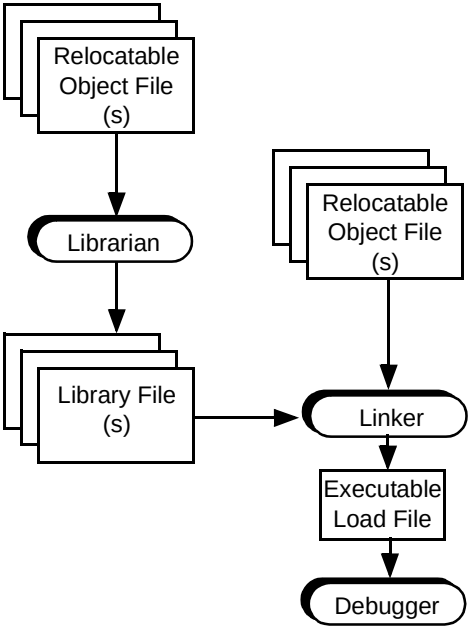

Figure 9. Linker Functional Relationship

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 corre-

sponding 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 10.

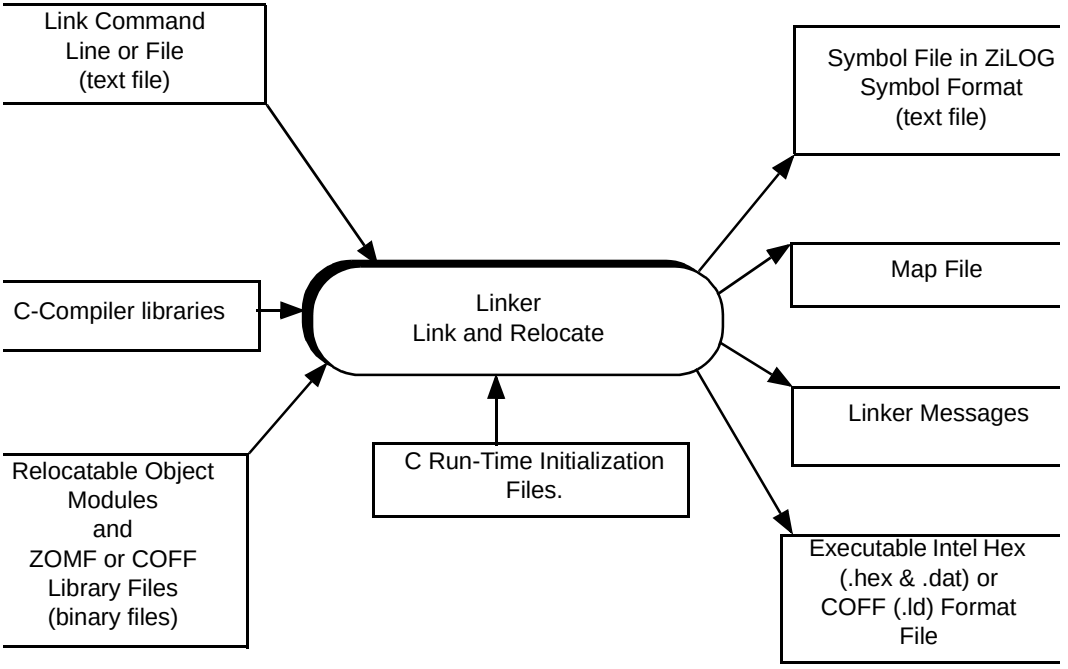

Figure 10. Linker components

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 pointer, and initializes the processor mode resister. Once these initializations are complete the program calls `main`, which is the C entry point.

Installed Files

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

Table 3. Linker Referenced Files

File	Description
eZ80boot.s	Assembly language source of example C startup module
eZ80.lnk	Example linker command file for eZ80
libc.lib	Standard C library without floating point support
libf.lib	Standard C library with floating point support
lhf.lib	Library of runtime helper functions
eZ80mmu.s	MMU static overlay manager

Note: Source files for the run-time initialization files are provided in Run Time Initialization File on page 40.

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: The user needs to include the C-run time initialization file that is appropriate for the compilation model chosen in the project. See Table 3 for a list of initialization files that are included with the C-Compiler. For more information on adding included files see Adding Included Files on page 9.

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. 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 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 for a list of command files that are include with the C-Compiler.

Linker Command Line Example

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

```
zld -otest -A lib-path\eZ80boot.o test.o lib-
path\libc.lib lib-path\eZ80180.link -mtest.map
```

This example generates a `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 50.

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 4 shows the symbols that are used by the linker.

Table 4. Linker Symbols

Symbol	Description
BSS_BASE	Base of .BSS section
BSS_LENGTH	Length of .BSS section
TOS	Top of stack

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 5. For linker options see Table 6.

Table 5. Summary of Linker Commands

Command	Description
Assign	Assigns a control section to an address space
Bankarea	Reserve space for overlay banks
Bank	Assign a section to an overlay bank
Bankvector	Specify an overlay manager vector
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
Locate	Set the base address for the control section
Noload	Set the control section attribute as no load
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 it is not required that commands be specified alphabetically in the command file. Command words and parameters not listed 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 BANKAREA

The BANKAREA linker command reserves an area of an address space for use as an overlay bank. This command is used in conjunction with the BANK linker command. The BANK AREA names the overlay bank, and is referenced by the BANK linker command.

Syntax:

BANKAREA <*bankarea*> <*address-space*> <*start-address*>: <*end-address*>

BANKAREA <*bankarea*> <*address-space*> <*start-address*>, <*length*>

The location and size of the overlay bank is specified in one of two ways:

- A colon-separated area start-address and end-address
- A comma-separated area start-address and length

Example

The following example creates an overlay area named OVERLAY in the ROM address space. The overlay area occupies the address range 08000h to 0BFFFh.

```
BANKAREA OVERLAY, ROM, 08000h: 0BFFFh
```

Linker Command BANK

The BANK linker command assigns a control section to an address space overlay bank. This command is used in conjunction with the BANKAREA linker command. The BANKAREA names the overlay bank, which is defined by the BANKAREA

linker command. The load-address specifies the physical address of the section. If the load-address is omitted, the linker determines the load address.

Syntax: BANK <*section*> <*bankarea*> [[,] <*load-address*>*]*

Example

The following example assigns a section named BOOTSECTION to an overlay area named OVERLAY. The load address of the section is 010000h.

```
BANK BOOTSECTION OVERLAY 010000h
```

Linker Command BANKVECTOR

The BANKVECTOR linker command specifies the vector address used for passing control to overlays. This command is used in conjunction with the BANKAREA and BANK linker commands. Valid values for the vector address depend upon the target processor. For the eZ80 family, the valid values are 0, 8, 16, 24, 32, 40, 48, and 56.

Syntax: BANKVECTOR <*address*>

Example

The following example specifies that vector address 56 should be used as the overlay manager vector address.

```
BANKVECTOR 56
```

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 using 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 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 LOCATE

This command sets the base address for a control section.

Syntax: LOCATE <*name*> <*address*>

Example:LOCATE .text 1000h

The *name* must be a control section name. The *address* be within the address range of the address space to which the control section belongs.

Linker Command NOLOAD

This command sets the attribute of the control section as no load.

Syntax: NOLOAD <*name1*> <*name2*>

Example: NOLOAD csec, dsec

The <*namen*> must be a control section name or a group name.

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 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 is 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 6.

Table 6. 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 must 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 (-).

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, shown in Figure 11, 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 11. 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 7.

Table 7. 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.

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 must perform the following tasks when it is called. Refer to Figure 12.

1. Push the frame pointer (`IX`) onto the stack.
2. Allocate the local frame.
3. If the function modifies `BC`, `DE` or `IY`, push them on the stack. Any other registers may be modified without preserving them.
4. Execute the code for the function.
5. If the function returns a scalar value, place it in the accumulator (HL for scalars greater than eight bits; A for eight-bit scalars), or in the HL:DE register pair for thirty-two bit scalars.
6. Deallocate the local frame.
7. Restore the caller’s frame pointer.
8. Return

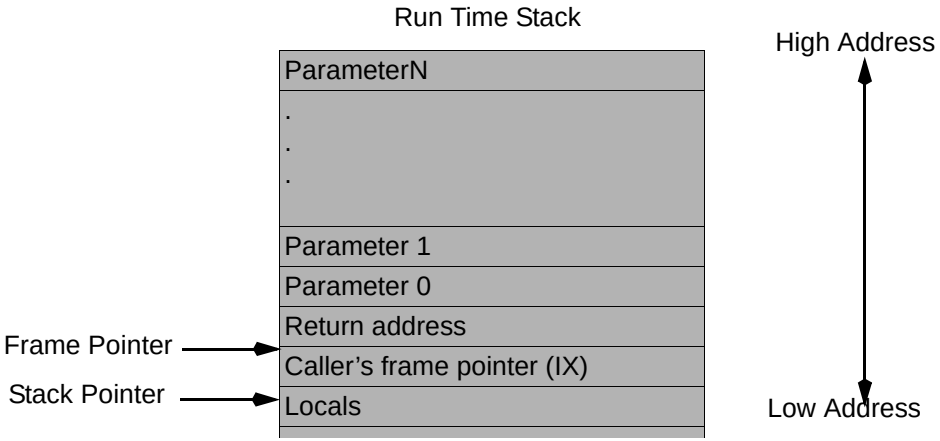

Figure 12. Frame Layout

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 or arguments an d 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.

OVERLAY SUPPORT

The compiler supports program overlays using the eZ80 memory management unit (MMU). Overlays allow the user to increase their maximum executable-file-size from 64 KB (without overlays) to 1 MB (with overlays).

The compiler creates overlays by dividing the application into a resident portion that is loaded upon the application execution. Overlays are then mapped into the MMU bank area as needed.

One load file is created by this method and makes it possible to run large programs. However, the disadvantages are:

- an increase in program execution time. This increase is due to the amount of overhead involved in manipulating the MMU.
- more space is needed because the application contains the code for the overlay manager.

Enabling Overlays

Enable overlays by using the –ZiLOG command-line option. After overlays are enabled all function calls to external modules make use of a `far call` instruction. This function uses four-bytes instead of the usual three-bytes required for a normal (or near) call.

The format of the `far call` is as follows.

`Rst IntNo`	*IntNo is the restart interrupt*
`.byte OverlayNumber`	*The overlay number*
`.word OverlayEntry`	*Offset of the entry point*

The restart instruction passes control to the static overlay manager which determines which overlay is being called, and maps it into the MMU bank area. The restart vector to use is specified using the linker's BANKVECTOR command.

Configuring Source Files

To enable overlay support the user must configure two assembly-language source files that are included in the compiler distribution. Theses files are:

- `eZ80boot.s` C run-time startup module
- `eZ80mmu.s` Static overlay manager

To configure the `eZ80boot.s` file define the symbol `OVERLAY`. When this symbol is defined the startup routine calls the overlay manager initialization routine. The overlay manager initialization routine is required for proper operation of the static overlay manager.

Note: Overlay support is enabled by default.

To configure the `eZ80mmu.s` file perform the following steps.

1. Define the symbol `.FCDEPTH`. This symbol is used to define the function call depth that the overlay manager should support (default is eight). Three bytes are required for each function call level.
2. Define the symbol `.CBAR`. This symbol defines the value to be loaded into the MMU's CBAR register. This can be defined in the `eZ80mmu .s` file itself, or defined in the linker command file or another source file. By default, this is declared as an external symbol and requires that it be defined in a linker command file or another source module. The CBAR register is an eight-bit register, the upper nibble defines the start address of common area 1 and the end address of the bank area. The lower nibble of CBAR defines the start address of the bank area. The static overlay manager assumes that all overlays are mapped into the bank area.

Note: Ensure that the function `__eZ80Overlay` (defined in `eZ80mmu.s`) services the restart interrupt specified using the linker's `BANKVECTOR` command. Do this by loading the appropriate restart vector with the address of `__eZ80Overlay`.

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 15 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 8 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.

The standard C runtime libraries are separated into two files. These two files consist of integer support (`libc.lib`) and floating-point support libraries (`libf.lib`). Both libraries are required to support floating point calculations. Both libraries contain versions of `printf()` and `scanf()`, and their variants to minimize the runtime library size for applications that do not require floating-point support. If floating-point versions of these library functions are required, then the library `libf.lib` should be specified before `libc.lib` in the project file.

Installed files

The header files in Table 8 are installed in the C-Compiler installation directory.

Table 8. Installed Library Files

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

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.

acos function

Header file statement: `#include<math.h>`

Syntax: `double acos (double x);`

Parameter	Description
x	Value whose arccosine is to be calculated

The acos functions return the arccosine of x in the range 0 to Pi radians. The value of x must be between -1 and 1.

Return Value

The acos functions return the arccosine result.

asin function

Header file statement: `#include<math.h>`

Syntax: `double asin (double x);`

Parameter	Description
x	Value whose arcsine is to be calculated

The asin functions calculate the arcsine of x in the range -Pi/2 to Pi/2 radians. The value of x must be between -1 and 1.

Return Value

The asin functions return the arcsine result.

atan, atan2 function

Header file statement: `#include<math.h>`

Syntax: `double atan` (double x)`;`

`double atan2(double y, double x);`

Parameter	Description
x,y	Any number

The atan family of functions calculates the arctangent of x, and the atan2 family of functions calculates the arctangent of y/x. The atan group returns a value in the range -pi/2 to pi/2 radians, and the atan2 group returns a value in the range -pi to pi radians.

The atan2 functions use the signs of both parameters to determine the quadrant of the return value. The atan2 functions are well defined for every point other than the origin, even if x equals 0 and y does not equal 0.

Return Value

The atan family of functions returns the arctangent result

_asm function

Header file statement: `#include <zilog.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. The assembly instruction is placed as is in the assembly file, and the user has to follow all the assembler conventions when emitting the assembly instructions through the `_asm` instruction.

Return Value

There is no return value.

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.

ceil function

Header file statement: `#include<math.h>`

Syntax:`double ceil (double x);`

Parameter	Description
x	Floating-point value

The ceil function returns a double value representing the smallest integer that is greater than or equal to x.

Return Value

This function returns the double result. There is no error return.

cos, cosh function

Header file statement: `#include<math.h>`

Syntax: `double cos (double x);`

`double cosh (double x);`

Parameter	Description
x	Angle in radians

The cos and cosh functions return the cosine and hyperbolic cosine, respectively, of x.

Return Value

The cos function returns the cosine result. The cosh function returns the hyperbolic cosine result.

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.

exp function

Header file statement: `#include<math.h>`

Syntax:`double exp (double x);`

Parameter	Description
x	Floating-point value

The exp function returns the exponential function of their floating-point parameter (x).

Return Value

This function returns the exponential value of x.

fabs function

Header file statement: `#include<math.h>`

Syntax: `double fabs (double x);`

Parameter	Description
x	Floating-point value

The fabs function calculates the absolute value of its floating-point parameter (x).

Return Value

This function returns the absolute value of its argument. There is no error return.

floor function

Header file statement: `#include <math.h>`

Syntax: `double floor (double x);`

Parameter	Description
x	Floating-point value

The floor function returns a floating-point value representing the largest integer that is less than or equal to x.

Return Value

This function returns the floating-point result. There is no error return.

fmod function

Header file statement: `#include <math.h>`

Syntax: `double fmod (double` x, double y`);`

Parameter	Description
x,y	Floating-point values

The fmod function calculates the floating-point remainder f of x / y such that x = i * y + f, where i is an integer, f has the same sign as x, and the absolute value of f is less than the absolute value of y.

Return Value

This function returns the floating-point remainder. If y is 0, the function returns 0.

frexp function

Header file statement: `#include <math.h>`

Syntax: `double frexp (double` x, int * expptr`);`

Parameter	Description
x,y	Floating-point value
expptr	Pointer to stored integer exponent

The frexp function breaks down the floating-point value (x) into a mantissa (m) and an exponent (n), such that the absolute value of m is greater than or equal to 0.5 and less than 1.0, and x equals m times (2 raised to the power of n). The integer exponent n is stored at the location pointed to by expptr.

Return Value

This function returns the mantissa. If x is 0, the function returns 0 for both the mantissa and the exponent. There is no error return.

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:

FunctionTest Condition

isalnumAlphanumeric (‘A’-‘Z’, ‘a’-‘z’, or ‘0’-‘9’)

isalphaLetter (‘A’-‘Z’ or ‘a’-‘z’)

iscntrl Control character (0x00 - 0x1F or 0x7F)

isdigitDigit (‘0’-‘9’)

isgraphPrintable character except space (‘ ‘)

islowerLowercase letter (‘a’-‘z’)

isprint Printable character (0x20 - 0x7E)

ispunct Punctuation character

isspace White-space character (0x09 - 0x0D or 0x20)

isupperUppercase letter (‘A’-‘Z’)

isxdigitHexadecimal 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 the interger does not satisfy the test condition.

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.

ldexp function

Header file statement: `#include <math.h>`

Syntax:`double ldexp` (double x, int exp);

Parameter	Description
x	Floating-point value
exp	Integer exponent

The ldexp function calculates the value of x * (2 raised to the power of exp).

Return Value

This function returns an exponential value.

ldiv function

Header file statement: `#include <stdlib.h>`

Syntax: `ldiv_t ldiv` (long int numer, long int denom);

Parameter	Description
numer	Numerator
denom	Denominator

The ldiv function divides numer by denom, computing the quotient and the 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 program will terminate with an error message.

The ldiv function is similar to the div function, with the difference being that the arguments and the members of the returned structure are all of type long int.

The ldiv_t structure, defined in STDLIB.H, contains the following elements.

Element	Description
long int quot	Quotient
long int rem	Remainder

Return Value

The ldiv function returns a structure of type ldiv_t, comprising both the quotient and the remainder.

log, log10 function

Header file statement: `#include <math.h>`

Syntax:`double log (double x);`

`double log10(double x);`

Element	Description
x	Value whose logarithm is to be found

The log and log10 functions calculate the natural logarithm and the base-10 logarithm, respectively, of x.

Return Value

The log functions return the logarithm of x.

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:

ValueMeaning

< 0buf1 less than buf2

= 0buf1 identical to buf2

> 0buf1 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`.

modf function

Header file statement: `#include <math.h>`

Syntax:`double modf (double` x, double *intptr);

Parameter	Description
x	Floating-point value
intptr	Pointer to stored integer portion

The modf functions breaks down the floating-point value x into fractional and integer parts, each of which has the same sign as x. The signed fractional portion of x is returned. The integer portion is stored as a floating-point value at the location pointed to by the intptr parameter.

Return Value

The modf function returns the signed fractional portion of x. There is no error return.

pow function

Header file statement: `#include <math.h>`

Syntax: `double pow (double` x, double y);

Parameter	Description
x	Number to be raised
y	Power of x

The `pow` function computes x raised to the power of `y`.

Return Value

The `pow` function returns the value of x^y.

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.

sin, sinh function

Header file statement: `#include<math.h>`

Syntax: `double sin (double x);`

double sinh (double x);

Parameter	Description
x	Angle in radians

The `sin` and `sinh` functions find the sine and hyperbolic sine of x, respectively.

Return Value

The `sin` function returns the sine result. The `sinh` function returns the hyperbolic sine result.

sprintf function

Header file statement: `#include <stdio.h>`

Syntax:`int sprintf (char` *buffer, const char *format [, argument]...);

Parameter	Description
buffer	Storage location for output
format	Format-control string
argument	Optional arguments
count	Maximum number of bytes to store

The `sprintf` function formats and stores a series of characters and values in buffer. Each argument (if any) is converted and output according to the corresponding for-

mat specification in the format. A null character is appended to the end of the characters written but is not counted in the return value.

Return Value

The sprintf function returns the number of characters stored in buffer, not counting the terminating null character.

Format Specifiers

Format specifications always begin with a percent sign (`%`) and are read left to right. When the first format specification is encountered, the value of the first argument after format is converted and is output accordingly. The second format specification causes the second argument to be converted and output, and so on. If there are more arguments than there are format specifications, the extra arguments are ignored. The results are undefined if there are not enough arguments for all the format specifications.

Each field of the format specification is a single character or a number signifying a particular format option. The simplest format specification contains only the percent sign and a type character (for example, `%s`). The optional fields, which appear before the type character, control other aspects of the formatting.

After the % sign, the following format specifiers can be used in the following sequence:

- Zero or more flags that modify the meaning of the conversion specification.
- An optional decimal integer specifying a minimum field width (%). If the converted value has fewer characters than the field width, it will be padded with spaces on the left (or right, if the left adjustment flag, described later, has been given) to the field width.
- An optional precision that gives the minimum number of digits to appear for the d, i, o, u, x and X conversions, the number of digits to appear after the decimal point character for e, E and f conversions, the maximum number of significant digits for the g and G conversions, or the maximum number of characters to be written from a string in s conversion. The precision takes the

form of a period (.) followed by an optional decimal integer; if the integer is omitted, it is treated as zero.

- An optional h specifying that a following d, i, o, u, x or X conversion specifier applies to a short int or unsigned short int argument (the argument will have been promoted according to the integral promotions, and its value shall be converted to short int or unsigned short int before printing); an optional h specifying that the following n conversion specifier applies to a pointer to a short int argument; an optional l (ell) specifying that a following d, i, o, u, x or X conversion specifier applies to a long int or unsigned long int argument; an optional l specifying that a following n conversion specifier applies to a pointer to a long int argument; or an optional L specifying that a following e, E, f, g or G conversion specifier applies to a long double argument. If an h, l or L appears with any other conversion specifier, the behavior is undefined.
- A character that specifies the type of conversion to be applied.
- A field width or precision, or both, may be indicated by an asterisk * instead of a digit string. In this case, an int argument supplies the field width or precision. The arguments specifying field width or precision, or both, shall appear (in that order) before the argument (if any) to be converted. A negative field width argument is taken as a - flag followed by a positive field width. A negative precision argument is taken as if it were missing.

  The flag characters and their meanings are:

  – A minus sign (-) means the result of the conversion will be left-justified within the field.

  – A plus sign (+) means the result of a signed conversion will always begin with a plus or minus sign.

  – A space will be prepended to the result if the first character of a signed conversion is not a sign, or if a signed conversion results in no characters. If the space and + flags both appear, the space flag will be ignored.

  – A pound sign means (#) the result will be converted to an “alternate form”. For o conversion, it increases the precision to force the first digit of the result to be a zero. For x (or X) conversion, a nonzero result will have 0x (or 0X) prepended to it. For e, E, f, g and G conversions, the

result will always contain a decimal point character, even if no digits follow it (normally, a decimal point character appears in the result of these conversions only if a digit follows it). For g and G conversions, trailing zeros will not be removed from the result.For other conversions, the behavior is undefined.

– Leading zeroes (0), following any indication of sign or base, are used to pad to the field width for d, i, o, u, x, X, e, E, f, g and G; no space padding is performed. If the O and – flags both appear, the 0 flag will be ignored. For d, i, o, u, x and X conversions, if a precision is specified, the 0 flag will be ignored. For other conversions, the behavior is undefined.

The conversion specifiers and their meanings are:

– The d, i, o, u, x, X specifiers convert the int argument into a signed decimal (d or i), unsigned octal (o), unsigned decimal (u) or unsigned hexadecimal notation (x or X); the letters abcdef are used for x conversion and the letters ABCDEF for X conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting a zero value with an explicit precision of zero is no characters.

– The f specifier converts the double argument into decimal notation using the style [-]ddd.ddd, where the number of digits after the decimal-point character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is explicitly zero, no decimal-point character appears. If a decimal-point character appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.

– The e, E specifiers convert the double argument into the style [-]d.ddde+/-dd, where there is one digit before the decimal-point character (which is nonzero if the argument is nonzero) and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6; if the precision is zero, no decimal-point character appears. The value is rounded to the appropriate number of digits. The E conversion specifier will produce a number with E instead of e introducing the exponent. The

exponent always contains at least two digits. If the value is zero, the exponent is zero.

– The g, G specifiers converts the double argument into the style f or e (or in style E in the case of a G conversion specifier), with the precision specifying the number of significant digits. If an explicit precision is zero, it is taken as 1. The style used depends on the value converted; style e will be used only if the exponent resulting from such a conversion is less than -4 or greater than or equal to the precision. Trailing zeros are removed from the fractional portion of the result; a decimal-point character appears only if it is followed by a digit.
– The c specifier converts the int argument into an unsigned char, and the resulting character is written.
– The s specifier indicates that the argument is a pointer to an array of character type(%%). Characters from the array are written up to (but not including) a terminating null character; if the precision is specified, no more than that many characters are written. If the precision is not specified or is greater than the size of the array, the array shall contain a null character.
– The p specifier indicates that the argument is a pointer to void. The value of the pointer is converted to a sequence of printable characters, in an implementation-defined manner.
– The n specifier indicates that the argument is a pointer to an integer that will contain the number of characters written to the output by the call to sprintf. No argument is converted.
– The percent (%) specifier indicates that no argument is converted. The complete conversion specification is %%.

Notes: The following are rules for the above specifiers:

1. If a conversion specification is invalid, the behavior is undefined.
2. If any argument is, or points to, a union or an aggregate (except for an array of character type using %s conversion, or a pointer cast to be a pointer to void using %p conversion) the behavior is undefined.

3. In no case does a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

sqrt FUNCTION

Header file statement: `#include<math.h>`

Syntax `double sqrt(double x);`

Parameter	Description
x	Nonnegative floating-point value

The sqrt functions calculate the square root of x.

Return Value

The sqrt functions return the square-root result.

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.

sscanf function

Header file statement: #include <stdio.h>

Syntax: `int sscanf (const` char *buffer, const char *format [, argument]...);

Parameter	Description
buffer	Stored data
format	Format-control string
argument	Optional arguments

The `sscanf` function reads data from buffer into the locations given by each argument. Every argument must be a pointer to a variable with a type that corresponds to a type specifier in format. The format controls the interpretation of the input fields.

Return Value

The `sscanf` function returns the number of fields that were successfully converted and assigned. The return value does not include fields that were read but not assigned.

The return value is EOF for an attempt to read at end-of-string. A return value of 0 means that no fields were assigned.

Format Specifiers

The format should be a multi-byte character sequence, beginning and ending in its initial shift state. The format is composed of:

- zero or more directives:
- one or more white-space characters
- an ordinary multi-byte character (not %); or a conversion specification.

Each conversion specification is introduced by the percent (%) character. After the %, the following appear in sequence:

- An optional assignment-suppressing character (*).
- An optional decimal integer that specifies the maximum field width.
- An optional h, l (ell) or L indicating the size of the receiving object. The conversion specifiers d, i and n shall be preceded by h if the corresponding argument is a pointer to short int rather than a pointer to int, or by l if it is a pointer to long int. Similarly, the conversion specifiers o, u and x shall be preceded by h if the corresponding argument is a pointer to unsigned short rather than a pointer to unsigned int, or by l if it is a pointer to unsigned long int. Finally, the conversion specifiers e, f and g shall be preceded by l if the corresponding argument is a pointer to double rather than a pointer to float, or by L if it is a pointer to long double. If an h, l or L appears with any other conversion specifier, the behavior is undefined.
- A character that specifies the type of conversion to be applied. The valid conversion specifiers are described below.

The sscanf function executes each directive of the format in turn. If a directive fails, as detailed below, the sscanf function returns. Failures are described as input failures (due to the unavailability of input characters), or matching failures (due to inappropriate input).

The following rules apply to the execution of a directive:

- A directive composed of white-space is executed by reading input up to the first non-white-space character (which remains unread), or until no more characters can be read.
- A directive that is an ordinary multi-byte character is executed by reading the next characters of the stream. If one of the characters differs from one comprising the directive, the directive fails, and the differing and subsequent characters remain unread.
- A directive that is a conversion specification defines a set of matching input sequences, as described below for each specifier.

A conversion specification is executed in the following steps:

– Input white-space characters (as specified by the isspace function) are skipped, unless the specification includes a [, c or n specifier.

– An input item is read, unless the specification includes an n specifier. An input item is defines as the longest sequence of input characters (up to any specified maximum field width) which is an initial subsequence of a matching sequence. The first character, if any, after the input item remains unread. If the length of the input item is zero, the execution of the directive fails: this condition is a matching failure, unless an error prevented input, in which case it is an input failure.

– Except in the case of a% specifier, the input item (or, in the case of a %n directive, the count of input characters) is converted to a type appropriate to the conversion specifier. If the input item is not a matching sequence, the execution of the directive fails: this condition is a matching failure. Unless assignment suppression was indicated by a *, the result of the conversion is placed in the object pointed to by the first argument following the format argument that has not already received a conversion result.

– If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the space provided, the behavior is undefined.

The following are valid conversion specifiers:

– The `d` specifier matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the strtol function with the value 10 for the base argument. The corresponding argument shall be a pointer to integer.

– The `I` specifier matches an optionally signed integer, whose format is the same as expected for the subject sequence of the strtol function with the value 0 for the base argument. The corresponding argument shall be a pointer to integer.

– The `o` specifier matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of the strtol function with

the value 8 for the base argument. The corresponding argument shall be a pointer to unsigned integer.

– The u specifier matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the strtol function with the value 10 for the base argument. The corresponding argument shall be a pointer to unsigned integer.
– The x specifier matches an optionally signed hexadecimal integer, whose format is the same as expected for the subject sequence of the strtol function with the value 16 for the base argument. The corresponding argument shall be a pointer to unsigned integer.
– The e, f, g specifiers match an optionally signed floating-point number, whose format is the same as expected for the subject string of the strtod function. The corresponding argument shall be a pointer to floating.
– The s specifier matches a sequence of non-white-space characters (%). The corresponding argument shall be a pointer to the initial character of an array large enough to accept the sequence and a terminating null character, which will be added automatically.
– The bracket ([) specifier matches a non-empty sequence of characters (%) from a set of expected characters (the scanset). The corresponding argument shall be a pointer to the initial character of an array large enough to accept the sequence and a terminating null character, which will be added automatically. The conversion specifier includes all subsequent characters in the format string, up to and including the matching right bracket (]). The characters between the brackets (the scanlist) comprise the scanset, unless the character after the left bracket is a circumflex (^), in which case the scanset contains all characters that do not appear in the scanlist between the circumflex and the right bracket. As a special case, if the conversion specifier begins with [] or [^], the right bracket character is in the scanlist and the next right bracket character is the matching right bracket that ends the specification. If a - character is in the scanlist and is not the first, nor the second where the

first character is a ^, nor the last character, the behavior is implementation-defined.

– The c specifier matches a sequence of characters (%) of the number specified by the field width (1 if no field width is present in the directive). The corresponding argument shall be a pointer to the first character of an array large enough to accept the sequence. No null character is added.
– The P specifier matches an implementation-defined set of sequences, which should be the same as the set of sequences that may be produced by the %P conversion of the sprintf function. The corresponding argument shall be a pointer to a pointer to void. The interpretation of the input item is implementation-defined; however, for any input item other than a value converted earlier during the same program execution, the behavior of the %P conversion is undefined.
– The n specifier indicates that no input is consumed. The corresponding argument shall be a pointer to integer that will contain the number of characters read from the input so far by this call to the sscanf function. Execution of a %n directive does not increment the assignment count returned at the completion of execution of the sscanf function.
– The percent sign (%) matches a single %; no conversion or assignment occurs. The complete conversion specification is %%.

Notes: The following are rules for the above specifiers

1. If a conversion specification is invalid, the behavior is undefined.
2. The conversion specifiers E, G and X are also valid and behave the same as, respectively, e, g and x.
3. If end-of-file is encountered during input, conversion is terminated.
4. If end-of-file occurs before any characters matching the current directive have been read (other than leading white-space, where permitted), execution of the current directive terminates with an input failure; otherwise, unless execution of the current directive is terminated with a matching failure, execution of the following directive (if any) is terminated with an input failure.
5. If conversion terminates on a conflicting input character, the offending input character is left unread on the input. Trailing white-space

(including new-line characters) is left unread unless matched by a directive. The success of literal matches and suppressed assignments is not directly determinable other than via the %n directive.

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

< 0string1 less than string2

= 0string1 identical to string2

> 0string1 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.

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:

ValueMeaning

< 0string1 less than string2

= 0string1 identical to string2

> 0string1 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, which consists 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.

tan, tanh function

Header file statement: `#include<math.h>`

Syntax: `double tan (double x);`

`double tanh (double x);`

Parameter	Description
x	Angle in radian

The `tan` and `tanh` functions find the tangent and hyperbolic tangent of x, respectively.

Return Value

The `tan` function returns the tangent result. The tanh function returns the hyperbolic tangent result.

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:

functionMacroDescription

tolowertolower Converts c to lowercase if appropriate

touppertoupper 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.

vsprintf function

Header file statement: `#include <stdio.h>`

`#include <stdarg.h>`

Syntax: `int vsprintf (char` *buffer, const char *format, va_list arg_ptr);

Parameter	Description
format	Format control
argptr	Pointer to list of arguments
buffer	Storage location for output
count	Maximum number of bytes

The `vsprintf` function formats data and outputs data to the memory pointed to by buffer. This functions are similar to its counterpart sprintf, but accepts a pointer to a list of arguments instead of an argument list.

The format argument has the same form and function as the format argument for the sprintf function; see sprintf for a description of format.

The argptr parameter has type va_list, which is defined in the include files STDARG.H. The argptr parameter points to a list of arguments that are converted and output according to the corresponding format specifications in the format.

Return Value

The return value for `vsprintf` is the number of characters written, not counting the terminating null character.

Initialization and Link Files

INITIALIZATION FILE

The following is the initialization file that is included with the eZ80 C-Compiler installation.

```
;*****************************************************************
;*          eZ80Boot: C Runtime Startup
;*          Copyright (c) ZiLOG, 1999
;*****************************************************************

;*****************************************************************
.           sect        " .bss                  "; In case no-one
else names it
;*****************************************************************
            .sect       ".startup"; This should be placed properly
            .def        _c_int0
            .def        __exit
            .ref        _main
            .ref        .BSS_BASE,.BSS_LENGTH
            .ref        .TOS

.OVERLAY    .equ        1   ; Overlay support?
.INITBSS    .equ    1   ; Zero the .bss section ?
.INITASCI   .equ        1   ; Initialize ASCI


;*********************************
; Program entry point
;*********************************

_c_int0:
            ld          sp,.TOS ; Setup SP
```

```
            .if .INITASCI

            ; Initialize ASCI0 to 57.6K, 8 data bits, no parity,
            ; 2 stop bits, no flow control.
            ld          a,%ff
            out0 (%46),a ; Port B AFSR, enable ASCI, CSIO
            ld          a,%80
            out0 (%1f),a ; CCR PHI = XTAL/1
            ld          a,%0
            out0 (%02),a     ; asci_cntlb0 = 0
            ld          a,8
            out0 (%1a),a ; asci_astc0l = 8
            ld          a,0
            out0 (%1b),a ; asci_astc0h = 0
            ld          a,%6c
            out0 (%12),a ; asci_asext0 = 0x6c
            in0 a,(%4)
            and         a,%fe
            out0 (%4),a  ; asci_stat0 = asci_stat0 and 0xfe
            ld          a,%65
            out0 (%0),a  ; asci_cntla0 = 0x65

            .endif


            .if         .INITBSS

;------ Initialize the .BSS section to zero

            ld          bc,.BSS_LENGTH; Check for non-zero length
            ld          a,0                     ; *
            cp          a,b                     ; *
            jr          nz,$f                   ; *
            cp          a,c                     ; *
            jr          z,_c_bss_done; .BSS is zero-length ...
$$:
            ld          hl,.BSS_BASE; [hl]=.bss
```

```
            ld          (hl),0
            dec         bc                                  ; 1st
byte's taken care of
            ld          a,b                                 ;
modify zero flag
            or          a,c
            jr          z,_c_bss_done; Just 1 byte ...
            ld          de,.BSS_BASE+1; [de]=.bss+1
            ldir
_c_bss_done:

            .endif                                  ; .INITBSS

;*******************************;
            .ifdef      .OVERLAY
            .ref        __eZ80OverlayInit
            call        __eZ80OverlayInit
            .endif
;*******************************;

;------ main()

            ld          hl,0                        ; hl=NULL
            push        hl                          ; argv[0] = NULL
            ld          ix,0
            add         ix,sp                       ; ix=&argv[0]
            push        ix                          ; &argv[0]
            push        hl                          ; argc==0
            nop
            call        _main                       ; main()
            pop         af                          ; clean the stack
            pop         af                          ; *
            pop         af                          ; *
__exit:
            jr          $                           ; ?
```

```
;*****************************************************************************
*
            .def        ___HUGE_VAL
___HUGE_VAL
            .long       80000000h
;*****************************************************************************
*
            .end
```

LINK FILE

The following is the linker initialization file that is included with the eZ80 C-Compiler installation.

```
************************************************
-a
-ohello
-mhello
assign .const rom
assign .startup rom
order .startup
range rom 08400h:0ffffh
define .TOS=highaddr of rom-1
define .BSS_BASE=base of .bss
define .BSS_LENGTH=length of .bss
define .CBAR=0BAh; Common/Bank Area Register (for C runtime)
eZ80boot.o
eZ80mmu.o
eZ80eval.o
hello.o
"..\lib\libc.lib"
"..\lib\lhf.lib"
```

MMU FILE

The following is the MMU initialization file that is included with the eZ80 C-Compiler installation.

```
;*****************************************************************
;*          eZ80mmu: C Runtime Overlay Manager
;*          Copyright (c) ZiLOG, 1999
;*****************************************************************

;*****************************************************************
.FCDEPTH    .equ        8    ; Maximum overlay call depth

            .def        __eZ80Overlay

            .define     .ovlhf,space=ROM
            .section .ovlhf
            jp          __eZ80Overlay


            .ref        .CBAR                   ; MMU Common/Bank
Area Register


;*****************************************************************
bbr         .equ        039h                    ; Bank Base
Register
cbar        .equ        03Ah                    ; Common/Bank Area
Register
;*****************************************************************
fcall       .struct
ret         .word
seg         .byte
fclen       .endstruct
fcstack     .tag        fcall
            .bss        fcstack,fclen*.FCDEPTH
            .bss        fcsp,2
;*****************************************************************
```

```
            .page
;***************************************************************
;           Initialize the overlay manager
;***************************************************************
            .def        __eZ80OverlayInit
__eZ80OverlayInit:
            ld          hl,fcstack; Prime the far call stack
pointer
            ld          (fcsp),hl; *
            ld          a,.CBAR                 ; Common/Bank Area
Register
            out0        (cbar),a; *
            ret                                 ; Done
;***************************************************************
            .page
;***************************************************************
; Far Call entry point
;---------------------------------------------------------------
-------
; Stack layout: [sp]->address of (RST p)+1
;***************************************************************

            .def        __eZ80Overlay
__eZ80Overlay:
            push        hl    ; Allocate 1 word for indirect call
            push        ix    ; Save frame pointer
            ld          ix,0  ; Establish our frame
            add         ix,sp ; *
            push        af     ; Save scratch
            push        hl     ; *
            push        de     ; *

            ld          l,(ix+4); [hl]=RSTp+1
            ld          h,(ix+5); *
            ld          a,(hl); (a)=Callee's Bank Base Register
            inc         hl     ; *
            ld          e,(hl) ; [de]=&(function to call)
```

```
            inc         hl      ; *
            ld          d,(hl) ; *
            inc         hl    ; [hl]=&(next sequential instruction)
            ld          (ix+2),e ; Patch in callee address
            ld          (ix+3),d ; *

            in0         e,(bbr) ; (e)=Caller's bank base register
            cp          a,e   ; Caller's BBR == Callee's BBR ?
            jr          nz,.stack ; No ... need to stack it ...

            ld          (ix+4),l; Patch in caller's return address
            ld          (ix+5),h; *
            jr          .common; Rejoin common

.stack:
            ld          (ix+4),<__eZ80OverlayRet ; Patch in kernel
            return address
            ld          (ix+5),>__eZ80OverlayRet; *
            ex          de,hl    ; (de)=Caller's return address
            ld          hl,(fcsp)  ; (hl)=Far call stack pointer
            ld          (hl),e     ; Save caller's return address
            inc         hl         ; *
            ld          (hl),d     ; *
            inc         hl        ; *
            in0         e,(bbr) ; (e)=Caller's bank base register
            ld          (hl),e ; Save caller's BBR
            inc         hl    ; *
            ld          (fcsp),hl; Update far call stack pointer
            out0        (bbr),a ; Map the callee into view

.common:
            pop         de                      ; Recover scratch
            pop         hl                      ; *
            pop         af                      ; *
            pop         ix                      ; Recover frame
pointer
```

```
            ret                                 ; Dispatch the callee
;*******************************************************************
            .page
;*******************************************************************
            .def        __eZ80OverlayRet
__eZ80OverlayRet:
            push        hl                      ; Save scratch
            push        af                      ; *
            push        de                      ; *
            ld          hl,(fcsp); Far call stack pointer
            dec         hl                      ; *
            ld          a,(hl) ; Caller's bank base register
            dec         hl                      ; *
            out0        (bbr),a                 ; Remap caller
            ld          d,(hl) ; Reload caller's return address
            dec         hl                      ; *
            ld          e,(hl)                  ; *
            ld          (fcsp),hl; Update far call stack pointer
            ex          de,hl ; (hl)=Caller's return address
            pop         de    ; Recover scratch
            pop         af                      ; *
            ex          (sp),hl ; Recover (hl); load caller's return
            ret                                 ; Back to caller
;*******************************************************************
            .end
```

ASCII Character Set

Table 9. 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
	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

Table 9. ASCII Character Set (Continued)

Graphic	Decimal	Hexadecimal	Comments
	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
(	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

Table 9. ASCII Character Set (Continued)

Graphic	Decimal	Hexadecimal	Comments
/	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
@	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

Table 9. ASCII Character Set (Continued)

Graphic	Decimal	Hexadecimal	Comments
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
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

Table 9. ASCII Character Set (Continued)

Graphic	Decimal	Hexadecimal	Comments
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
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

Table 9. ASCII Character Set (Continued)

Graphic	Decimal	Hexadecimal	Comments
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

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. #

Software Version

Manual Number

Host Computer Description/Type

ZiLOG, Inc.
System Test/Customer Support
910 E. Hamilton Ave., Suite 110, MS 4-2
Campbell, CA 95008
Fax Number: (408) 558-8536
Email: tools@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 Inter change.
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.
C-Compiler	A compiler program that is used to link and build files written in C, convert them into assembly and create a hex file that can be downloaded or run on a processor.
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
DSP	Digital Signal Processing. A specialized microprocessor 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.
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.
USART	Universal Synchronous/Asynchronous Receiver/Transmitter. Can handle synchronous as well as asynchronous transmissions.
USB	Universal Serial Bus.
USC	Universal Serial Controller.
UTB V	Use Test Box. A board or system to test a particular chip in an end-use application. Volt, Overflow Flag.
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.
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.