

# ZiLOG Z8 Software Development Tools

## **ANSI C Compiler User's and Reference Guide**

### **ZiLOG**

910 E. Hamilton Avenue  
Campbell, California 95008 USA

Sales and Customer Service:  
Toll-free in the U.S. and Canada (877) ZiLOGCS  
International: (512) 306-4169  
FAX: (512) 306-4072

The information in this document is subject to change without notice.

ZiLOG software products are copyrighted by and shall remain the property of ZiLOG. Use, duplication, or disclosure is subject to restrictions as stated in ZiLOG's applicable software license.

ZiLOG makes no warranty of any kind with regard to this material, including, but not limited to, implied warranties of merchantability and fitness for a particular purpose, *except* as stated in ZiLOG's applicable software license.

No part of this document may be copied or reproduced in any form without the prior written consent of ZiLOG.

Copyright © 2000 ZiLOG. All rights Reserved.

*Intel* is a trademark of Intel Corporation.

*MSDOS* is a trademark of Microsoft.

*NOHAU* is a trademark of NOHAU Corporation.

UM005601-Z8X0400

# Introduction

# Chapter 1

---

## 1.1 Requirements

The ZiLOG Z8 ANSI C Cross Compiler (**Z8CC**) requires an IBM PC or true compatible with:

DOS 5.0 or higher,

a hard drive, and

a 3 1/2" floppy drive.

The real mode version of the compiler requires at least 640K of RAM.

The protected mode version requires:

an 80386 or 80486 CPU and

2M bytes of RAM with at least 1M byte of extended memory free.

## 1.2 Installation

Insert the ZiLOG C Compiler CD into the PC's CD-ROM drive, and follow the installation prompts.

# Getting Started

# Chapter 2

---

This chapter discusses how to invoke, configure, and link with the Z8 C compiler (**Z8CC**).

## 2.1 Command Line Syntax

Once installed, the **Z8CC** compiler may be invoked by entering **Z8CC** followed by the C source file name. The syntax for the entire command line is:

**Z8CC** <switches><filename list>[<switches><filename list>]...

Each switch applies only to the file listed after the switch. Switches must be preceded by a forward slash (/) or a minus sign (-).

Wildcards may be used for filenames. Filenames without an extension are assumed to have a **.c** extension.

Files with **.obj** and **.lib** extensions will be included in the object module list of the link control file generated by the compiler driver.

## 2.2 Command Line Switches

The following is a description of the compiler command line switches.

- |                                      |   |
|--------------------------------------|---|
| <b>-ALIAS</b>                        | Enable alias checking. The compiler will assume that program variables can be aliased. See the <b>-NOALIAS</b> switch for further discussion. This is the default.  |
| <b>-ASM</b>                          | Assemble compiler generated assembly file. This switch results in the generation of an object module. The assembly file is deleted if no assemble errors are detected and the <b>-KEEPASM</b> switch is not given. The default is <b>-NOASM</b> . |
| <b>-ASMSW:"&lt;asm switches&gt;"</b> | Pass <asm switches> to the assembler when assembling the compiler generated assembly file.  |

**-ROM:"<low>-<high>"**

Select range of target ROM. Used in the generation of the link control file to specify target ROM to the linker.

**-XRAM: "<low>-<high>"**

Specify the range of external data memory.

**-DEFALL**

Default all file names. Disables the semicolon (;) capability on the command line.

**-DEBUG**

Generate symbol debug information. Results in the creation of a **.deb** file to be used by the symbolic debugger.

**-DEFINE:<symbol>=<value>**

Define a symbol and set it to the constant **<value>**. Equivalent to the C **#define** statement.

Example: **-define:DEBUG=0**      switch  
          **#define DEBUG 0**      C macro

**-EDATA: "<low>-<high>"**

Specifies the range of external data.

**-EMBEDDED**

Run the compiler in embedded mode. This switch allows the near, far, interrupt, and other embedded keywords to be used. The **Z8CC** compiler requires this switch for proper code generation.

**-EXPMAC**

Expand macros in displayed source lines when an error is detected. This is useful for debugging macro expansions. The default is **-NOEXPMAC**.

**-FPLIB**

Link with the floating-point emulation library.

**-HEXFILE:"<filename>"**

Define the name of the linker generated hexfile to be **<filename>**.

**-INTRINSICS:** Specifies that intrinsic functions are to be expanded to inline code.

**-INTSRC**

Integrate source code with the compiler assembly language output. This is the default.

- KEEPASM** Keep the compiler generated assembly file. The default is **-NOKEEPASM**.
- KEEPLNK** Keep the **.LNK** link control file. This file is created in order to link applications. The default is **-NOKEEPLNK**.
- KEEPLST** Keep the assembly listing file.
- LINK** Link after assembly.
- LINKSW:"<link switches>"**  
Pass **<link switches>** to the linker when linking an application.
- LIST** Generate a **.LIS** source listing file. The default is **-NOLIST**.
- LISTINC** Display included files in the compiler listing file.
- LOCALCSE** Turns on machine level common subexpression elimination optimization. This is the default.
- MAP** Generate a link map.
- MAXERRS= <count>**  
Specify the maximum number of errors allowed before aborting the compilation process. The default is 50.
- MODEL=<model>**  
Select the memory model, where **<model>** is **S** (small) and **L** (large). See Chapter titled “Advanced Topics”, for a discussion of the ZiLOG Z8 memory models. The default is Large Model.
- NOALIAS** Disable alias checking. Use of this switch can reduce the size of executable files and speed program execution. However, before specifying **-NOALIAS**, be sure that the program does not use aliases, either directly or indirectly.

An alias occurs when two variables can reference the same memory location. The following example illustrates an alias:

```
func{ }  
{  
    int x,*p;  
    p = &x; /* both “x” and “*p” refer to same location */  
    .  
    .  
    .  
}
```

If both **\*p** and **x** are used below the assignment, then malignant aliases exist and the **-NOALIAS** switch should not be used. Otherwise, alias is benign and the **-NOALIAS** switch may be used.

<b>-NOASM</b>	Do not assemble the compiler generated assembly file. This is the default.
<b>-NOBSS</b>	Do not put uninitialized static data in section 3, instead, put it in section 2 and initialize it at link time.
<b>-NODEFALL</b>	Turn off the <b>-DEFALL</b> switch and reenale the semicolon capability on the command line.
<b>-NOEXPMAC</b>	Do not expand macros in the compiler listing file. This is the default.
<b>-NOFPCP</b>	Generate calls to floating-point emulator.
<b>-NOFPLIB</b>	Do not link with floating-point emulation library.
<b>-NOINTSRC</b>	Do not integrate source code with the assembly language output. The default is <b>-INTSRC</b> .
<b>-NOKEEPASM</b>	Delete the compiler generated assembly file. This is the default.
<b>-NOLINK</b>	Do not link. This is the default.
<b>-NOLIST</b>	Do not produce a source listing. All errors are identified on the console. This is the default.
<b>-NOLISTINC</b>	Do not show included files in the compiler listing file.
<b>-NOMAP</b>	Do not generate a link map.

---

<b>-NONINTRINSICS:</b>	Specifies that intrinsic functions are not to be expanded to inline code.
<b>-NOOPT</b>	Disables optimization.
<b>-NOOPTLINK</b>	Disable optimized linkage and enable subroutines to be nested to any depth.
<b>-NOSDIOPT</b>	Disable span dependent instruction optimizations. In the event that assembly language statements inserted into the assembly file via the <b>#pragma asm</b> preprocessor directive cause out of range errors in the assembler, use this switch to disconnect it.
<b>-NOSTKCK</b>	Do not perform stack checking. This is the default.
<b>-NOSTRICT</b>	Do not flag warnings for obsolete features. This is the default.
<b>-OF:&lt;name&gt;</b>	Select object format of linker output. The default is OMF695 ( <b>.LOD</b> ).
<b>-OPTLINK</b>	Use a static frame for local variables and function arguments. This switch is required if the supplied runtime library will be used. Although this switch is not required in other cases, it will result in smaller, faster executables by minimizing use of the stack.
<b>-OPTSIZE</b>	Optimize code for size.
<b>-OPTSPEED</b>	Optimize code for speed. This is the default.
<b>-RDATA:"&lt;low&gt;-&lt;high&gt;"</b>	Define the range of addresses for internal RAM (the register file.)
<b>-SDIOPT</b>	Perform span dependent instruction optimization. This optimization results in branches generated by the compiler taking the shortest form possible. This is the default.
<b>-STARTUP:"&lt;fname&gt;"</b>	Define <b>&lt;fname&gt;</b> as the startup code object module.



**-STDINC:<path list>**

Set the path for the standard include files. This defines the location of include files using the **#include <file.h>** syntax. Multiple paths are separated by semicolons.

Example: **-STDINC:"c:\Z8rtl;c:\myinc"**

In this example, the compiler looks for the include file in

1. the default directory,
2. the directory **c:\Z8rtl**, and
3. the directory **c:\myinc**.

If the file is not found after searching the entire path, the compiler flags an error.

Omitting this switch tells the compiler to search only the current directory.

**-STKCK**

Perform stack checking.

**-STRICT**

Check for conformance to the **ANSI** standard and its obsolescent features. These include old style parameter type declarations, empty formal parameter lists, and calling functions with no prototype in scope. When any of these features are used a warning is flagged. The **Z8CC** compiler requires this switch for proper code generation since it makes use of a static frame.

**-STRABSZ:<size>**

Set the amount of memory to use to store strings and identifier names. The default and maximum are both 32000 bytes.

**-USRINC:<path list>**

Set the search path for user include files. This defines the location for include files using the **#include "file.h"** syntax. The paths must be separated by semicolons.

Example: **-USRINC:"c:\ZiLOGrtl;c:\myinc"**

In this example, the compiler looks for the include file in

1. the default directory,
2. the directory **c:\ZiLOGrtl**, and

3. the directory **c:\myinc**.

If the file is not found after searching the entire path, the compiler flags an error.

Omitting this switch tells the compiler to search only the current directory.

**-WATCH**      Display passes of the compiler as they are invoked.

## 2.3 Pragmas

Pragmas can permanently attach a compiler option to a specific file.

The following table shows some pragmas and their switch equivalents.

<b>Pragma</b>	<b>Switch Equivalent</b>
<b>#pragma alias</b>	<b>-ALIAS</b>
<b>#pramga fpcp</b>	<b>-FPCP</b>
<b>#pragma noalias</b>	<b>-NOALIAS</b>
<b>#pragma nobss</b>	<b>-NOBSS</b>
<b>#pragma nofpcp</b>	<b>-NOFPCP</b>
<b>#pragma noopt</b>	<b>-NOOPT</b>
<b>#pragma nooptlink</b>	<b>-NOOPTLINK</b>
<b>#pragma nosdiopt</b>	<b>-NOSDIOPT</b>
<b>#pragma nostkck</b>	<b>-NOSTKCK</b>
<b>#pragma nostrict</b>	<b>-NOSTRICT</b>
<b>#pragma optlink</b>	<b>-OPTLINK</b>
<b>#pragma optsize</b>	<b>-OPTSIZE</b>
<b>#pragma optspeed</b>	<b>-OPTSPEED</b>
<b>#pragma sdiopt</b>	<b>-SDIOPT</b>

<b>#pragma stkck</b>	<b>-STKCK</b>
<b>#pragma strict</b>	<b>-STRICT</b>

For more information about pragmas, see section titled “Embedded Assembly Language.”

## 2.4 Configuration Files

Configuration files hold switch settings so that the user does not have to type switches each time the compiler is invoked.

All configuration files must be named **Z8CC.CON**. The standard configuration file must be in the directory `\<root>\bin`, where `<root>` is the directory that holds the compiler. A user configuration file, if desired, must also be named **Z8CC.CON** and reside in the default directory.

A configuration file is made up of switches in exactly the same syntax as discussed in section titled "Command Line Switches." However, the configuration file may include multiple lines of switches. It can also include comments of the form `/* ... */` if they do not span more than one line. The following shows the supplied system configuration file.

```
-STDINC:"C:\Z8\INCLUDE"
/* set include path */
-OPTSPEED           /* optimizes for speed */
-INTSRC             /* source mixed with assembly language */
-MAXERRS:50         /* sets maximum error count to 50 */
-STKCK              /* enables stack overflow check */
-STRABSZ:3200       /* sets preprocessor string table */
-DEFALL             /* default all filenames -- disable “;” */
```

During the installation process, **install** adds switches to this file, depending on your response to configuration prompts.

## 2.5 Default Settings Hierarchy

The compiler gets switch settings from four sources, in this order:

1. the system configuration file
2. the user configuration file
3. the command line, and
4. **#pragma** statements in the source code.

Switches set in a source override switches in previous sources. For instance, if the source code includes the line

```
#pragma alias
```

but the command line includes the **-NOALIAS** flag, the compiler still checks for aliases.

## 2.6 Linking

The easiest way to link an application is by using the compiler itself. When the **-link** option is on, the compiler generates a link control file which includes all **.obj** files either created by the compiler or explicitly named on the command line, along with any **.lib** files. These files are combined with startup and runtime library code to create an executable program. The linking process may be controlled through the **-linksw**, **-startup**, **-data**, **-code**, **-map**, and **-of** compiler switches.

To link the files manually, the best way to create a link control file is to allow the compiler to generate it, then modify it to include the proper contents. To retain the link control file, include the **-keeplnk** switch. The compiler names the file **<fname>.lnk**, where **<fname>** is the name of the resulting executable file.

The executable file can be named by using the **-hexfile** switch. In the absence of this switch, the compiler names the executable file after the first file listed on the command line.

For example:

```
Z8CC main.c func1.c func2.obj library.lib
```

compiles **main.c** and **func1.c**, and links **main.obj**, **func1.obj**, **func2.obj**, and **library.lib** into the file **main.ihx**. If a link map is generated, it is held in **main.map**. If the **-keeplnk** switch is used, the file **main.lnk** contains the link control file.

```
Z8CC -hexfile:test *.c *.obj
```

compiles all files in the current directory with the **.c** extension and links them with all files in the current directory with the **.obj** extension. The resulting files are **test.ihx**, **test.map**, and **test.lnk**.

The **-OF** (output format) switch selects between Intel hex-records (the default) and Motorola S-records or OMF695 format. Intel hex-records have the extension **.ihx**. Motorola S-records use the extension **.hex**, OMF695 uses the **.lod** extension.

For example:

```
Z8CC -hexfile:test -of:"SREC" *.c
```

creates **test.hex**.

```
Z8CC -hexfile:test -of:"INTEL" *.c
```

creates **test.ihx**.

# Advanced Topics

## Chapter 3

---

### 3.1 Call Frames

Call frames (or activation records as they are sometimes called) hold the arguments, local variables, and other pertinent information of an instantiation of a procedure or function at runtime. The ZiLOG Z8 compiler supports two types of call frames: static and dynamic.

Dynamic call frames are the usual way of storing information about an instance of a function call. Dynamic call frames are most often allocated on the runtime stack. This feature enables a function to be reentrant or recursive.

Static call frames are a method of allocating call frames in static memory for architectures with little or no hardware stack support. Static call frames can significantly increase code efficiency on these architectures. However, since call frames are allocated statically in memory, the linker must receive information from the compiler about the structure of the entire application. That is, the compiler must supply a program call graph (PCG). Using the PCG a linker can selectively overlay static call frames to minimize the memory requirements of an application.

**NOTE:** Static call frames are not implemented in the huge memory model.

### 3.2 Memory Models

**Z8CC** supports four memory models:

small, and

large

Each of these models tell the compiler where to place data and code in memory to maximize the efficiency of an application. This feature allows the user to control where initialized and uninitialized data are stored, as well as where frames are allocated.

**NOTE:**

These memory models are supplied only as a convenience to the user. All defaults supplied for each memory model may be overridden using the **const**, **near**, and **far** storage class language extensions described in the section titled "Language Extensions."

### 3.2.1 Small Model

In the small model, all data are stored in onboard RAM in the range 20h to 7Fh (including static call frames), except for objects declared using the **const** or **far** storage class, which are stored in program ROM or XRAM, respectively. Dynamic call frames are allocated on the hardware stack in onboard RAM.

The small model is useful for small programs or programs that require very efficient code.

Small Model Memory Model

### 3.2.2 Large Model

In the large model, all data objects and call frames are allocated in internal RAM in the range of 80h to the top of internal RAM. Code and **const** data are stored in program ROM. All dynamic call frames utilize the hardware stack in internal RAM.

In the large model, there are no limitations on code, except that it must be less than 64 KB.

The large model is best suited for large programs that require large amounts of storage and don't have strict timing requirements. The RAM stack enables recursive routines to be used as well as routines that require large amounts of dynamic data storage.

## 3.3 Language Extensions

To give the user more control over the way the compiler allocates storage and to enhance its ability to handle common realtime constructs, the compiler implements several extensions the ANSI C standard. These language extensions are discussed below.

### 3.3.1 Interrupt Keyword

The **interrupt** keyword is a storage class that applies only to functions. It signifies that a function is to be an interrupt handler. This causes the compiler to produce code to preserve the current machine state at the beginning of the function and restore it on exit.

The following code fragment shows how a simple interrupt routine can be implemented with the **Z8CC**.

```
#include <Z8.h>

interrupt sint_hndlr(void)
{
    DI();      /* Disable interrupts */
              /* Process interrupt */
    EI();      /* Enable interrupts */
}

void main(void)
{
    SET_VECTOR(SINT,sint_hndlr); /* Set SINT hndlr address */
    EI();                       /* Enable interrupts */
    for(;;);                    /* Loop forever */
}
```

### 3.3.2 Reentrant Keyword

All call frames are assumed static by the compiler unless the **reentrant** storage class is used in the function declaration. The **reentrant** keyword causes the compiler to allocate the call frame dynamically. For all models the call frame will be in onboard RAM.

The following code fragment shows a reentrant routine with a dynamic call frame.

```
reentrant nf(int a,int b,int c)
{
    int x,y,z;

    ...
}
```

### 3.3.3 Near, Far, and Huge Keywords

The **near** and **far** keywords are storage class specifiers and are used to control the allocation of data objects by the compiler. They may be used on individual data objects similar to the **const** and **volatile** keywords in the ANSI standard. Used in this way, an object may be placed in onboard RAM or extended RAM using the **near** and **far** keywords. For example, the following code fragment allocates an array to be stored in onboard RAM in the range of 40h to



7Fh, one to be stored in internal RAM in the range of 80h to the top of internal RAM, and one to be stored in extended RAM.

```
near int na[10];
far int fa[10];
```

Similarly, a data object may be placed in program ROM using the following declaration.

```
const int ca[] = {0,1,2,3,4,5,6,7,8,9};
```

### 3.3.4 Far, Const, and Near Pointers

When the **near**, **const**, and **far** keywords are used in declarations, special care must be taken to ensure that pointers to those objects have the correct type. This is true also for the **const** and **volatile** specifiers in the ANSI standard. To properly access **near**, **far**, and **const** objects via a pointer, the type of the pointer must correctly specify the storage class specifiers for each indirection. Another way of looking at it is to say that for the Z8 there are four types of pointers:

near pointers,

far pointers,

const pointers

which point into onboard RAM from 40h to 7Fh, onboard RAM from 80h to the top of internal RAM, extended RAM, and program ROM respectively.

Below are a few examples of different pointer declarations.

char *p;	/*p is a pointer to a character dependent upon the model */
char far *fp;	/*fp is a pointer to a far char, fp is stored dependent upon the model */
char far * far fpf;	/* fpf is a pointer to a far char, fpf is stored in far memory */
char * near pn;	/* pn is a pointer to a char stored dependent upon the model, pn is stored in near memory. */
char near * far npf;	/* npf is a pointer to a near char, npf is stored in far memory */
char const * near cpn;	/* cpn is a pointer to a const char, cpn is stored in near memory. */

The rule of thumb is that **near**, **far**, **huge**, and **const** apply to the asterisk (\*) to their right. If there is no asterisk then it applies to the object itself.

## 3.4 Embedded Assembly Language

There are two methods of inserting assembly language within C code. The first uses the **#pragma** feature of ANSI C with the following syntax:

```
#pragma asm "<assembly line>"
```

**#pragma** may be inserted anywhere within the C source file. The contents of **<assembly line>** must be legal assembly language syntax. Note that the usual C escape sequences (i.e. **\n**, **\t**, **\r**, ...) are properly translated. Currently there is no processing done on the **<assembly line>**. Except for escape sequences it is passed through the compiler verbatim.

The second method of inserting assembly language is via the **asm** statement. The **asm** statement can appear only within the body of a function and takes the following form:

```
asm("<assembly line>");
```

As with the **#pragma asm** form no processing is performed on the assembly line except for the standard C escape sequences.

## 3.5 Reentrancy

### 3.5.1 Reentrant keyword and elipses (,...)

Reentrancy is the process of entering a block of code more than once before it is exited. Many compilers come with runtime libraries that are not reentrant. This may cause problems with some types of applications such as interrupt handlers and device drivers. The runtime library supplied with the compiler is written to be mostly reentrant. However, there are utilities defined in the ANSI standard that by definition are not reentrant.

These functions are:

<b>div</b>	<b>ldiv</b>	<b>rand</b>	<b>strtok</b>
<b>gmtime</b>	<b>localtime</b>	<b>srand</b>	

As long as these functions are not called from code designed to be reentrant, there should be no problem with the **Z8CC** runtime routines.

Though the **Z8CC** runtime library is written to be mostly reentrant it is shipped in a form that is not reentrant. This is because the default for the compiler is to use static call frames which use statically allocated memory to hold arguments and local variables. Using static call frames prohibits reentrancy but results in the smallest and most efficient code.

The only exception to the default use of static call frames is when a function has a prototype in scope that uses the ellipses (...) notation to signify a variable number of arguments. Since static call frames prohibit variable arguments all such functions are forced to have dynamic call frames and are therefore reentrant.

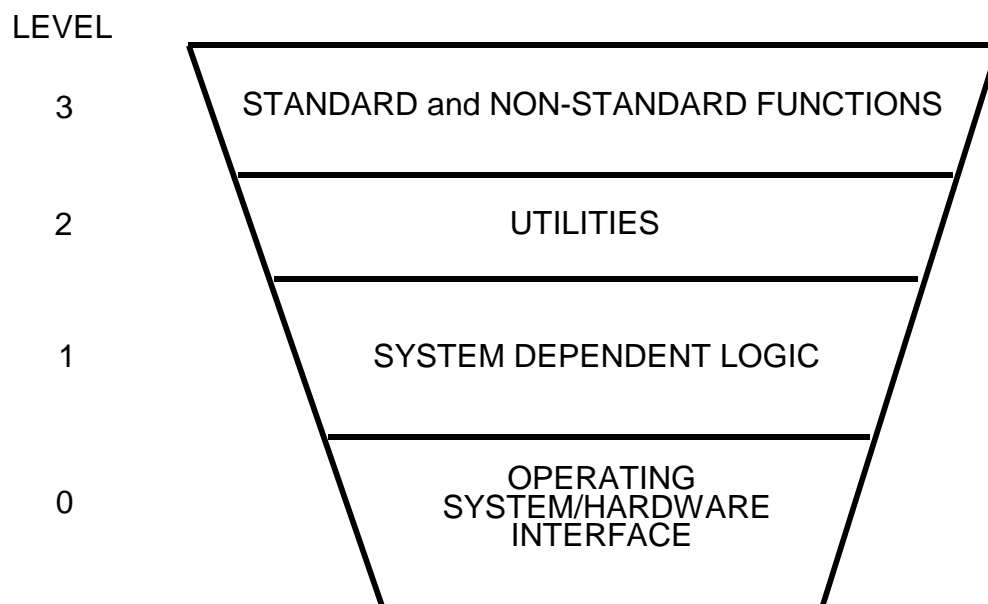
## 3.6 Register Variables

Register variables are assigned automatically by the compiler. However, variables and arguments can be dictated as register variables using the register keyword. R0-R15 and RR0-RR14 may be used as register variables.

## 3.7 Object Sizes

The following table gives the sizes of each basic type of object.

Type	Size	
Characters	8 bits	
Shorts	16 bits	
Integers	16 bits	
Longs	32 bits	
Float	32 bits	
Double	64 or 32 bits	



## 3.8 Alignment

When porting code and/or data from one system to another, it may be important to understand how objects are aligned in memory so that any padding injected by the compiler can be accounted for. The following paragraphs describe how each of the types of objects are aligned in memory.

Short, int, and long variables, including structs and arrays, are aligned on even byte boundaries.

## 3.9 Runtime Library Funnel

The runtime library is built in four layers. The topmost layer (layer 3) consists of the functions described in chapter titled " Functions by Name." Layer 2 interfaces layer 3 and layer 1, which consists of the logic specific to the operating system. Layer 0, the bottom layer, is what actually talks to the hardware.

The following are the routines making up level 0:

Function	File	Description
getch	sio.c	Get a character from the UART.
getche	sio.c	Get a character from the UART and echo it.
putch	sio.c	Put a character to the UART.
kbhit	sio.c	Test for character waiting at UART.
startup	startup.asm	Program startup code/ entry point.

## 4.1 Headers

Each library function is declared in a *header*. The **#include** preprocessor directive makes the header's contents available. The header declares a set of related functions plus any necessary types and additional macros needed to facilitate their use. All external identifiers declared in any of the headers are reserved, whether or not the associated header is included. All external identifiers and macro names that begin with an underscore are also reserved. If the program redefines a reserved external identifier, even with a semantically equivalent form, the behavior is indeterminant.

The standard headers are

<b>&lt;assert.h&gt;</b>	<b>&lt;math.h&gt;</b>	<b>&lt;stdio.h&gt;</b>
<b>&lt;ctype.h&gt;</b>	<b>&lt;setjmp.h&gt;</b>	<b>&lt;stdlib.h&gt;</b>
<b>&lt;float.h&gt;</b>	<b>&lt;stdarg.h&gt;</b>	<b>&lt;string.h&gt;</b>
<b>&lt;limits.h&gt;</b>	<b>&lt;stddef.h&gt;</b>	

The non standard headers are:

**<nonstdio.h>**  
**<Z8.h>**

If a file with any of these names, not provided as part of the compiler, is placed in any of the standard places for a source file to be included, the behavior is undefined.

Headers may be included in any order; each may be included more than once in a given scope, with no adverse effect. Headers are included in the code before the first reference to any of the functions it declares, types or macros it defines.

## 4.2 Use Of Library Functions

Below are six “rules of thumb” that apply to library functions. These rules apply unless explicitly stated otherwise in the detailed descriptions that follow.

If an argument to a function has an invalid value (such as a value outside the domain of the function, or a pointer outside the address space of the program, or a null pointer), the behavior is undefined.

Any function declared in a header may be implemented as a macro defined in the header, so a library function should not be declared explicitly if its header is included.

Any macro definition of a function can be suppressed locally by enclosing the name of the function in parentheses. Because the name is not followed by the left parenthesis the expansion of a macro cannot take place. For the same syntactic reason, it is permitted to take the address of a library function even if it is also defined as a macro.

Using **#undef** to remove any macro definition also ensures that the compiler refers to an actual function.

Any invocation of a library function that is implemented as a macro expands to code that evaluates each of its arguments exactly once. Each argument is fully protected by parentheses where necessary, so it is generally safe to use arbitrary expressions as arguments.

Since a library function can be declared without reference to any type defined in a header, it is also permissible to declare the function, explicitly or implicitly, and use it without including its associated header.

### Examples

The function **atoi** may be used in any of several ways:

by use of its associated header (possibly generating a macro expansion);

```
#include <stdlib.h>
const char *str;
/*...*/
i = atoi(str);
```

by use of its associated header (assuredly generating a true function reference)

```
#include <stdlib.h>
#undef atoi
const char *str;
/*...*/
i = atoi(str);
```

or

```
#include <stdlib.h>
const char *str;
/*...*/
i = (atoi)(str);
```

by explicit declaration

```
extern int atoi(const char *);
const char *str;
/*...*/
i = atoi(str);
```

by implicit declaration.

```
const char *str;
/*...*/
i = atoi(str);
```

## 4.3 Errors <errno.h>

The header <errno.h> defines several macros, all relating to the reporting of error conditions.

### MACROS:

**EDOM**       Expands to a distinct nonzero integral constant expression.

**ERANGE**      Expands to a distinct nonzero integral constant expression.

**errno** Expands to a modifiable lvalue that has type volatile **int**, the value of which is set to a positive error number by several library functions. It is initialized to zero at program startup, but is never set to zero by any library function. The value of **errno** may be set to nonzero by a library function call whether or not there is an error, provided the use of **errno** is not documented in the description of the function in the Standard.

Additional macro definitions, beginning with **E** and an uppercase letter, may also be specified by the implementation.

## 4.4 Standard Defines <stddef.h>

The following types and macros are defined in several headers referred to in the descriptions of the functions declared in that header, as well as the common standard header <stddef.h>.

### TYPES:

**ptrdiff\_t** Signed integral type of the result of subtracting two pointers.

**size\_t** Unsigned integral type of the result of the **sizeof** operator.

**wchar\_t** Integral type whose range of values can represent distinct codes for all members of the largest extended character set specified among the supported locales.

### MACROS:

**errno** Expands to a modifiable lvalue that has type volatile **int**, the value of which is set to a positive error number by several library functions. It is initialized to zero at program startup, but is never set to zero by any library function. The value of **errno** may be set to nonzero by a library function call whether or not there is an error, provided the use of **errno** is not documented in the description of the function in the Standard.

**NULL** Expands to a null pointer constant.

**offsetof** (*type, identifier*)  
The offset in bytes, from the beginning of a structure designated by *type*, to the member designated by *identifier*.



## 4.5 Diagnostics <assert.h>

The header <assert.h> declares one macro and refers to another macro.

### MACRO:

**NDEBUG** Is *not* defined by <assert.h>. If **NDEBUG** is defined at the point in the source file where <assert.h> is included, use of the **assert** macro will have no effect.

### FUNCTION:

**void assert**(int expression); Tests for logic error.

## 4.6 Character Handling <ctype.h>

The header <ctype.h> declares several functions useful for testing and mapping characters. In all cases the argument is an **int**, the value of which must be representable as an **unsigned char** or must equal the value of the macro **EOF**. If the argument has any other value, the behavior is undefined.

### FUNCTIONS:

#### Character testing

The functions in this section return non-zero (true) if, and only if, the value of the argument **c** conforms to that in the description of the function.

The term *printing character* refers to a member of a set of characters, each of which occupies one printing position on a display device; the term *control character* refers to a member of a set of characters that are not printing characters.

**int isalnum**(int c); Tests for alphanumeric character.  
**int isalpha**(int c); Tests for alphabetic character.  
**int iscntrl**(int c); Tests for control character.  
**int isdigit**(int c); Tests for decimal digit.  
**int isgraph**(int c); Tests for printable character except space.  
**int islower**(int c); Tests for lowercase character.  
**int isprint**(int c); Tests for printable character.  
**int ispunct**(int c); Tests for punctuation character.  
**int isspace**(int c); Tests for white-space character.  
**int isupper**(int c); Tests for uppercase character.  
**int isxdigit**(int c); Tests for hexadecimal digit.

### Character case mapping

**int tolower**(int c); Tests character and converts to lowercase if uppercase.

**int toupper**(int c); Tests character and converts to uppercase if lowercase.

## 4.7 Limits <float.h> and <limits.h>

The headers <float.h> and <limits.h> define several macros that expand to various limits and parameters.

### MACROS:

#### Sizes of integral types

<b>CHAR_BIT</b>	Maximum number of bits for smallest object that is not a bit-field( <b>byte</b> ).
<b>CHAR_MAX</b>	Maximum value for an object of type <b>char</b> .
<b>CHAR_MIN</b>	Minimum value for an object of type <b>char</b> .
<b>INT_MAX</b>	Maximum value for an object of type <b>int</b> .
<b>INT_MIN</b>	Minimum value for an object of type <b>int</b> .
<b>LONG_MAX</b>	Maximum value for an object of type <b>long int</b> .
<b>LONG_MIN</b>	Minimum value for an object of type <b>long int</b> .
<b>SCHAR_MAX</b>	Maximum value for an object of type <b>signed char</b> .
<b>SCHAR_MIN</b>	Minimum value for an object of type <b>signed char</b> .
<b>SHRT_MAX</b>	Maximum value for an object of type <b>short int</b> .
<b>SHRT_MIN</b>	Minimum value for an object of type <b>short int</b> .
<b>UCHAR_MAX</b>	Maximum value for an object of type <b>unsigned char</b> .
<b>UINT_MAX</b>	Maximum value for an object of type <b>unsigned int</b> .

<b>ULONG_MAX</b>	Maximum value for an object of type <b>unsigned long int</b> .
<b>USHRT_MAX</b>	Maximum value for an object of type <b>unsigned short int</b> .

If the value of an object of type **char** sign-extends when used in an expression, the value of **CHAR\_MIN** must be the same as that of **SCHAR\_MIN** and the value of **CHAR\_MAX** must be the same as that of **SCHAR\_MAX**.

If the value of an object of type **char** does not sign-extend when used in an expression, the value of **CHAR\_MIN** is 0 and the value of **CHAR\_MAX** is the same as that of **UCHAR\_MAX**.

Characteristics of floating types

<b>DBL_DIG</b>	Number of decimal digits of precision.
<b>DBL_MANT_DIG</b>	Number of base- <b>FLT_RADIX</b> digits in the floating-point mantissa.
<b>DBL_MAX</b>	Maximum representable floating-point numbers.
<b>DBL_MAX_EXP</b>	Maximum integer such that <b>FLT_RADIX</b> raised to that power approximates a floating-point number in the range of representable numbers.
<b>DBL_MAX_10_EXP</b>	Maximum integer such that 10 raised to that power approximates a floating-point number in the range of representable value $((\text{int})\log_{10}(\text{FLT\_MAX}), \text{etc})$ .
<b>DBL_MIN</b>	Minimum representable positive floating-point numbers.
<b>DBL_MIN_EXP</b>	Minimum negative integer such that <b>FLT_RADIX</b> raised to that power approximates a positive floating-point number in the range of representable numbers.
<b>DBL_MIN_10_EXP</b>	Minimum negative integer such that 10 raised to that power approximates a positive floating-point number in the range of representable values $((\text{int})\log_{10}(\text{FLT\_MIN}), \text{etc})$ .
<b>FLT_DIG</b>	Number of decimal digits of precision.

<b>FLT_MANT_DIG</b>	Number of base- <b>FLT_RADIX</b> digits in the floating-point mantissa.										
<b>FLT_MAX</b>	Maximum representable floating-point numbers.										
<b>FLT_MAX_EXP</b>	Maximum integer such that <b>FLT_RADIX</b> raised to that power approximates a floating-point number in the range of representable numbers.										
<b>FLT_MAX_10_EXP</b>	Maximum integer such that 10 raised to that power approximates a floating-point number in the range of representable value $((\text{int})\log_{10}(\text{FLT\_MAX}), \text{etc})$ .										
<b>FLT_MIN</b>	Minimum representable positive floating-point numbers.										
<b>FLT_MIN_EXP</b>	Minimum negative integer such that <b>FLT_RADIX</b> raised to that power approximates a positive floating-point number in the range of representable numbers.										
<b>FLT_MIN_10_EXP</b>	Minimum negative integer such that 10 raised to that power approximates a positive floating-point number in the range of representable values $((\text{int})\log_{10}(\text{FLT\_MIN}), \text{etc})$ .										
<b>FLT_RADIX</b>	Radix of exponent representation.										
<b>FLT_ROUNDS</b>	Rounding mode for floating-point addition.  <table><tr><td>-1</td><td>indeterminable</td></tr><tr><td>0</td><td>toward zero</td></tr><tr><td>1</td><td>to nearest</td></tr><tr><td>2</td><td>toward positive infinity</td></tr><tr><td>3</td><td>toward negative infinity</td></tr></table>	-1	indeterminable	0	toward zero	1	to nearest	2	toward positive infinity	3	toward negative infinity
-1	indeterminable										
0	toward zero										
1	to nearest										
2	toward positive infinity										
3	toward negative infinity										
<b>LDBL_DIG</b>	Number of decimal digits of precision.										
<b>LDBL_MANT_DIG</b>	Number of base- <b>FLT_RADIX</b> digits in the floating-point mantissa.										
<b>LDBL_MAX</b>	Maximum representable floating-point numbers.										
<b>LDBL_MAX_EXP</b>	Maximum integer such that <b>FLT_RADIX</b> raised to that power approximates a floating-point number in the range of representable numbers.										

**LDBL\_MAX\_10\_EXP**

Maximum integer such that 10 raised to that power approximates a floating-point number in the range of representable value  $((\text{int})\log_{10}(\text{FLT\_MAX}), \text{etc})$ .

**LDBL\_MIN**

Minimum representable positive floating-point numbers.

**LDBL\_MIN\_EXP**

Minimum negative integer such that **FLT\_RADIX** raised to that power approximates a positive floating-point number in the range of representable numbers.

**LDBL\_MIN\_10\_EXP**

Minimum negative integer such that 10 raised to that power approximates a positive floating-point number in the range of representable values  $((\text{int})\log_{10}(\text{FLT\_MIN}), \text{etc})$ .

## 4.8 Mathematics <math.h>

The header <math.h> declares several mathematical functions and defines three macros. The functions take **double**-precision arguments and return **double**-precision values. Integer arithmetic functions and conversion functions are discussed later.

### MACROS:

**EDOM**

Expands to distinct non-zero integral constant expressions.

**ERANGE**

Expands to distinct non-zero integral constant expressions.

**HUGE\_VAL**

Expands to a positive **double** expression, not necessarily representable as a **float**.

### 4.8.1 Treatment of error conditions

The behavior of each of these functions is defined for all values of its arguments. Each function must return as if it were a single operation, without generating any externally visible exceptions.

For all functions, a *domain error* occurs if an input argument to the function is outside the domain over which the function is defined. On a domain error, the function returns a specified value; the integer expression **errno** acquires the value of the macro **EDOM**.

Similarly, a *range error* occurs if the result of the function cannot be represented as a **double** value. If the result overflows (the magnitude of the result is so large that it cannot be represented in an object of the specified type), the function returns the value of the macro **HUGE\_VAL**, with the same sign as the correct value of the function; the integer expression **errno** acquires the value of the macro **ERANGE**. If the result underflows (the magnitude of the result is so small that it cannot be represented in an object of the specified type), the function returns zero.

## FUNCTIONS:

### Trigonometric

<b>double acos</b> (double x);	Calculates arc cosine of x.
<b>double asin</b> (double x);	Calculates arc sine of x.
<b>double atan</b> (double x);	Calculates arc tangent of x.
<b>double atan2</b> (double y, double x);	Calculates arc tangent of y/x.
<b>double cos</b> (double x);	Calculates cosine of x.
<b>double sin</b> (double x);	Calculates sine of x.
<b>double tan</b> (double x);	Calculates tangent of x.

### Hyperbolic

<b>double cosh</b> (double x);	Calculates hyperbolic cosine of x.
<b>double sinh</b> (double x);	Calculates hyperbolic sine of x.
<b>double tanh</b> (double x);	Calculates hyperbolic tangent of x.

### Exponential and logarithmic

<b>double exp</b> (double x);	Calculates exponential function of x.
<b>double frexp</b> (double value, int *exp);	Shows x as product of mantissa (the Value returned by frexp) and 2 to the n.
<b>double ldexp</b> (double x, int exp);	Calculates x times 2 to the exp.
<b>double log</b> (double x);	Calculates natural logarithm of x.
<b>double log10</b> (double x);	Calculates base 10 logarithm of x.
<b>double modf</b> (double value,	

`double *iptr;` Breaks down x into integer (the value returned by `modf`) and fractional (n) parts.

Power

**`double pow(double x, double y);`** Calculates x to the y.

**`double sqrt(double x);`** Finds square root of x.

Nearest integer, abs, rem

**`double ceil(double x);`** Finds integer ceiling of x.

**`double fabs(double x);`** Finds absolute value of x.

**`double floor(double x);`** Finds largest integer less than or equal to x.

**`double fmod(double x, double y);`** Finds floating-point remainder of x/y.

## 4.9 Non-Local Jumps <setjmp.h>

The header <setjmp.h> declares two functions and one type, for bypassing the normal function call and return discipline.

**TYPE:**

**`jmp_buf`** An array type suitable for holding the information needed to restore a calling environment.

**FUNCTIONS:**

Save calling environment

**`int setjmp(jmp_buf env);`** Saves a stack environment.

Restore calling environment

**`void longjmp(jmp_buf env, int val);`** Restores a saved stack environment.

## 4.10 Variable Arguments <stdarg.h>

The header <stdarg.h> declares a type and a function and defines two macros, for advancing through a list of arguments whose number and types are not known to the called function when it is translated.

A function may be called with a variable number of arguments of varying types. As described in Appendix A, “Function Definitions”, its parameter list contains one or more parameters. The rightmost parameter plays a special role in the access mechanism, and will be designated *parmN* in this description.

### TYPE:

**va\_list**     An array type suitable for holding information needed by the macro **va\_arg** and the function **va\_end**. The called function must declare a variable (referred to as **ap** in this section) having type **va\_list**. The variable **ap** may be passed as an argument to another function.

### 4.10.1 Variable argument list access macros and function

The **va\_start** and **va\_arg** macros described in this section must be implemented as macros, not as real functions. If **#undef** is used to remove a macro definition and obtain access to a real function, the behavior is undefined.

### FUNCTIONS:

**void va\_start(va\_list ap, parmN);**     Sets pointer to beginning of argument list.  
**type va\_arg (va\_list ap, type);**     Retrieves argument from list.  
**void va\_end(va\_list ap);**     Resets pointer.

## 4.11 Input/Output <stdio.h>

### 4.11.1 Introduction

The header <stdio.h> declares several functions for performing input and output.

### FUNCTIONS:

Formatted input/output

**int printf(const char \*format, ...);**     Writes formatted data to stdout.



**int scanf**(const char \*format, ...); Reads formatted data from stdin.  
**int sprintf**(char \*s, const char \*format, ...); Writes formatted data to string.  
**int sscanf**(const char \*s, const char \*format, ...); Reads formatted data from string.  
**int vsprintf**(char \*s, const char \*format, va\_list arg); Writes formatted data to a string.  
 Character input/output

**int getc**(FILE \*stream); Reads a character from stream (macro version).  
**int getchar**(void); Reads a character from stdin.  
**char \*gets**(char \*s); Reads a line from stdin.  
**int putchar**(int c); Writes a character to stdout (macro version).  
**int puts**(const char \*s); Writes a line to stdout.

## 4.12 General Utilities <stdlib.h>

The header <stdlib.h> declares several types, functions of general utility, and macros.

### TYPES:

**div\_t** Structure type that is the type of the value returned by the **div** function.  
**ldiv\_t** Structure type that is the type of the value returned by the **ldiv** function.  
**size\_t** Unsigned integral type of the result of the **sizeof** operator.  
**wchar\_t** Integral type whose range of values can represent distinct codes for all members of the largest extended character set specified among the supported locales.

### MACROS:

**EDOM** Expands to distinct nonzero integral constant expressions.  
**ERANGE** Expands to distinct nonzero integral constant expressions.

<b>HUGE_VAL</b>	Expands to a positive <b>double</b> expression, not necessarily representable as a <b>float</b> .
<b>MB_CUR_MAX</b>	Expands to a positive integer expression whose value is the maximum number of bytes in multi-byte character for the extended character set specified by the current locale.
<b>NULL</b>	Expands to a null pointer constant.
<b>RAND_MAX</b>	Expands to an integral constant expression, the value of which is the maximum value returned by the <b>rand</b> function.

## FUNCTIONS:

### String conversion

The functions **atof**, **atoi**, and **atol** do not affect the value of the macro **errno** on an error. If the result cannot be represented, the behavior is undefined.

**double atof**(const char \*nptr);    Converts string to float.  
**int atoi**(const char \*nptr);    Converts string to int.  
**long int atol**(const char \*nptr);    Converts string to long.  
**double strtod**(const char \*nptr,  
                char \*\*endptr);    Converts string pointed to by nptr  
  to a double.  
**long int strtol**(const char \*nptr,  
                char \*\*endptr, int base);    Converts string to a long decimal  
  integer that is equal to a number  
  with the specified radix.

### Pseudo-random sequence generation

**int rand**(void)    Gets a pseudo-random number.  
**void srand**(unsigned int seed);    Initializes pseudo-random series.

### Memory management

The order and contiguity of storage allocated by successive calls to the **calloc**, **malloc**, and **realloc** functions is unspecified. The pointer returned if the allocation succeeds is suitably aligned so that it may be assigned to a pointer to any type of object and then used to access such an object in the space allocated (until the space is explicitly freed or reallocated).

**void \*calloc**(size\_t nmemb,  
                size\_t size);    Allocates storage for array.

<b>void free</b> (void *ptr);	Frees a block allocated with <code>calloc</code> , <code>malloc</code> , or <code>realloc</code> .
<b>void *malloc</b> (size_t size);	Allocates a block.
<b>void *realloc</b> (void *ptr, size_t size);	Reallocates a block.

#### Searching and sorting utilities

<b>void *bsearch</b> (const void *key, const void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *));	Performs binary search.
<b>void qsort</b> (void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *));	Performs quick sort.

#### Integer arithmetic

<b>int abs</b> (int j);	Finds absolute value of integer value.
<b>div_t div</b> (int numer, int denom);	Computes integer quotient and remainder.
<b>long int labs</b> (long int j);	Finds absolute value of long integer value.
<b>ldiv_t ldiv</b> (long int numer, long int denom);	Computes long quotient and remainder.

## 4.13 String Handling <string.h>

### 4.13.1 String function conventions

The header <**string.h**> declares several functions useful for manipulating character arrays and other objects treated as character arrays. Various methods are used for determining the lengths of arrays, but in all cases a **char\*** or **void\*** argument points to the initial (lowest addressed) character of the array. If an array is written beyond the end of an object, the behavior is undefined.

#### TYPE:

<b>size_t</b>	Unsigned integral type of the result of the <b>sizeof</b> operator.
---------------	---

**MACRO:**

**NULL**      Expands to a null pointer constant.

**FUNCTIONS:**

## Copying

**void \*memcpy**(void \*s1, const  
void \*s2, size\_t n);      Copies a specified number of  
characters from one buffer to  
another.

**void \*memmove**(void \*s1, const  
void \*s2, size\_t n);      Moves a specified number of  
characters from one buffer to  
another.

**char \*strcpy**(char \*s1, const  
char \*s2);      Copies one string to another.

**char \*strncpy**(char \*s1, const  
char \*s2, size\_t n);      Copies n characters of one string  
to another.

## Concatenation

**char \*strcat**(char \*s1, const  
char \*s2);      Appends a string.

**char \*strncat**(char \*s1, const  
char \*s2, size\_t n);      Appends n characters of string.

## Comparison

The sign of the value returned by the comparison functions is determined by the sign of the difference between the values of the first pair of characters that differ in the objects being compared.

**int memcmp**(const void \*s1,  
const void \*s2, size\_t n);      Compares the first n characters.

**int strcmp**(const char \*s1,  
const char \*s2);      Compares two strings.

**int strncmp**(const char \*s1,  
const char \*s2, size\_t n);      Compares n characters of two strings.

## Search

<b>void *memchr</b> (const void *s, int c, size_t n);	Returns a pointer to the first occurrence, within a specified number of characters, of a given character in the buffer.
<b>char *strchr</b> (const char *s, int c);	Finds first occurrence of a given character in string.
<b>size_t strspn</b> (const char *s1, const char *s2);	Finds first occurrence of a character from a given character in string.
<b>char *strpbrk</b> (const char *s1, const char *s2);	Finds first occurrence of a character from one string to another.
<b>char *strrchr</b> (const char *s, int c);	Finds last occurrence of a given character in string.
<b>size_t strspn</b> (const char *s1, const char *s2);	Finds first substring from a given character set in string.
<b>char *strstr</b> (const char *s1, const char *s2);	Finds first occurrence of a given string in another string.
<b>char *strtok</b> (char *s1, const char *s2);	Finds next token in string.

## Miscellaneous

<b>void *memset</b> (void *s, int c, size_t n);	Uses a given character to initialize a specified number of bytes in the buffer.
<b>size_t strlen</b> (const char *s);	Finds length of string.

## 4.14 Non Standard IO Functions <nonstdio.h>

These are some non standard io functions whose use is non-standard.

### FUNCTIONS:

<b>int kbhit</b> ();	Returns number of characters waiting at the UART.
<b>int getch</b> ();	Returns the next key stroke at the UART.

<b>int getche</b> (void);	Returns the next character at the UART after echoing it.
<b>void init_uart</b> (void);	Initialize the Z8 UART.
<b>void putch</b> (char);	Sends a character to the UART.

## 4.15 Architecture Specific Functions <Z8.h>

The Z8 has many special function registers (SFRs) and bits that are difficult to manage. The **Z8.h** header file defines many useful macros and structures that simplify the utilization of the Z8 architecture.

EI()	Enable interrupts intrinsic
DI()	Disable interrupts intrinsic
SET_VECTOR(vec,func)	Set interrupt vector intrinsic where vec should be one of:

RESET	Reset vector
IRQ0	Interrupt request zero
IRQ1	Interrupt request one
IRQ2	Interrupt request two
IRQ3	Interrupt request three
IRQ4	Interrupt request four
IRQ5	Interrupt request five

SPL	Stack pointer bits 7-0
SPH	Stack pointer bits 15-8
RP	Register pointer
FLAGS	Program control flags
IMR	Interrupt mask register
IRQ	Interrupt request register
IPR	Interrupt priority register
P01M	Ports 0 & 1 mode
P2M	Port 2 mode
P3M	Port 3 mode
PRE0	T0 prescaler
T0	Timer/counter 0
PRE1	T1 prescaler
T1	Timer/counter 1
SIO	Serial I/O
P0	Port 0
P1	Port 1
P2	Port 2
P3	Port 3

SET_REG(regnum,val)	Set working register intrinsic; regnum must be one of: R0,R1,R2,...R15
---------------------	--

SET\_REG(regnum,val) Set working register intrinsic; regnum must be one of: R0,R1,R2,...R15

SET\_RREG(regnum,val) Set working register pair intrinsic; regnum must be one of: RR0,RR2,RR4,...R14

GET\_REG(regnum) Get value of a working register; regnum must be one of: R0,R1,R2,...R15

GET\_RREG(regnum) Get value of a working register pair; regnum must be one of: RR0,RR2,RR4,...R14

This chapter contains all functions by name in alphabetical order for quick reference. Each section contains a synopsis, description, and a return value of a function. Note that non-standard functions are marked as such and therefore should be avoided if portable code is desired.

The synopsis gives the calling sequence of the function and any include files necessary. The description discusses the semantics of the function and the return paragraph describes the value returned by the function, if any.

Also included in this chapter are the variables made available for use by the runtime library which may be useful to the user. All of these variables are system dependent and are marked as non-standard and should be avoided if portable code is desired.



## 5.1 **abs**

### **Synopsis**

```
#include <stdlib.h>  
int abs(int j);
```

### **Description**

The **abs** function computes the absolute value of an integer **j**. If the result cannot be represented, the behavior is undefined.

### **Returns**

The **abs** function returns the absolute value.

## 5.2 **acos**

### **Synopsis**

```
#include <math.h>
double acos(double x);
```

### **Description**

The **acos** function computes the principal value of the arc cosine of **x**. A domain error occurs for arguments not in the range  $[-1,+1]$ .

### **Returns**

The **acos** function returns the arc cosine in the range  $[0,\pi]$ .

## 5.3 asin

### Synopsis

```
#include <math.h>
double asin(double x);
```

### Description

The **asin** function computes the principal value of the arc sine of **x**. A domain error occurs for arguments not in the range  $[-1,+1]$ .

### Returns

The **asin** function returns the arc sine the range  $[-\pi/2,+\pi/2]$ .

## 5.4 assert

### Synopsis

```
#include <assert.h>
void assert(int expression);
```

### Description

The **assert** macro puts diagnostics into programs. When it is executed, if **expression** is false (that is, evaluates to zero), the **assert** macro writes information about the particular call that failed (including the text of the argument, the name of the source file, and the source line number - the latter are respectively the values of the preprocessing macros **\_\_FILE\_\_** and **\_\_LINE\_\_**) on the standard error file. It then calls the **abort** macro.

### Returns

If **expression** is true (that is, evaluates to non-zero), the **assert** macro returns no value.

## 5.5 atan

### Synopsis

```
#include <math.h>
double atan(double x);
```

### Description

The **atan** function computes the principal value of the arc tangent of **x**.

### Returns

The **atan** function returns the arc tangent in the range  $(-\pi/2, +\pi/2)$ .

## 5.6 atan2

### Synopsis

```
#include <math.h>
double atan2(double y, double x);
```

### Description

The **atan2** function computes the principal value of the arc tangent of **y/x**, using the signs of both arguments to determine the quadrant of the return value. A domain error occurs if both arguments are zero.

### Returns

The **atan2** function returns the arc tangent of **y/x**, in the range  $[-\pi, +\pi]$ .

## 5.7 atof

### Synopsis

```
#include <stdlib.h>
double atof(char *nptr);
```

### Description

The **atof** function converts the string pointed to by **nptr** to **double** representation. Except for the behavior on error, it is equivalent to

```
strtod(nptr, (char **)NULL)
```

### Returns

The **atof** function returns the converted value.

## 5.8 atoi

### Synopsis

```
#include <stdlib.h>
int atoi(char *nptr);
```

### Description

The **atoi** function converts the string pointed to by **nptr** to **int** representation. Except for the behavior on error, it is equivalent to

```
(int)strtol(nptr, (char **)NULL, 10)
```

### Returns

The **atoi** function returns the converted value.



## 5.9 atol

### Synopsis

```
#include <stdlib.h>
long int atol(char *nptr);
```

### Description

The **atol** function converts the string pointed to by **nptr** to **long int** representation. Except for the behavior on error, it is equivalent to

```
strtol(nptr, (char **)NULL, 10)
```

### Returns

The **atol** function returns the converted value.

## 5.10 bsearch

### Synopsis

```
#include <stdlib.h>
void *bsearch(void *key, void *base, size_t nmemb, size_t size,
int (*compar)(void *, void *));
```

### Description

The **bsearch** function searches an array of **nmemb** objects, the initial member of which is pointed to by **base**, for a member that matches the object pointed to by **key**. The size of each object is specified by **size**.

The array must have been previously sorted in ascending order according to a comparison function pointed to by **compar**, which is called with two arguments that point to the objects being compared. The function must return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second.

### Returns

The **bsearch** function returns a pointer to the matching member of the array, or a null pointer if no match is found.

## 5.11 calloc

### Synopsis

```
#include <stdlib.h>
void *calloc(size_t nmemb, size_t size);
```

### Description

The **calloc** function allocates space for an array of **nmemb** objects, each of whose size is **size**. The space is initialized to all bits zero.

### Returns

The **calloc** function returns a pointer to the start (lowest byte address) of the allocated space. If the space cannot be allocated, or if **nmemb** or **size** is zero, the **calloc** function returns a null pointer.

## 5.12 **ceil**

### **Synopsis**

```
#include <math.h>
double ceil(double x);
```

### **Description**

The **ceil** function computes the smallest integer not less than **x**.

### **Returns**

The **ceil** function returns the smallest integer not less than **x**, expressed as a double.

## 5.13 **cos**

### **Synopsis**

```
#include <math.h>
double cos(double x);
```

### **Description**

The **cos** function computes the cosine of **x** (measured in radians). A large magnitude argument may yield a result with little or no significance.

### **Returns**

The **cos** function returns the cosine value.

## 5.14 cosh

### Synopsis

```
#include <math.h>
double cosh(double x);
```

### Description

The **cosh** function computes the hyperbolic cosine of **x**. A range error occurs if the magnitude of **x** is too large.

### Returns

The **cosh** function returns the hyperbolic cosine value.

## 5.15 DI

### Synopsis

```
#include <Z8.h>
void DI(void);
```

### Description

The **DI** intrinsic is a Z8 family-specific macro that embeds assembly language instructions into C source code which, when executed, disables all interrupts. The following code fragment shows its use.

```
#include <Z8.h>

extern interrupt void reset(void);

void main(void)
{
    SET_VECTOR(RESET,reset); /* setup reset vector */
    DI();                   /* disable interrupts */
    ...                     /* initialize system */
    EI();                   /* enable interrupts */
}
```

NOTE: The **DI** intrinsic is a nonstandard feature of the **Z8CC** compiler.

## 5.16 div

### Synopsis

```
#include <stdlib.h>
div_t div(int numer, int denom);
```

### Description

The **div** function computes the quotient and remainder of the division of the numerator **numer** by the denominator **denom**. If the division is inexact, the sign of the quotient is that of the mathematical quotient, and the magnitude of the quotient is the largest integer less than the magnitude of the mathematical quotient. If the result cannot be represented, the behavior is undefined.

### Returns

The **div** function returns a structure of type **div\_t**, comprising both the quotient and the remainder. The structure must contain the following members, in either order.

```
int quot; /* quotient */
int rem; /* remainder */
```



## 5.17 EI

### Synopsis

```
#include <Z8.h>
void EI(void);
```

### Description

The **EI** macro is an Z8 family specific macro that embeds assembly language instructions into C source code which when executed enable all interrupts. The following code fragment shows its use.

```
#include <Z8.h>

extern interrupt void reset(void);

void main(void)
{
    SET_VECTOR(RESET,reset); /* setup reset vector */
    DI();                   /* disable interrupts */
    ...                     /* initialize system */
    EI();                   /* enable interrupts */
}
```

NOTE: The **EI** intrinsic is a nonstandard feature of the **Z8CC** compiler.

## 5.18 exp

### Synopsis

```
#include <math.h>
double exp(double x);
```

### Description

The **exp** function computes the exponential function of **x**. A range error occurs if the magnitude of **x** is too large.

### Returns

The **exp** function returns the exponential value.

## 5.19 fabs

### Synopsis

```
#include <math.h>
double fabs(double x);
```

### Description

The **fabs** function computes the absolute value of a floating-point number **x**.

### Returns

The **fabs** function returns the absolute value of **x**.

## 5.20 floor

### Synopsis

```
#include <math.h>
double floor(double x);
```

### Description

The **floor** function computes the largest integer not greater than **x**.

### Returns

The **floor** function returns the largest integer not greater than **x**, expressed as a double.

## 5.21 fmod

### Synopsis

```
#include <math.h>
double fmod(double x, double y);
```

### Description

The **fmod** function computes the floating-point remainder of **x/y**. If the quotient of **x/y** cannot be represented, the behavior is undefined.

### Returns

The **fmod** function returns **x** if **y** is zero. Otherwise it returns the value **f**, which has the same sign as **x**, such that  $x - i * y + f$  for some integer *i*, where the magnitude of **f** is less than the magnitude of **y**.

## 5.22 free

### Synopsis

```
#include <stdlib.h>
void free(void *ptr);
```

### Description

The **free** function causes the space pointed to by **ptr** to be deallocated, that is, made available for further allocation. If **ptr** is a null pointer, no action occurs. Otherwise, if the argument does not match a pointer earlier returned by the **calloc**, **malloc**, or **realloc** function, or if the space has been deallocated by a call to **free** or **realloc**, the behavior is undefined. If freed space is referenced, the behavior is undefined.

### Returns

The **free** function returns no value.

## 5.23 frexp

### Synopsis

```
#include <math.h>
double frexp(double value, int *exp);
```

### Description

The **frexp** function breaks a floating-point number into a normalized fraction and an integral power of 2. It stores the integer in the **int** object pointed to by **exp**.

### Returns

The **frexp** function returns the value **x**, such that **x** is a **double** with magnitude in the interval  $[1/2, 1]$  or zero, and **value** equals **x** times 2 raised to the power **\*exp**. If **value** is zero, both parts of the result are zero.

## 5.24 getch

### Synopsis

```
#include <sio.h>
int getch(void);
```

### Description

The **getch** function waits for the next character to appear at the serial port, and returns its value. This function does not wait for end-of-line to return as **getchar** does. **getch** does not echo the character received.

### Return

Returns the next character that appears at the serial.

**NOTE** Use of this function is non-standard.



## 5.25 getche

### Synopsis

```
#include <sio.h>
int getche(void);
```

### Description

The **getche** function waits for the next character to appear at the serial port, echoes it, and returns its value. This function does not wait for end-of-line to return as **getchar** does.

### Return

Returns the next character that appears at the serial port.

**NOTE** Use of this function is non-standard.

## 5.26 getchar

### Synopsis

```
#include <stdio.h>
int getchar(void);
```

### Description

The **getchar** function is equivalent to **getc** with the argument **stdin**.

### Returns

The **getchar** function returns the next character from the input stream pointed to by **stdin**. If the stream is at end-of-file, the end-of-file indicator for the stream is set and **getchar** returns **EOF**. If a read error occurs, the error indicator for the stream is set and **getchar** returns **EOF**.

## 5.27 gets

### Synopsis

```
#include <stdio.h>
char *gets(char *s);
```

### Description

The **gets** function reads characters from the input stream pointed to by **stdin**, into the array pointed to by **s**, until end-of-file is encountered or a new-line character is read. Any new-line character is discarded, and a null character is written immediately after the last character read into the array.

### Returns

The **gets** function returns **s** if successful. If end-of-file is encountered and no characters have been read into the array, the contents of the array remain unchanged and a null pointer is returned. If a read error occurs during the operation, the array contents are indeterminate and a null pointer is returned.

## 5.28 isalnum

### Synopsis

```
#include <ctype.h>
int isalnum(int c);
```

### Description

The **isalnum** function tests for any character for which **isalpha** or **isdigit** is true.

## 5.29 isalpha

### Synopsis

```
#include <ctype.h>
int isalpha(int c);
```

### Description

The **isalpha** function tests for any character for which **isupper** or **islower** is true.

## 5.30 iscntrl

### Synopsis

```
#include <ctype.h>
int iscntrl(int c);
```

### Description

The **iscntrl** function tests for any control character.

## 5.31 isdigit

### Synopsis

```
#include <ctype.h>
int isdigit(int c);
```

### Description

The **isdigit** function tests for any decimal digit.

## 5.32 isgraph

### Synopsis

```
#include <ctype.h>
int isgraph(int c);
```

### Description

The **isgraph** function tests for any printing character except space (' ').



## 5.33 islower

### Synopsis

```
#include <ctype.h>
int islower(int c);
```

### Description

The **islower** function tests for any lowercase letter as defined in Appendix A, Character Sets.

## 5.34 isprint

### Synopsis

```
#include <ctype.h>
int isprint(int c);
```

### Description

The **isprint** function tests for any printing character including space (‘ ’).

## 5.35 ispunct

### Synopsis

```
#include <ctype.h>
int ispunct(int c);
```

### Description

The **ispunct** function tests for any printing character except space ( ' ') or a character for which **isalnum** is true.

## 5.36 isspace

### Synopsis

```
#include <ctype.h>
int isspace(int c);
```

### Description

The **isspace** function tests for the following white-space characters: space (' '), form feed ('\f'), new line ('\n'), carriage return ('\r'), horizontal tab ('\t'), or vertical tab ('\v').

## 5.37 isupper

### Synopsis

```
#include <ctype.h>
int isupper(int c);
```

### Description

The **isupper** function tests for any uppercase letter as defined in Appendix A, Character Sets.

## 5.38 isxdigit

### Synopsis

```
#include <ctype.h>
int isxdigit(int c);
```

### Description

The **isxdigit** function tests for any hexadecimal digit as defined in Appendix A, Integer Constants.

## 5.39 kbhit

### Synopsis

```
#include <sio.h>
int kbhit(void);
```

### Description

The **kbhit** function determines whether or not there is a character waiting at the serial port.

### Returns

Returns the number of characters waiting at the serial port. Zero means there are no characters waiting.

**NOTE:** Use of this function is non-standard.

## 5.40 labs

### Synopsis

```
#include <stdlib.h>
long int labs(long int j);
```

### Description

The **labs** function is similar to the **abs** function, except that the argument and the returned value each have type **long int**.



## 5.41 ldexp

### Synopsis

```
#include <math.h>
double ldexp(double x, int exp);
```

### Description

The **ldexp** function multiplies a floating-point number by an integral power of 2. A range error may occur.

### Returns

The **ldexp** function returns the value of **x** times 2 raised to the power of **exp**.

## 5.42 ldiv

### Synopsis

```
#include <stdlib.h>
ldiv_t ldiv(long int numer, long int denom);
```

### Description

The **ldiv** function is similar to the **div** function, except that the arguments and the members of the returned structure (which has type **ldiv\_t**) all have type **long int**.

## 5.43 log

### Synopsis

```
#include <math.h>
double log(double x);
```

### Description

The **log** function computes the natural logarithm of **x**. A domain error occurs if the argument is negative. A range error occurs if the argument is zero.

### Returns

The **log** function returns the natural logarithm.

## 5.44 log10

### Synopsis

```
#include <math.h>
double log10(double x);
```

### Description

The **log10** function computes the base-ten logarithm of **x**. A domain error occurs if the argument is negative. A range error occurs if the argument is zero.

### Returns

The **log10** function returns the base-ten logarithm.

## 5.45 longjmp

### Synopsis

```
#include <setjmp.h>
void longjmp(jmp_buf env, int val);
```

### Description

The **longjmp** function restores the environment saved by the most recent call to **setjmp** in the same invocation of the program, with the corresponding **jmp\_buf** argument. If there has been no such call, or if the function containing the call to **setjmp** has executed a return statement in the interim, the behavior is undefined.

All accessible objects have values as of the time **longjmp** was called, except that the values of objects of automatic storage class that do not have volatile type and have been changed between the **setjmp** and **longjmp** call are indeterminate.

As it bypasses the usual function call and returns mechanisms, the **longjmp** function must execute correctly in contexts of interrupts, signals and any of their associated functions. However, if the **longjmp** function is invoked from a nested signal handler (that is, from a function invoked as a result of a signal raised during the handling of another signal), the behavior is undefined.

### Returns

After **longjmp** is completed, program execution continues as if the corresponding call to **setjmp** had just returned the value specified by **val**. The **longjmp** function cannot cause **setjmp** to return the value 0; if **val** is 0, **setjmp** returns the value 1.

## 5.46 malloc

### Synopsis

```
#include <stdlib.h>
void *malloc(size_t size);
```

### Description

The **malloc** function allocates space for an object whose size is specified by **size**.

### Returns

The **malloc** function returns a pointer to the start (lowest byte address) of the allocated space. If the space cannot be allocated, or if **size** is zero, the **malloc** function returns a null pointer.

## 5.47 memchr

### Synopsis

```
#include <string.h>
void *memchr(void *s, int c, size_t n);
```

### Description

The **memchr** function locates the first occurrence of **c** (converted to an **unsigned char**) in the initial **n** characters of the object pointed to by **s**.

### Returns

The **memchr** function returns a pointer to the located character, or a null pointer if the character does not occur in the object.

## 5.48 memcmp

### Synopsis

```
#include <string.h>
int memcmp(void *s1, void *s2, size_t n);
```

### Description

The **memcmp** function compares the first **n** characters of the object pointed to by **s2** to the object pointed to by **s1**.

### Returns

The **memcmp** function returns an integer greater than, equal to, or less than zero, according as the object pointed to by **s1** is greater than, equal to, or less than the object pointed to by **s2**.



## 5.49 memcpy

### Synopsis

```
#include <string.h>
void *memcpy (void *s1, void *s2, size_t n);
```

### Description

The **memcpy** function copies **n** characters from the object pointed to by **s2** into the object pointed to by **s1**. Copying between objects that overlap must take place correctly.

### Returns

The **memcpy** function returns the value of **s1**.

## 5.50 memmove

### Synopsis

```
#include <string.h>
void *memmove (void *s1, void *s2, size_t n);
```

### Description

The **memmove** function moves *n* characters from the object pointed to by **s2** into the object pointed to by **s1**. If the two regions overlap the behavior is undefined.

### Returns

The **memmove** function returns the value of **s1**.

## 5.51 **memset**

### Synopsis

```
#include <string.h>
void *memset(void *s, int c, size_t n);
```

### Description

The **memset** function copies the value of **c** (converted to an **unsigned char**) into each of the first **n** characters of the object pointed to by **s**.

### Returns

The **memset** function returns the value of **s**.

## 5.52 modf

### Synopsis

```
#include <math.h>
double modf(double value, double *iptr);
```

### Description

The **modf** function breaks the argument **value** into integral and fractional parts, each of which has the same sign as the argument. It stores the integral part as a **double** in the object pointed to by **iptr**.

### Returns

The **modf** function returns the signed fractional part of **value**.

## 5.53 pow

### Synopsis

```
#include <math.h>
double pow(double x, double y);
```

### Description

The **pow** function computes **x** raised to the power of **y**. A domain error occurs if **x** is zero and **y** is less than or equal to zero, or if **x** is negative and **y** is not an integer. A range error may also occur.

### Returns

The **pow** function returns the value of **x** raised to the power **y**.

## 5.54 printf

### Synopsis

```
#include <stdio.h>
int printf(char *format, ...);
```

### Description

The **printf** function writes output to the stream pointed to by **stdout**, under control of the string pointed to by **format** that specifies how subsequent arguments are converted for output.

A format string contains two types of objects: plain characters, which are copied unchanged to **stdout**, and conversion specifications, each of which fetch zero or more subsequent arguments. The results are undefined if there are insufficient arguments for the format. If the format is exhausted while arguments remain, the excess arguments are evaluated but otherwise ignored. The **fprintf** function returns when the end of the format string is encountered.

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

Zero or more *flags* that modify the meaning of the conversion specification.

An optional decimal integer specifying a minimum *field width*. If the converted value has fewer characters than the field width, it will be padded on the left (or right, if the left adjustment flag, described later, has been given) to the field width. The padding is with spaces unless the field width integer starts with a zero, in which case the padding is with zeros.

An optional *precision* that gives the minimum number of digits to appear for the **d**, **i**, **o**, **u**, **x**, and **X** conversions, the number of digits to appear after the decimal point for **e**, **E**, and **f** conversions, the maximum number of significant digits for the **g** and **G** conversions, or the maximum number of characters to be written from a string in **s** conversion. The precision takes the form of a period (.) followed by an optional decimal integer; if the integer is omitted, it is treated as zero. The amount of padding specified by the precision overrides that specified by the field width.

An optional **h** 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 must be converted to **short int** or **unsigned short int** before printing); 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; 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, it is ignored.

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 must appear 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

- The result of the conversion will be left-justified within the field.
- + The result of a signed conversion will always begin with a plus or a minus sign.

*space* If the first character of a signed conversion is not a sign, a space will be prepended to the result. If the *space* and **+** flags both appear, the *space* flag will be ignored.

**#** The result is to be converted to an “alternate form”. For **c, d, i, s,** and **u** conversions, the flag has no effect. For **o** conversion, it increases the precision to force the first digit of the result to be a zero. For **x** (or **X**) conversion, a non-zero result will always contain a decimal point, even if no digits follow the point (normally, a decimal point appears in the result of these conversions only if a digit follows it). For **g** and **G** conversions, trailing zeros will not be removed from the result, as they normally are.

The conversion specifiers and their meanings are:

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

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

**e,E** The double argument is converted in the style *[-]d.ddde+dd*, where there is one digit before the decimal point and the number of digits after it is equal to the precision; when the precision is missing, six digits are produced; if the precision is zero, no decimal point appears. The value is rounded to the appropriate number of digits. The **E** conversion specifier will produce a number with **E** instead of **e** introducing the exponent. The exponent always contains at least two digits. However, if the magnitude to be converted is greater than or equal to  $1\text{E}+100$ , additional exponent digits will be written as necessary.

**g,G** The double argument is converted in style **f** or **e** (or in style **E** in the case of a **G** conversion specifier), with the precision specifying the number of significant digits. The style used depends on the value converted; style **e** will be used only if the exponent resulting from the conversion is less than -4 or greater than the precision. Trailing zeros are removed from the result; a decimal point appears only if it is followed by a digit.

**c** The **int** argument is converted to an **unsigned char**, and the resulting character is written.



- s**      The argument is taken to be a (**const char \***) pointer to a string. Characters from the string are written up to, but not including, the terminating null character, or until the number of characters indicated by the precision are written. If the precision is missing it is taken to be arbitrarily large, so all characters before the first null character are written.
- p**      The argument is taken to be a (**const void \***) pointer to an object. The value of the pointer is converted to a sequence of hex digits.
- n**      The argument is taken to be an (**int \***) pointer to an integer into which is *written* the number of characters written to the output stream so far by this call to **fprintf**. No argument is converted.
- %**      A % is written. No argument is converted.

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

### Returns

The **printf** function returns the number of characters transmitted, or a negative value if an output error occurred.

## 5.55 **putch**

### **Synopsis**

```
#include <sio.h>
void putch(int c);
```

### **Description**

The **putch** function writes **c** to the serial port.

### **Returns**

The **putch** function returns nothing.

**NOTE:**The **putch** function is non-standard.

## 5.56 putchar

### Synopsis

```
#include <stdio.h>
int putchar(int c);
```

### Description

The **putchar** function writes **c** to the serial port.

### Returns

The **putchar** function returns the character written. If a write error occurs, **putchar** returns **EOF**.

## 5.57 puts

### Synopsis

```
#include <stdio.h>
int puts(char *s);
```

### Description

The **puts** function writes the string pointed to by **s** to the stream pointed to by **stdout**, and appends a new-line character to the output. The terminating null character is not written.

### Returns

The **puts** function returns **EOF** if an error occurs; otherwise it returns a nonnegative value.

## 5.58 qsort

### Synopsis

```
#include <stdlib.h>
void qsort(void *base, size_t nmemb, size_t size,
int (*compar)(void *, void *));
```

### Description

The **qsort** function sorts an array of **nmemb** objects, the initial member of which is pointed to by any **base**. The size of each object is specified by **size**.

The array is sorted in ascending order according to a comparison function pointed to by **compar**, which is called with two arguments that point to the objects being compared. The function must return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second.

If two members compare as equal, their order in the sorted array is unspecified.

### Returns

The **qsort** function returns no value.

## 5.59 rand

### Synopsis

```
#include <stdlib.h>
int rand (void)
```

### Description

The **rand** function computes a sequence of pseudo-random integers in the range 0 to **RAND\_MAX**.

### Returns

The **rand** function returns a pseudo-random integer.

## 5.60 realloc

### Synopsis

```
#include <stdlib.h>
void *realloc(void *ptr, size_t size);
```

### Description

The **realloc** function changes the size of the object pointed to by **ptr** to the size specified by **size**. The contents of the object will be unchanged up to the lesser of the new and old sizes. If **ptr** is a null pointer, the **realloc** function behaves like the **malloc** function for the specified size. Otherwise, if **ptr** does not match a pointer earlier returned by the **calloc**, **malloc**, or **realloc** function, or if the space has been deallocated by a call to the **free** or **realloc** function, the behavior is undefined. If the space cannot be allocated, the **realloc** function returns a null pointer and the object pointed to by **ptr** is unchanged. If **size** is zero, the **realloc** function returns a null pointer and, if **ptr** is not a null pointer, the object it points to is freed.

### Returns

The **realloc** function returns a pointer to the start (lowest byte address) of the possibly moved object.

## 5.61 scanf

### Synopsis

```
#include <stdio.h>
int scanf(char *format, ...);
```

### Description

The **scanf** function reads input from the stream pointed to by **stdin**, under control of the string pointed to by **format** that specifies the admissible input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the object to receive the converted input. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but otherwise ignored.

The format is composed of zero or more directives: one or more white-space characters; an ordinary character (not %); or a conversion specification. Each conversion specification is introduced by the character %. After the %, the following appear in sequence:

An optional assignment-suppressing character \*.

An optional decimal integer that specifies the maximum field width.

An optional **h**, **l** or **L** indicating the size of the receiving object. The conversion specifiers **d**, **i**, **n**, **o**, and **x** may be preceded by **h** to indicate that the corresponding argument is a pointer to **short int** rather than a pointer to **int**, or by **l** to indicate that it is a pointer to **long int**. Similarly, the conversion specifier **u** may be preceded by **h** to indicate that the corresponding argument is a pointer to **unsigned short int** rather than a pointer to **unsigned int**, or by **l** to indicate that it is a pointer to **unsigned long int**. Finally, the conversion specifiers **e**, **f**, and **g** may be preceded by **l** to indicate that the corresponding argument is a pointer to **double** rather than a pointer to **float**, or by **L** to indicate a pointer to **long double**. If an **h**, **l**, or **L** appears with any other conversion specifier, it is ignored.

A Character that specifies the type of conversion to be applied. The valid conversion specifiers are described below.



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

A directive composed of white space is executed by reading input up to the first non-white-space character (which remains unread), or until no more characters can be read. A white-space directive fails if no white-space character can be found.

A directive that is an ordinary character is executed by reading the next character of the stream. If the character differs from the one comprising the directive, the directive fails, and the character remains unread.

A directive that is a conversion specification defines a set of matching input sequences, as described below for each 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 from the stream, unless the specification includes an **n** specifier. An input item is defined as the longest sequence of input characters (up to any specified maximum field width) which is an initial subsequence of a matching sequence. The first character, if any, after the input item remains unread. If the length of the input item is zero, the execution of the directive fails. This condition is a matching failure, unless an error prevented input from the stream, in which case it is an input failure.

Except in the case of a **%** 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 conversion specifiers are valid:

- d** Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the **strtol** function with the value 10 for the **base** argument. The corresponding argument must be a pointer to integer.
- i** Matches an optionally signed integer, whose format is the same as expected for the subject sequence of the **strtol** function with the

value 0 for the **base** argument. The corresponding argument must be a pointer to integer.

- o** Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of the **strtoul** function with the value 8 for the **base** argument. The corresponding argument must be a pointer to integer.
- u** Matches an unsigned decimal integer, whose format is the same as expected for the subject sequence of the **strtoul** function with the value 10 for the **base** argument. The corresponding argument must be a pointer to unsigned integer.
- x** Matches an optionally signed hexadecimal integer, whose format is the same as expected for the subject sequence of the **strtoul** function with the value of 16 for the **base** argument. The corresponding argument must be a pointer to integer.
- e,f,g** Matches an optionally signed floating-point number, whose format is the same as expected for the subject string of the **strtod** function. The corresponding argument must be a pointer to floating.
- s** Matches a sequence of non-white-space characters. The corresponding argument must 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.
- [** Matches a sequence of expected characters (the *scanset*). The corresponding argument must 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 is the **format** string, up to and including the matching right bracket (]). The characters between the brackets (the **scanlist**) comprise the scanset, unless the character after the left bracket is a circumflex (^), in which case the scanset contains all characters that do not appear in the scanlist between the circumflex and the right bracket. As a special case, if the conversion specifier begins with [] or [^], the right bracket character is in the scanlist and next right bracket character is the matching right bracket that ends the specification. If a - character is in the scanlist and is neither the first nor the last character, the behavior is indeterminant.

- c** Matches a sequence of characters of the number specified by the field width (1 if no field width is present in the directive). The corresponding argument must be a pointer to the initial character of an array large enough to accept the sequence. No null character is added.
- p** Matches a hexadecimal number. The corresponding argument must be a pointer to a pointer to **void**.
- n** No input is consumed. The corresponding argument must be a pointer to integer into which is to be written the number of characters read from the input stream so far by this call to the **scanf** function. Execution of a **%n** directive does not increment the assignment count returned at the completion of execution of the **scanf** function.
- %** Matches a single %; no conversion or assignment occurs.

If a conversion specification is invalid, the behavior is undefined.

The conversion specifiers **e**, **g**, and **x** may be capitalized. However, the use of upper case is ignored.

If end-of-file is encountered during input, conversion is terminated. If end-of-file occurs before any characters matching the current directive have been read (other than leading white space, where permitted), execution of the current directive terminates with an input failure; otherwise, unless execution of the current directive is terminated with a matching failure, execution of the following directive (if any) is terminated with an input failure.

If conversion terminates on a conflicting input character, the offending input character is left unread in the input stream. Trailing white space (including new-line characters) is left unread unless matched by a directive. The success of literal matches and suppressed assignments is not directly determinable other than via the **%n** directive.

### Returns

The **scanf** function returns the value of the macro **EOF** if an input failure occurs before any conversion. Otherwise, the **scanf** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early conflict between an input character and the format.

## 5.62 setjmp

### Synopsis

```
#include<setjmp.h>
int setjmp(jmp_buf env);
```

### Description

The **setjmp** function saves its calling environment in its **jmp\_buf** argument, for later use by the **longjmp** function.

### Returns

If the return is from a direct invocation, the **setjmp** function returns the value zero. If the return is from a call to the **longjmp** function, the **setjmp** function returns a non-zero value.

## 5.63 SET\_REG

### Synopsis

```
#include <Z8.h>
void SET_REG(int regnum,int val);
```

### Description

The **SET\_REG** macro assigns a value to a working register. The compiler generates the proper assembly code to place the value of **val** into the register. At this point, the compiler may determine that the register is dead; if so, it optimizes the code. **regnum** must be one of the following:

R0,R1,R2,R3,R4,R5,R6,R7,R8  
R9,R10,R11,R12,R13,R14, or R15

### Example:

```
int a;
SET_REG(R0,a + 10);
```

**NOTE:**This function is nonstandard.

## 5.64 SET\_RREG

### Synopsis

```
#include <Z8.h>
void SET_RREG(int regnum,int val);
```

### Description

The **SET\_RREG** function assigns a value to a working register pair. The compiler generates the proper assembly code to place the value of **val** into the register. At this point, the compiler may determine that the register is dead; if so, it optimizes the code. **regnum** must be one of the following:

RR0,RR2,RR4,RR6,RR8,RR10,RR12, or RR14

### Example:

```
int a;
SET_RREG(RR0,a + 10);
```

**NOTE:**This function is nonstandard.

## 5.65 SET\_VECTOR

### Synopsis

```
#include <Z8.h>
void SET_VECTOR(int vectnum,void (*hndlr)(void));
```

### Description

The **SET\_VECTOR** intrinsic is a Z8 family-specific function that embeds assembly language into C source code to specify the address of an interrupt handler. **vectnum** is the interrupt vector number to be set. The vectors supported by **Z8CC** are:

RESET	IRQ0	IRQ1
IRQ2	IRA3	IRQ4
IRQ		

See Section 5.15 for more information on interrupt number vectors.

**hndlr** is the address of the interrupt handler. This functions must have the **interrupt** specifier for proper execution.

Since the interrupt vectors of the Z8 are in ROM, they cannot be initialized at runtime. Therefore, the **SET\_VECTOR** intrinsic works by switching to a special segment and placing the address of the interrupt handler at the proper address. This implementation does not generate executable code.

The following code fragment shows a typical use of the **SET\_VECTOR** intrinsic.

```
#include <Z8.h>

extern interrupt void reset(void);

void main(void)
{
    SET_VECTOR(RESET,reset); /* setup reset vector */
    DI();                   /* disable interrupts */
    ...                     /* initialize system */
    EI();                   /* enable interrupts */
}
```

**NOTE:**The **SET\_VECTOR** interrupt is a nonstandard feature of the **Z8CC** compiler.

## 5.66 sin

### Synopsis

```
#include <math.h>
double sin(double x);
```

### Description

The **sin** function computes the sine of **x** (measured in radians). A large magnitude argument may yield a result with little or no significance.

### Returns

The **sin** function returns the sine value.



## 5.67 sinh

### Synopsis

```
#include <math.h>
double sinh(double x);
```

### Description

The **sinh** function computes the hyperbolic sine of **x**. A range error occurs if the magnitude of **x** is too large.

### Returns

The **sinh** function returns the hyperbolic sine value.

## 5.68 sprintf

### Synopsis

```
#include <stdio.h>
int sprintf(char *s, char *format, ...);
```

### Description

The **sprintf** function is equivalent to **printf**, except that the argument **s** specifies an array into which the generated output is to be written, rather than to a stream. A null character is written at the end of the characters written; it is not counted as part of the returned sum.

### Returns

The **sprintf** function returns the number of characters written in the array, not counting the terminating null character.

## 5.69 sqrt

### Synopsis

```
#include <math.h>
double sqrt(double x);
```

### Description

The **sqrt** function computes the non-negative square foot of **x**. A domain error occurs if the argument is negative.

### Returns

The **sqrt** function returns the value of the square root.

## 5.70 srand

### Synopsis

```
#include <stdlib.h>
void srand(unsigned int seed);
```

### Description

The **srand** function uses the argument as a seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to **rand**. If **srand** is then called with the same seed value, the sequence of pseudo-random numbers will be repeated. If **rand** is called before any calls to **srand** have been made, the same sequence will be generated as when **srand** is first called with a seed value of 1.

### Returns

The **srand** function returns no value.

## 5.71 sscanf

### Synopsis

```
#include <stdio.h>
int sscanf(char *s, char *format, ...);
```

### Description

The **sscanf** function is equivalent to **scanf**, except that the argument **s** specifies a string from which the input is to be obtained, rather than from a stream. Reaching the end of the string is equivalent to encountering end-of-file for the **scanf** function.

### Returns

The **sscanf** function returns the value of the macro **EOF** if an input failure occurs before any conversion. Otherwise, the **sscanf** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early conflict between an input character and the format.

## 5.72 strcat

### Synopsis

```
#include <string.h>
char *strcat(char *s1, char *s2);
```

### Description

The **strcat** function appends a copy of the string pointed to by **s2** (including the terminating null character) to the end of the string pointed to by **s1**. The initial character of **s2** overwrites the null character at the end of **s1**.

### Returns

The **strcat** function returns the value of **s1**.

## 5.73 strchr

### Synopsis

```
#include <string.h>
char *strchr(char *s, int c);
```

### Description

The **strchr** function locates the first occurrence of **c** (converted to a **char**) in the string pointed to by **s**. The terminating null character is considered to be part of the string.

### Returns

The **strchr** function returns a pointer to the located character, or a null pointer if the character does not occur in the string.

## 5.74 strcmp

### Synopsis

```
#include <string.h>
int strcmp(char *s1, char *s2);
```

### Description

The **strcmp** function compares the string pointed to by **s1** to the string pointed to by **s2**.

### Returns

The **strcmp** function returns an integer greater than, equal to, or less than zero, according as the string pointed to by **s1** is greater than, equal to, or less than the string pointed to by **s2**.



## 5.75 strcpy

### Synopsis

```
#include <string.h>
char *strcpy (char *s1, char *s2);
```

### Description

The **strcpy** function copies the string pointed to by **s2** (including the terminating null character) into the array pointed to by **s1**. If copying takes place between objects that overlap, the behavior is undefined.

### Returns

The **strcpy** function returns the value of **s1**.

## 5.76 strcspn

### Synopsis

```
#include <string.h>
size_t strcspn(char *s1, char *s2);
```

### Description

The **strcspn** function computes the length of the initial segment of the string pointed to by **s1** which consists entirely of characters *not* from the string pointed to by **s2**. The terminating null character is not considered part of **s2**.

### Returns

The **strcspn** function returns the length of the segment.

## 5.77 strlen

### Synopsis

```
#include <string.h>
size_t strlen(char *s);
```

### Description

The **strlen** function computes the length of the string pointed to by **s**.

### Returns

The **strlen** function returns the number of characters that precede the terminating null character.

## 5.78 strncat

### Synopsis

```
#include <string.h>
char *strncat(char *s1, char *s2, size_t n);
```

### Description

The **strncat** function appends not more than *n* characters of the string pointed to by **s2** (not including the terminating null character) to the end of the string pointed to by **s1**. The initial character of **s2** overwrites the null character at the end of **s1**. A terminating null character is always appended to the result.

### Returns

The **strncat** function returns the value of **s1**.

## 5.79 strncmp

### Synopsis

```
#include <string.h>
int strncmp(char *s1, char *s2, size_t n);
```

### Description

The **strncmp** function compares not more than **n** characters from the string pointed to by **s1** to the string pointed to by **s2**.

### Returns

The **strncmp** function returns an integer greater than, equal to, or less than zero, according as the string pointed to by **s1** is greater than, equal to, or less than the string pointed to by **s2**.

## 5.80 strncpy

### Synopsis

```
#include <string.h>
char *strncpy(char *s1, char *s2, size_t n);
```

### Description

The **strncpy** function copies not more than *n* characters from the string pointed to by **s2** to the array pointed to by **s1**. If copying takes place between objects that overlap, the behavior is undefined.

If the string pointed to by **s2** is shorter than *n* characters, null characters are appended to the copy in the array pointed to by **s1**, until *n* characters in all have been written.

### Returns

The **strncpy** returns the value of **s1**.

## 5.81 **strpbrk**

### Synopsis

```
#include <string.h>
char *strpbrk(char *s1, char *s2);
```

### Description

The **strpbrk** function locates the first occurrence in the string pointed to by **s1** of any character from the string pointed to by **s2**.

### Returns

The **strpbrk** function returns a pointer to the character, or a null pointer if no character from **s2** occurs in **s1**.

## 5.82 strrchr

### Synopsis

```
#include <string.h>
char *strrchr(char *s, int c);
```

### Description

The **strrchr** function locates the last occurrence of **c** (converted to a **char**) in the string pointed to by **s**. The terminating null character is considered to be part of the string.

### Returns

The **strrchr** function returns a pointer to the character, or a null pointer if **c** does not occur in the string.



## 5.83 strspn

### Synopsis

```
#include <string.h>
size_t strspn(char *s1, char *s2);
```

### Description

The **strspn** function computes the length of the initial segment of the string pointed to by **s1** which consists entirely of characters from the string pointed to by **s2**.

### Returns

The **strspn** function returns the length of the segment.

## 5.84 strstr

### Synopsis

```
#include <string.h>
char *strstr(char *s1, char *s2);
```

### Description

The **strstr** function locates the first occurrence of the string pointed to by **s2** in the string pointed to by **s1**.

### Returns

The **strstr** function returns a pointer to the located string, or a null pointer if the string is not found.

## 5.85 strtod

### Synopsis

```
#include <stdlib.h>
double strtod(char *nptr, char **endptr);
```

### Description

The **strtod** function converts the string pointed to by **nptr** to **double** representation. The function recognizes an optional leading sequence of white-space characters (as specified by the **isspace** function), then an optional plus or minus sign, then a sequence of digits optionally containing a decimal point, then an optional letter **e** or **E** followed by an optionally signed integer, then an optional floating suffix. If an inappropriate character occurs before the first digit following the **e** or **E**, the exponent is taken to be zero.

The first inappropriate character ends the conversion. If **endptr** is not a null pointer, a pointer to that character is stored in the object **endptr** points to; if an inappropriate character occurs before any digit, the value of **nptr** is stored.

The sequence of characters from the first digit or the decimal point (whichever occurs first) to the character before the first inappropriate character is interpreted as a floating constant according to the rules of section, except that if neither an exponent part or a decimal point appears, a decimal point is assumed to follow the last digit in the string. If a minus sign appears immediately before the first digit, the value resulting from the conversion is negated.

### Returns

The **strtod** function returns the converted value, or zero if an inappropriate character occurs before any digit. If the correct value would cause overflow, plus or minus **HUGE\_VAL** is returned (according to the sign of the value), and the macro **errno** acquires the value **ERANGE**. If the correct value would cause underflow, zero is returned and the macro **errno** acquires the value **ERANGE**.

## 5.86 strtok

### Synopsis

```
#include <string.h>
char *strtok(char *s1, char *s2);
```

### Description

A sequence of calls to the **strtok** function breaks the string pointed to by **s1** into a sequence of tokens, each of which is delimited by a character from the string pointed to by **s2**. The first call in the sequence has **s1** as its first argument, and is followed by calls with a null pointer as their first argument. The separator string pointed to by **s2** may be different from call to call.

The first call in the sequence searches **s1** for the first character that is not contained in the current separator string **s2**. If no such character is found, there are no tokens in **s1**, and the **strtok** function returns a null pointer. If such a character is found, it is the start of the first token.

The **strtok** function then searches from there for a character that is contained in the current separator string. If no such character is found, the current token extends to the end of the string pointed to by **s1**, and subsequent searches for a token will fail. If such a character is found, it is overwritten by a null character, which terminates the current token. The **strtok** function saves a pointer to the following character, from which the next search for a token will start.

Each subsequent call, with a null pointer as the value of the first argument, starts searching from the saved pointer and behaves as described above.

### Returns

The **strtok** function returns a pointer to the first character of a token, or a null pointer if there is no token.

### Example

```
#include <string.h>
static char str[] = "?a???b, , #c";
char *t;
t = strtok(str, "?"); /* t points to the token "a" */
t = strtok(NULL, ","); /* t points to the token "b" */
t = strtok(NULL, "#"); /* t points to the token "c" */
t = strtok(NULL, "?"); /* t is a null pointer */
```

## 5.87 strtol

## Synopsis

```
#include <stdlib.h>
long int strtol(char *nptr, char **endptr, int base);
```

## Description

The **strtol** function converts the string pointed to by **nptr** to **long int** representation. The function recognizes an optional leading sequence of white-space characters (as specified by the **isspace** function), then an optional plus or minus sign, then a sequence of digits and letters, then an optional integer suffix.

The first inappropriate character ends the conversion. If **endptr** is not a null pointer, a pointer to that character is stored in the object **endptr** points to; if an inappropriate character occurs before the first digit or recognized letter, the value of **nptr** is stored.

If the value of **base** is 0, the sequence of characters from the first digit to the character before the first inappropriate character is interpreted as an integer constant according to the rules of section. If a minus sign appears immediately before the first digit, the value resulting from the conversion is negated.

If the value of **base** is between 2 and 36, it is used as the base for conversion. Letters from **a** (or **A**) through **z** (or **Z**) are ascribed the values 10 to 35; a letter whose value is greater than or equal to the value of base ends the conversion. Leading zeros after the optional sign are ignored, and leading **0x** or **0X** is ignored if the value of **base** is 16. If a minus sign appears immediately before the first digit or letter, the value resulting from the conversion is negated.

## Returns

The **strtol** function returns the converted value, or zero if an inappropriate character occurs before the first digit or recognized letter. If the correct value would cause overflow, **LONG\_MAX** or **LONG\_MIN** is returned (according to the sign of the value), and the macro **errno** acquires the value **ERANGE**.

## 5.88 tan

### Synopsis

```
#include <math.h>
double tan(double x);
```

### Description

The **tan** function returns the tangent of **x** (measured in radians). A large magnitude argument may yield a result with little or no significance.

### Returns

The **tan** function returns the tangent value.

## 5.89 tanh

### Synopsis

```
#include <math.h>
double tanh(double x);
```

### Description

The **tanh** function computes the hyperbolic tangent of **x**.

### Returns

The **tanh** function returns the hyperbolic tangent of **x**.

## 5.90 tolower

### Synopsis

```
#include <ctype.h>
int tolower(int c);
```

### Description

The **tolower** function converts an uppercase letter to the corresponding lowercase letter.

### Returns

If the argument is an uppercase letter, the **tolower** function returns the corresponding lowercase letter, if any; otherwise the argument is returned unchanged.



## 5.91 toupper

### Synopsis

```
#include <ctype.h>
int toupper(int c);
```

### Description

The **toupper** function converts a lowercase letter to the corresponding uppercase letter.

### Returns

If the argument is a lowercase letter, the **toupper** function returns the corresponding uppercase letter, if any; otherwise the argument is returned unchanged.

## 5.92 **va\_arg**

### Synopsis

```
#include <stdarg.h>
type va_arg (va_list ap, type);
```

### Description

The **va\_arg** macro expands to an expression that has the type and value of the next argument in the call. The parameter *ap* must be the same as the **va\_list** **ap** initialized by **va\_start**. Each invocation of **va\_arg** modifies **ap** so that successive arguments are returned in turn. The parameter *type* is a type name such that the type of a pointer to an object that has the specified type can be obtained simply by postfixing a *\** to *type*. If *type* disagrees with the type of the actual next argument (as promoted, according to the default argument conversions, into **int**, **unsigned int**, or **double**), the behavior is undefined.

### Returns

The first invocation of the **va\_arg** macro after that of the **va\_start** macro returns the value of the argument after that specified by *parmN*. Successive invocations return the values of the remaining arguments in succession.

## 5.93 va\_end

### Synopsis

```
#include <stdarg.h>
void va_end(va_list ap);
```

### Description

The **va\_end** function facilitates a normal return from the function whose variable argument list was referenced by the expansion of **va\_start** that initialized the **va\_list ap**. The **va\_end** function may modify **ap** so that it is no longer usable (without an intervening invocation of **va\_start**). If the **va\_end** function is not invoked before the return, the behavior is undefined.

### Returns

The **va\_end** function returns no value.

### Example

The function **f1** gathers into an array a list of arguments that are pointers to strings (but not more than **MAXARGS** arguments), then passes the array as a single argument to function **f2**. The number of pointers is specified by the first argument to **f1**.

```
#include <stdarg.h>
#define MAXARGS 31

void f1(int n_ptrs, ...)
{
    va_list ap;
    char *array[MAXARGS];
    int ptr_no = 0;
    if (n_ptrs > MAXARGS)
        n_ptrs = MAXARGS;
    va_start(ap, n_ptrs);
    while (ptr_no < n_ptrs)
        array[ptr_no++] = va_arg(ap, char *);
    va_end(ap);
    f2(n_ptrs, array);
}
```

Each call to **f1** must have in scope the definition of the function or a declaration such as

```
void f1(int, ...);
```

## 5.94 **va\_start**

### Synopsis

```
#include <stdarg.h>
void va_start(va_list ap, parmN);
```

### Description

The **va\_start** macro must be executed before any access to the unnamed arguments.

The parameter **ap** points to an object that has type **va\_list**. The parameter *parmN* is the identifier of the rightmost parameter in the variable parameter list in the function definition (the one just before the , ...). The **va\_start** macro initializes **ap** for subsequent use by **va\_arg** and **va\_end**.

### Returns

The **va\_start** macro returns no value.

## 5.95 vprintf

### Synopsis

```
#include <stdarg.h>
#include <stdio.h>
int vprintf(char *format, va_list arg);
```

### Description

The **vprintf** function is equivalent to **printf**, with the variable argument list replaced by **arg**, which has been initialized by the **va\_start** macro (and possibly subsequent **va\_arg** calls). The **vprintf** function does not invoke the **va\_end** function.

### Returns

The **vprintf** function returns the number of characters transmitted, or a negative value if an output error occurred.

## 5.96 vsprintf

### Synopsis

```
#include <stdarg.h>
#include <stdio.h>
int vsprintf(char *s, char *format, va_list arg);
```

### Description

The **vsprintf** function is equivalent to **sprintf**, with the variable argument list replaced by **arg**, which has been initialized by the **va\_start** macro (and possibly subsequent **va\_arg** calls). The **vsprintf** function does not invoke the **va\_end** function.

### Returns

The **vsprintf** function returns the number of characters written in the array, not counting the terminating null character.





## 6.1 .FRAME and .FCALL

As discussed in Section 4.1, static call frames require that a program call graph (PCG) be passed to the linker so that static frames may be overlaid. The compiler performs this function by using two assembler directives named **.FRAME** and **.FCALL**.

The **.FRAME** directive defines the beginning of the static call frame of a function. It continues to the next SEG directive and has the following form:

**.FRAME** <funcname>,<segname>,<space>

<funcname> is the name of the function being declared. For a **far** function, this name is prefixed with **\_n\_** or **\_f\_**.

<segname> is the name of the segment holding the local data.

<space> is the address space that holds the static frame.

Within an **.FRAME** segment, all functions called by the function must have a **.FCALL** directive so that the linker can calculate the PCG. The form of the **.FCALL** directive is as follows:

**.FCALL** <funcname>

<funcname> is the name of the function being called.

## 6.2 Call Frame Layouts

As discussed in Chapter 4, the compiler supports two types of call frames:

- static

- dynamic

Both static and dynamic frames are implemented as both near and far functions. This allows four separate ways of allocating storage for a given function. The following sections discuss each form.

### 6.2.1 Near Static Call Frames

Near static call frames are the default for small, compact, and medium memory models. They may also be declared manually as follows:

```
near void near_func(int a,int b)
{
    int x,y;
    ...
}
```

The code for a near static call frame function will have the form:

```
_near_func:
    call    __s_nstkck
    ...
    ret
    .FRAME  _near_func, $_near_func, RDATA
_0_near_func: DS 2
_1_near_func: DS 2
_1x:        DS 2
_1y:        DS 2
```

The call to a `__s_nstkck` (near stack check) routine is omitted if the **-nostkck** switch is used.

For all static call frames, the arguments and local variables are stored within the **.FRAME** segment. Arguments are named such that they can be uniquely identified within an application. Each argument name has the form:

**\_<arg #>\_<func name>**

**<arg #>** is the zero-based argument number allocated from left to right.  
**<func name>** is the name of the function that the argument belongs to.

Calling a static call frame function from assembly language requires that the assembly routine also have an **.FRAME** directive as well as an **.FCALL** directive for each function called. Then, passing the arguments to the callee is a

simple matter of generating an extern for each argument as using the naming conventions discussed above and moving the actual parameters into the memory locations identified by the symbolic argument names. In the example above, the following code fragment would constitute a call to function **func**.

```

XREF      _0_near_func
XREF      _1_near_func
XREF      _near_func

_near_call:

ld        _0_near_func,#0
ld        _0_near_func+1,#10
ld        _1_near_func,#0
ld        _1_near_func+1,20
call      _near_func

.FRAME    _near_call, $_near_call, RDATA
.FCALL    _near_func

```

## 6.2.2 Far Static Call Frames

Far static call frame functions are very similar in form to near static call frame functions. The only differences are that the function **\_s\_fstkck** is called for a far stack check, the frame is stored in external memory as opposed to onboard memory, and the far flag of the **.FRAME** directive must be a 1.

The definition and calling of a far static call frame take the form:

```

__far_func:
    call    _s_fstkck

    ret
    .FRAME  _far_func, _f_far_func, XDATA
_0_far_func: DS    2          ; arg a
_1_far_func: DS    2          ; arg b
_2x:        DS    2          ; local x
_2y:        DS    2          ; local y

XREF      _0_far_func
XREF      _1_far_func
XREF      _far_func

```

```
_far_call:
    ld        R0,#high(_0_far_func)
    ld        R1,#low(_0_far_func)
    clr      R2
    lde       @RR0,R2
    ld        R2,#10
    incw      RR0
    lde       @RR0,R2
    ld        R0,#high(_1_far_func)
    ld        R1,#low(_1_far_func)
    clr      R2
    lde       @RR0,R2
    ld        R2,#20
    incw      RR0
    lde       @RR0,R2
    call      _far_func

.FRAME      _far_call, _f_far_call, XDATA
.FCALL      _far_func
```

### 6.2.3 Near Dynamic Call Frames

Near dynamic call frames use the hardware stack in onboard RAM to store arguments and local variables. Because they do not use static call frames near dynamic frame functions do not have **.FRAME** or **.FCALL** directives. Near dynamic call frames are generated by **Z8CC** when **-model = small** is used and a function is defined using the reentrant keyword.

Near dynamic call frame functions use R15 as a near frame pointer into the hardware stack. All arguments and local variables are referenced as an offset from the R15 frame pointer. Though this ties up one register for the life of a function, it simplifies memory accesses, and results in more efficient code.

When a near dynamic call frame function is called, the arguments are pushed onto the stack in reverse order. Once control is passed to the function, it preserves the old frame pointer (R15) and sets up the new frame pointer to be the stack pointer address. Space is then allocated on the stack for local variables. If the **-stkck** switch is active, a call to the **\_s\_nstkck** near stack check function is called to look for stack overflow. Before exiting the function, the local variables are popped off the stack and the stack pointer and old frame pointer are restored. The arguments are NOT popped yet, since there may be an unknown number of arguments passed.

Upon return from from the call, the caller pops the arguments off of the stack by updating the stack pointer, and execution continues.

The following code fragments show an example of calling and setting up a near dynamic call frame:

```

_near_call:
    LD      R0,#20
    PUSH    R0
    CLR     R0
    PUSH    R0
    LD      R0,#10
    PUSH    R0
    CLR     R0
    PUSH    R0
    CALL    _near_func
    ADD     SPL,#4
    RET

_near_func:
    PUSH    R15
    LD      R15,SPL
    ADD     SPL,#4

...

    LD      SPL,R15
    POP     R15
    RET

;***** _near_func *****
;Name          Addr/Register    Size  Type

;j             R15 + 5          2     parameter
;i             R15 + 5          2     parameter
;

;Local Frame Size: 6 (bytes)
;

.FRAME _near_func,_seg__near_func,RDATA
      XDEF _near_call
      XDEF _near_func
END

_far_call:
    LD      R0,#20

```

```
        PUSH    R0
        CLR     R0
        PUSH    R0
        LD      R0,#10
        PUSH    R0
        CLR     R0
        PUSH    R0
        CALL    _far_func
        ADD     SPL,#4
        ADC     SPH,#0
        RET

_far_func:
        PUSH    R14
        PUSH    R15
        LD      R14,SPH
        LD      R15,SPH
        ADD     SPL,#4
        ADC     SPH,#0

...

        LD      SPL,R15
        LD      SPH,R14
        POP     R15
        POP     R14
        RET
```

## 6.2.4 Far Dynamic Call Frames

Far dynamic call frames are by far the most expensive. They are implemented using a simulated stack in external RAM. Two 16-bit pointers held in onboard RAM names **\_\_sp** and **\_\_fp** represent the far stack and frame pointers, respectively. As with all **far** functions, **\_\_s\_fstkck** is called unless the **-nostkck** switch is active.

The mechanism for accessing arguments and local variables, and passing arguments is similar to that of near dynamic call frame functions, but much more cumbersome and inefficient, since a simulated stack held in external RAM is used. The following code fragments show the general form of a call to a far dynamic frame function.

```
_far_call:
    ld      R0,__sp
    ld      R1,__sp + 1
    clr     R2
    lde     @RR0,R2
```

```

incw    RR0
ld      R2,#20
lde     @RR0,R2
clr     R2
ld      @RR0,R2
incw    RR0
ld      R2,#10
lde     @RR0,R2
call    _far_func
ld      R0,___sp + 1
add     R0,#252
ld      ___sp + 1,R0
addc    R0,#255
ld      ___sp,a
...

_far_func
call    ___s_fenter
dw      4           ;Local var size
call    ___s_fstkck
...

call    ___s_fsexit
dw      4           ;Local var size
ret

```

## 6.3 Return Values

When a function returns a value to its caller it is stored in a register.

Caution should be used when returning a structure from a function. This type of function result causes the code to be non-reentrant, since a pointer to the structure is returned. Also, one should be sure that the structure returned is not allocated on the stack since any interrupt handler that uses the stack may cause the values in the structure to be altered if an interrupt occurs between the time the stack is deallocated and the structure is used. To get around this problem, be sure to copy the structure to a static location before returning it as a function result.

<b>Value</b>	<b>Return Register</b>
char	R0
short	WR0
int	WR0
long	DR0
float	DR0
double	DR0:DR4
near pointer	R0
far pointer	WR0
huge pointer	DR0

## 6.4 Segments

The compiler reserves several segments to hold various types of code and data. The following are the segments used by the compiler.

CODE	Is in space "ROM" and holds code.
TEXT	Is in space "ROM" and holds initialized read only data.
NEAR_BSS	Is in space "data" and holds uninitialized near data.
FAR_BSS	Is in space "xdata" and holds uninitialized far data.
NEAR_DATA	Is in space "data" and holds initialized near data.
FAR_DATA	Is in space "xdata" and holds initialized far data.
FAST_CODE	Is in space "ROM" and holds code for onboard RAM.

## 6.5 Variable Names

Names of global static variables are prefixed with an underscore.

Names of local static variables are prefixed with an underscore followed by a function number so as not to get assembly errors when the same local static variable occurs more than once in the same file.



# Troubleshooting

## Chapter 7

---

This chapter addresses common problems which may occur while using the compiler or runtime support.

\*\*\*\*\* Couldn't spawn P1.Status:<n> \*\*\*\*\*

usually means that either Z8CC environment variable is set improperly or not enough RAM is available to load the program P1.EXE.

\*\*\*\*\* Couldn't spawn P3.Status:<n> \*\*\*\*\*

usually means that either not enough RAM is available to load the code generator program or the user has specified a code generator that does not exist.

\*\*\*\*\* Couldn't execute assembler \*\*\*\*\*

usually means that the assembler is not in the path.



# Language

# Appendix A

## A.1 Conventions

A colon (:) following a non-terminal introduces its definition.

Alternative definitions are listed on separate lines, except when prefaced by the words “one of”.

An optional symbol is indicated by the subscript “opt,” so that

$$\{ \textit{expression opt} \}$$

indicates an optional expression enclosed in braces.

## A.2 Character Sets

The compiler supports two character sets: the ASCII character set and tri-graphs used to generate ASCII characters that some terminals do not support.

In a character constant or string literal, characters in the execution set are represented by corresponding characters in the source character set or by *escape sequences* consisting of the backslash (\) followed by one or more characters. A representation with all bits set to 0, called the *null character*, exists in the execution character set, and is used to terminate string literals.

The following characters are in the source and execution character sets: the 52 uppercase and lowercase letters of the English alphabet,

```
a b c d e f g h i j k l m n
o p q r s t u v w x y z
A B C D E F G H I J K L M N
O P Q R S T U V W X Y Z
```

the 10 decimal digits,

```
0 1 2 3 4 5 6 7 8 9
```

the following 29 graphic characters,

! “ # % & ’ ( ) \* + , - . / :  
; < = > ? [ \ ] ^ \_ { | } ~

the space character, and control characters representing horizontal tab, vertical tab, and form feed.

In source files, a standard character sequence indicates the end of each line of text; this standard treats such an end-of-line indicator as if it were a single new-line character. In the execution character set, a control character represents the new line.

If the compiler encounters any other characters in a source file (except in a preprocessing token, a character constant, a string literal, or a comment), the behavior is undefined.

## A.2.1 Trigraph Sequences

All occurrences in a source file of the following sequences of three characters (called *trigraph sequences*) are replaced with the corresponding single character.

Trigraph Sequence	Character
??=	#
??(	[
??/	\
??)	]
??’	^
??<	{
??!	
??>	}
??-	~

No other trigraph sequences exist. The compiler does not change ? characters that do not begin one of the trigraphs listed above.

### Example

The following source line

```
printf("What???/n");
```

becomes (after replacement of the trigraph sequence `??/`)

```
printf("What?\n");
```

## A.3 Lexical Elements

### Syntax

*token:*

*keyword*

*identifier*

*constant*

*string-literal*

*operator*

*punctuator*

A *token* is the minimal lexical element of the language and can consist of any keyword, identifier, or constant separated by *white space* from any otherwise adjacent keyword, identifier, or constant. The categories of tokens are:

keywords,

identifiers,

constants,

string literals,

operators, and

punctuators.

The compiler ignores *white space*, such as space characters, horizontal tabs, new-line characters, vertical tabs, form feeds, and comments (described later), except as it separates tokens.

When parsing an input string, the compiler considers the next token to be the longest sequence of characters that could constitute a token.

**NOTE:** An identifier or a constant constitutes a single token. White space should not appear between the characters that constitute the tokens in the syntax descriptions in this section.

### A.3.1 Keywords

The following tokens, constructed of lower-case letters, are reserved for use as keywords, and should not be used otherwise.

auto	do	goto	register	typedef
bit*	double	huge*	return	union
break	else	if	short	unsigned
case	enum	int	signed	void
char	extern	interrupt*	sizeof	volatile
const	far*	long	static	while
continue	float	near*	struct	
default	for	reentrant*	switch	

\* Language extension for embedded applications.

### A.3.2 Identifiers

#### Syntax

*identifier:*  
    *nondigit*  
    *identifier nondigit*  
    *identifier digit*

*nondigit:* one of  
    a b c d e f g h i j k l m  
    n o p q r s t u v w x y z  
    A B C D E F G H I J K L M  
    N O P Q R S T U V W X Y Z

*digit:* one of  
    0 1 2 3 4 5 6 7 8 9

#### Description

An identifier is a sequence of nondigit characters (including the underscore (`_`) and the lowercase and uppercase letters) and digits. The first character should be a nondigit character.

#### Constraints

An identifier should not consist of the same sequence of characters as a keyword. No specific limit exists on the maximum length of an identifier.



## Semantics

An identifier denotes one of the following:

a function;

an object;

a tag or a member of a structure, union, or enumeration;

a typedef name;

a label name; or

a macro name.

Macro names are not considered further here, because the compiler replaces macro names with their corresponding token sequences before the semantic phase of program translation.

### A.3.2.1 Scopes of identifiers

An identifier is *visible* (that is, can be used) only within a region of program text called its *scope*. Four kinds of scopes exist:

function,

file,

block, and

function prototype.

A *function prototype* is a declaration of a function that declares the types of its parameters.

A label name is the only kind of identifier that has *function scope*. It can be used (in a **goto** statement) anywhere in the function in which it appears, and is declared implicitly by its syntactic appearance (followed by a colon and a statement). Label names should be unique within a function.

The compiler determines the scope of all other identifiers by the placement of their declarations:



If the declaration appears outside any block, the identifier has *file scope*, which extends from the completion of its declarator to the end of the source file, including all source files included by preprocessing directives.

If the declaration appears within a block or in the list of parameter identifiers in a function definition, the identifier has *block scope*, which extends from the completion of the declarator to the “}” that closes the associated block.

If an outer declaration of a lexically identical identifier exists, the compiler “hides” the identifier until it encounters the “}” that closes the block.

If the declaration appears within the list of parameters in a function prototype, the identifier has *function prototype scope*, which extends from the completion of the declaration of the parameter to the end of the function declarator.

### A.3.2.2 Linkages of identifiers

Two lexically identical identifiers declared in different scopes can refer to the same function or object through a process called *linkage*. Three kinds of linkage exist: external, internal, and none.

#### External Linkage

In the set of source files and libraries that constitute an entire program, every instance of a particular identifier with *external linkage* denotes the same function or object. If the declarations in two source files or in disjoint scopes within one source file differ, the behavior is undefined.

If its first lexical declaration with file scope in the source does not contain the keyword **static**, an identifier of a function or object declared with file scope has external linkage.

If the declaration of a function or object contains the keyword **extern**, the identifier has the same linkage as any previous declaration of the identifier with file scope. If no previous declaration exists, the identifier has external linkage.

If an identifier declared with external linkage is used in an expression, the entire program should hold exactly one external definition for the identifier (a declaration that has external linkage and for which storage is allocated). If an identifier declared with external linkage is not used in an expression, the identifier needs no external definition.

#### Internal Linkage

Within one source file, each instance of an identifier with *internal linkage* denotes the same function or object.

If its first lexical declaration with file scope in the source contains the keyword **static**, an identifier of a function or object declared with file scope has internal linkage.

## No Linkage

Identifiers with *no linkage* denote unique entities.

An identifier that declares anything other than a function or an object has no linkage. An identifier declared as a function parameter has no linkage.

Except for a declaration of a function, an identifier declared within a block without the keyword **extern** should have only one declaration in the same scope. Such an identifier has no linkage.

### A.3.2.3 Name spaces of identifiers

If more than one declaration of a particular identifier is visible at any point in a source file, the syntactic context may clarify uses that refer to different entities. Because of these cases, separate *name spaces* exist for various categories of identifiers:

The *label names* (clarified by the syntax of the label declaration and use).

The *tags* of structures, unions, and enumerations. Although they are clarified by the preceding **struct**, **union**, or **enum** keyword, tags share the same name space.

The *members* of structures or unions. Each structure or union has a separate name space for its member, clarified by the type of the expression used to access the member through the `.` or `->` operator.

All other identifiers, called *ordinary identifiers* (declared in ordinary declarators or as enumeration constants).

### A.3.2.4 Storage durations of objects

An object has *storage duration* that determines its lifetime. There are two storage durations:

static and

automatic.

An object declared with *static storage duration* is created and initialized only once, when the program begins execution. It exists and retains its value, unless explicitly modified, throughout the execution of the entire program.

A new instance of an object declared with *automatic storage duration* is created on each normal entry into the block in which it is declared or on a jump to a label in the block or in an enclosed block. If an initialization is specified, it is performed on each normal entry, but not if the block ends by

reaching the terminating “}”,

jumping to an enclosing block, or

executing a break or **return** statement.

### A.3.2.5 Types

An identifier's type determines its permitted use. If the identifier designates an object, the type is characterized by the set of values that such an object may contain. The meaning of the value of an object is determined by the types of the identifiers in an expression used to access it.

Characters, integers, and floating-point numbers are collectively called the *basic types*.

An object declared as a character (**char**) is large enough to store any member of the required source character set. If such a character is stored in a **char** object, its value is non-negative.

#### Signed integer types

Four *signed integer types* exist:

**signed char**,

**short int**,

**int**, and

**long int**.

A **signed char** occupies the same amount of storage as a “plain” **char**. The natural size of a “plain” **int** is suggested by the architecture of the execution

environment: the size must be large enough to contain any value in the range **INT\_MIN** to **INT\_MAX** as defined in the header **<limits.h>**). The other types are provided to meet special needs.

The set of values of each signed integral type is a subset of the values of the next type in the list above. Even if the implementation defines two or more types of integers to have the same set of values, they are nevertheless different types.

For **signed char**, **short int**, and each type of **int**, a corresponding *unsigned integer type* exists, declared with the keyword **unsigned**, that uses the same amount of storage, including sign information. The set of non-negative values of a signed type is a subset of its corresponding unsigned type. A computation involving unsigned objects can never overflow, because a result that does not fit in an object of that size is reduced by the modulo of the largest unsigned number that can be represented plus one.

### Enumerated types

The *enumerated types* are **enumeration** and **void**. An **enumeration** comprises a set of named integer constant values. The **void** type specifies an empty set of values; no object may have type **void**.

Any number of *derived types* can be constructed from the basic and enumerated types:

*arrays* comprising a contiguously allocated set of members of any one type of object;

*structures* comprising a sequentially allocated set of named members of various types of objects;

- *unions* comprising an overlapping set of named members of various types of objects;

*functions* that accept arguments of various types and return a value of any one type of object except an array;

*pointers* to functions, to objects of any type, and to **void**.

The user can apply these methods of constructing derived types recursively.

### Integral Types

Types **char** and **int** (of all sizes), both signed and unsigned, and enumerations are collectively called *integral types*. The values of integral types should be interpreted in a pure binary numeration system.

### Floating Types

Types **float**, **double**, and **long double** are collectively called *floating types*. The representations of floating types are unspecified.

### Miscellaneous Types

Integral and floating types are collectively called *arithmetic types*. Arithmetic types and pointers are collectively called *scalar types*. Arrays and structures are collectively called *aggregate types*.

The type of a function comprises the type of the value returned, the number and types of the parameters (if specified), and whether a variable number of arguments is permitted.

## A.3.3 Constants

### Syntax

*constant:*  
*floating-constant*  
*integer-constant*  
*enumeration-constant*  
*character-constant*

### Constraint

The value of a constant should be representable in the storage appropriate for its type in the execution environment.

### Semantics

Each constant has a type, determined by its form and value.

### A.3.3.1 Integer constants

#### Syntax

*integer-constant:*  
*decimal-constant integer-suffix opt*  
*octal-constant integer-suffix opt*  
*hexadecimal-constant integer-suffix opt*

*decimal-constant:*  
    *nonzero-digit*  
    *decimal-constant digit*

*octal-constant:*  
    0  
    *octal-constant octal-digit*

*hexadecimal-constant:*  
    **0x** *hexadecimal-digit*  
    **0X** *hexadecimal-digit*  
    *hexadecimal-constant hexadecimal-digit*

*nonzero-digit*: one of  
1 2 3 4 5 6 7 8 9

*octal-digit*: one of  
0 1 2 3 4 5 6 7

*hexadecimal-digit*: one of  
0 1 2 3 4 5 6 7 8 9  
a b c d e f  
A B C D E F

*integer-suffix*:  
*unsigned-suffix long-suffix opt*  
*long-suffix unsigned-suffix opt*

*unsigned-suffix*: one of  
u U

*long-suffix*: one of  
l L

## **Description**

An integer constant begins with a digit, but has no decimal point or exponent part. It may have a prefix that specifies its base and a suffix that specifies its type. Below is a table describing integer constants.



## Semantics

Below is a table showing each integer constant type and its numerical base.

The lexically first digit is the most significant.

The type of an integer constant is the first of the corresponding list in which its values can be represented.

Unsuffixes decimal: **int**, **long int**, **unsigned long int**

Unsuffixes octal or hexadecimal: **int**, **unsigned int**, **long int**, **unsigned long int**

Suffixes by the letter **u** or **U**: **unsigned int**, **unsigned long int**

Suffixes by the letter **l** or **L**: **long int**, **unsigned long int**

Suffixes by both the letters **u** or **U** and **l** or **L**: **unsigned long int**

### A.3.3.2 Enumeration Constants

#### Syntax

*enumeration-constant:*  
*identifier*

#### Semantics

An identifier declared as an enumeration constant has type **int**.

### A.3.3.3 Character constants

#### Syntax

*character-constant:*

*L* *opt* 'c-char-sequence'

*c-char-sequence:*

*c-char*

*c-char-sequence c-char*

*c-char:*

any character in the source character set except  
the single-quote ('), backslash (\), or new-line character

*escape-sequence*

*escape-sequence:* one of

|' |" |? ||

|o |oo |ooo

|xh |xhh |xhhh

|a |b |f |n |r |t |v

#### Description

A character constant is a sequence of one or more characters enclosed in single quotes, as in '**x**' or '**ab**'. A wide character constant is the same, except prefixed by the letter **L**. With a few exceptions detailed later, the characters are any characters in the source character set.

The single quote ('), the double quote ("), the question mark (?), the backslash (\), and arbitrary integral values are representable according to the following table of escape sequences:

single-quote (')	'
double-quote (")	"
question-mark (?)	?
backslash (\)	
octal integer	o  oo  ooo
hexadecimal integer	xh  xhh  xhhh
alert	a
backspace	b
form feed	f
new line	n
carriage return	r
horizontal tab	t
vertical tab	v

The double quote (") and question mark (?) are representable by themselves or by the escape sequences \" and (?) respectively, but the single quote (') should be represented by the escape sequence \'.)

The octal digits that follow the backslash in the escape sequences \o, \oo, and \ooo are taken to be part of the construction of a single character. The numerical value of the octal integer so formed specifies the value of the desired character.

The hexadecimal digits that follow the backslash and the letter x in the escape sequences \xh, \xhh, and \xhhh are taken to be part of the construction of a single character. The numerical value of the hexadecimal integer so formed specifies the value of the desired character.

\a (alert) produces an audible or visible alert. The active position is not changed.

\b (backspace) moves the active position to the position of the previous character. If the active position is at the initial position of a line, the behavior is unspecified.

\f (form feed) moves the active position to the initial position at the start of the next logical page.

\n (new line) moves the active position to the initial position of the next line.

\r (carriage return) moves the active position to the initial position of the current line.

\t (horizontal tab) moves the active position to the next horizontal tabulation position on the current line. If the active position is at or past the last defined horizontal tabulation position, the behavior is unspecified.

\v (vertical tab) moves the active position to the initial position of the next vertical tabulation position. If the active position is at or past the defined vertical tabulation position, the behavior is unspecified.

If an escape sequence other than one of these is encountered, the behavior is undetermined.

## Semantics

A character constant has type **int**. The value of a character constant containing a character that maps into a character in the execution character set, is the numerical value of the representation of the mapped character interpreted as an integer. The value of a character constant containing more than one character, or containing a character or escape sequence not represented in the execution character set, is flagged as an error.

A wide character constant has type **wchar\_t**, an integral type defined in **<stddef.h>**. The value of a wide character constant containing a single multibyte

character that maps into a member of the extended execution character set, is the *wide character* (code) corresponding to that multibyte character.

### Examples

The construction `\0` represents the null character.

## A.3.4 String literals

### Syntax

*string-literal*:

`Lopt "s-char-sequenceopt"`

*s-char-sequence*:

`s-char`

`s-char-sequence s-char`

*s-char*:

any member of the source character set except  
the double-quote (`"`), backslash (`\`), or new-line character

*escape-sequence*

### Description

A string literal is a sequence of zero or more characters enclosed in double-quotes, as in `"xyz"`.

The same considerations apply to each element to a sequence in a character string literal or a wide string literal as if it were an integer character constant or a wide character constant, except that the single-quote (`'`) is representable by itself or by the escape sequence (`\'`), but the double-quote (`"`) should be represented by the escape sequence `\"`; in addition, the user may use the same escape sequences as described for character constants.

### Semantics

A *string literal* is a character array with static storage duration. The compiler initializes the literal with the given characters; it concatenates string literals that are adjacent tokens into a single string literal, then appends a null character.

A *character string literal* is a character array that the compiler initializes with the individual bytes of the multibyte character sequence.

A *wide string literal* is an array of type `wchar_t`. The compiler initializes this literal with the sequence of wide characters corresponding to the multibyte character sequence.

Identical string literals need not be distinct. If the program attempts to modify a string literal, the behavior is undefined.

### A.3.5 Operators

#### Syntax

*operator*: one of  
 [ ] ( ) . ->  
 ++ -- & \* + - ~ ! sizeof  
 / % << >> < > <= >= == != ^ | & & ||  
 ? :  
 = \*= /= %= += -= <<= >>= &= ^= |=  
 , # ##

#### Constraints

The operators [ ], ( ), and ? : should occur in pairs, optionally separated by expressions. The operators # and ## may occur in macro-defining preprocessor directives only.

#### Semantics

An operator performs an operation that yields an *evaluation*, or a value. An *operand* is an entity on which an operator acts.

### A.3.6 Punctuator

#### Syntax

*punctuator*: one of  
 [ ] ( ) { } \* , : = ; ... #

### Constraints

The punctuators [ ], ( ), and { } should occur in pairs, optionally separated by expressions, declarations, or statements. The punctuator # may occur in preprocessor directives only.

### Semantics

A punctuator is a symbol that has independent syntactic significance but does not specify an operation to be performed that yields a value. Depending on context, the same symbol may also represent an operator or part of an operator.

## A.3.7 Comments

Except within a character constant, a string literal, the quoted argument to a **#include** or **#line** preprocessing directive, the arguments to a **#error** or **#pragma** preprocessing directive, or a comment, the characters **/\*** introduce a comment. The compiler examines contents of a comment only to find the characters **\*/** that terminate it. The new style of comment is also supported. These comments begin with **//** and are terminated by a new line.

## A.4 Conversions

Several operators may convert operand values from one type to another. This section specifies the result required from such an *implicit conversion*.

Conversion of an operand to the same type causes no change.

### A.4.1 Arithmetic types

#### A.4.1.1 Characters and integers

A **char**, a **short int**, or an **int** bit-field, or their signed or unsigned varieties, may be used in an expression wherever an **int** or **unsigned int** may be used. In all cases the value is converted to an **int** if an **int** can represent all values of the original type; otherwise, it is converted to an **unsigned int**. These are called the *integral promotions*.

The integral promotions preserve the value, including its sign. As discussed earlier, “plain” **char** is treated as signed.

When an integer is demoted to a shorter unsigned integer or **unsigned int** bit-field or **unsigned char**, the result is the remainder modulo the largest unsigned number that can be represented plus one. When an integer is demoted to a shorter

signed integer or **int** bit-field or **signed char**, the least significant bits of the integer are used.

#### A.4.1.2 Signed and unsigned integers

The following types are equivalent:

<b>signed int</b>	=	<b>int</b>
<b>signed short int</b>	=	<b>short int</b>
<b>signed long int</b>	=	<b>long int</b>

When an unsigned integer is promoted to a longer integer, its value is unchanged.

When a positive signed integer is converted to an unsigned integer of equal or greater length, its value is unchanged. Otherwise, if the unsigned integer is longer, the signed integer is first promoted to a signed integer of the same length as the unsigned integer, then converted to unsigned by adding to it the largest number that can be represented in the unsigned integer plus one.

#### A.4.1.3 Floating and integral

When a value of floating type is converted to integral type, the fractional part is discarded. If the value of the integral part cannot be represented in the space provided, the behavior is undefined.

When a value of integral type is converted to floating type, some loss of precision occurs if the destination lacks sufficient precision.

#### A.4.1.4 Usual arithmetic conversions

Many binary operators that expect operands of arithmetic type cause conversions and yield result types in a similar way. The purpose is to yield a common type, which is also the type of the result.

This pattern is called the *usual arithmetic conversions*:

First, if either operand has type **unsigned long int**, the other operand is converted to **unsigned long int**.

Otherwise, if either operand has type **long int**, the other operand is converted to **long int**.

Otherwise, the integral promotions are performed. Then, if either operand has type **unsigned int**, the other operand is converted to **unsigned int**.

Otherwise, both operands have type **int**.

Operands may be converted to other types, provided neither range nor precision is lost; if so, the type of the result is not changed.

## A.4.2 Other Operands

### A.4.2.1 Lvalues and function designators

An *lvalue* is an expression (with an object type or an incomplete type other than **void**) that identifies an object. The type of an object is determined by the lvalue used to identify the object.

A *modifiable lvalue* is an lvalue that does not have

array type,

an incomplete type, or

a const-qualified type.

In addition, if it is a structure or union, the lvalue does not have any member (including, recursively, any member of all contained structures or unions) with a const-qualified type.

An lvalue that is not an array is converted to the value stored in the designated object and is no longer an lvalue, except when it is the operand of

the **sizeof** operator,

the unary **&** operator,

the **++** operator,

the **--** operator, or

the left operand of the **.** operator or an assignment operator.

If the lvalue has qualified type, the value has the unqualified version of the type of the lvalue; otherwise, the value has the type of the lvalue. If the lvalue has an incomplete type and does not have array type, the behavior is undefined.



If the lvalue is an array of a certain type, the compiler converts it to a pointer of that type that points to the initial element of the array type and is not an lvalue. Exceptions occur if the lvalue is

the operand of the **sizeof** operator or the unary **&** operator,

a character string literal that initializes an array of character type, or

a wide string literal that initializes an array with element type compatible with **wchar\_t**.

A *function designator* is an expression that has function type. A function designator that returns a type is converted to a pointer to a function returning that type, except when it is the operand of the **sizeof** operator or the unary **&** operator.

#### A.4.2.2 void

The (nonexistent) value of a **void** expression is not used in any way. Implicit or explicit conversions should not be applied to such an expression.

#### A.4.2.3 Pointers

A pointer to **void** may be converted to a pointer to an object of any type. A pointer to an object of any type may be converted to a pointer to **void** and back again; the result is equal to the original pointer.

An integral constant expression with the value 0 or such an expression cast to type **void \*** is called a *null pointer constant*. If a null pointer constant is assigned to or compared for equality to a pointer, the constant is converted to a pointer of that type. Such a pointer, called a *null pointer*, does not point to any object or function.

#### A.4.2.4 const and volatile

The properties associated with the **const** type specifier are meaningful only for expressions that are lvalues.

An expression that has a type declared with the **const** type specifier is not a modifiable lvalue.

An expression that has a type not declared with the **const** type specifier acquires the **const** attribute if converted to a type declared with the **const** type specifier.

An expression declared to be a pointer to a type without the **const** attribute may be assigned to a pointer to a type with the **const** attribute. This pointer does not modify the object.

Except by an explicit cast, an expression declared to be a pointer to a type with the **const** attribute is not assigned to a pointer to a type without the **const** attribute. If the user attempts to modify an object that has a type declared with the **const** type specifier by means of an lvalue that has a type not declared with the **const** type specifier, the behavior is undefined.

An object declared with the **volatile** type specifier may be modified in ways unknown to the compiler or have other unknown side effects. Therefore, the compiler evaluates any expression referring to such an object strictly according to the sequence rules of the abstract machine. Furthermore, at every sequence point the value of the object in storage agrees with that prescribed by the abstract machine, except as modified by the unknown factors mentioned previously.

An expression declared to be a pointer to a type without the **volatile** attribute may be assigned to a pointer to a type with the **volatile** attribute. In this case, the compiler obeys the rules for volatile objects when the volatile pointer refers to the object.

Except by an explicit cast, an expression declared to be a pointer to a type with the **volatile** attribute is not assigned to a pointer to a type without the **const** attribute. If the user attempts to modify an object declared to be a type with the **const** type specifier by means of an lvalue that has a type not declared with the **const** type specifier, the behavior is undefined.

If an aggregate object type is declared with the **const** or **volatile** type specifier, the type of each member of the aggregate is implicitly declared with that specifier.

### Example

An object declared

```
extern volatile const real_time_clock;
```

may be modified by hardware, but cannot be assigned to, incremented, or decremented.

## A.5 Expressions

An *expression* is a sequence of operators and operands that specifies how to compute a value, how to generate side effects, or both. The order of evaluation is unspecified except for

the function-call operator ( ),

the unary plus operator,

**&&**,

**||**,

**?:**, and

comma operators).

The compiler may evaluate subexpressions in any order, even if the subexpressions produce side effects. The order in which side effects take place is unspecified.

An expression involving more than one occurrence of the same commutative and associative binary operator (**\***, **+**, **&**, **^**, **|**) may be regrouped arbitrarily, even in the presence of parentheses, provided that this regrouping does not change the types of the operands or of the results. Two ways exist to force a particular grouping of operations:

by explicitly assigning the value of the expression to group to an object, or

by preceding the grouping parentheses with a unary plus operator.

Some operators, such as the unary operator **~** and the binary or bitwise operators (**<<**, **>>**, **&**, **^**, and **|**), have operands with integral type.

If an *exception* occurs during the evaluation of an expression (that is, if the result is not mathematically defined or not representable), the behavior is undefined.

## A.5.1 Primary expressions

### Syntax

*primary-expression:*

*identifier*

*constant*

*string-literal*

( *expression* )

### Semantics

An *identifier* is a primary expression if it is an lvalue declared as designating an object or a function locator designating a function.

A *constant* is a primary expression. Its type depends on its form, as detailed in the section titled, “Constants” in this Appendix A.

A *string literal* is a primary expression. It is an lvalue of an *array of char*, as detailed in the section titled, “String Literals” in this Appendix A.

A parenthesized *expression* is a primary expression. Its type and value are identical to those of the unadorned expression. It is an lvalue if the unadorned expression is an lvalue, or a function locator if the unadorned expression is a function locator.

## A.5.2 Postfix operators

### Syntax

*postfix-expression:*

*primary-expression*

*postfix-expression* [ *expression* ]

*postfix-expression* ( *argument-expression-list* *opt* )

*postfix-expression* . *identifier*

*postfix-expression* -> *identifier*

*postfix-expression* ++

*postfix-expression* --

*argument-expression-list:*

*assignment-expression*

*argument-expression-list* , *assignment-expression*

### A.5.2.1 Array subscripting

#### Constraints

One of the expressions is a pointer to *type*. The other expression has integral type. The result is of *type*.

#### Semantics

A postfix expression followed by an expression in square brackets [] is a subscripted designation of a member of an array object.

For instance, **E1** is identical to **(\* (E1+(E2)))**. Because of the conversion rules that apply to the binary **+** operator, if **E1** is an array object and **E2** is an integer, **E1 [E2]** designates the **E2-th** member of **E1** (counting from zero).

Successive subscript operators designate members of a multi-dimensional array object. If **E** is an  $n$ -dimensional array with dimensions  $i \times j \times \dots \times k$ , then **E** (used as other than an lvalue) is converted to a pointer to an  $(n-1)$ -dimensional array with dimensions  $j \times \dots \times k$ . If the unary **\*** operator is applied to this pointer explicitly, or implicitly as a result of subscripting, the result is the pointed-to  $(n-1)$ -dimensional array, which itself is converted into a pointer if used as other than lvalue. Therefore, arrays are stored in row-major order: the last subscript varies fastest.

### Example

Consider the array object defined by the declaration

```
int x[3] [5];
```

Here, **x** is a 3x5 array of **ints**; more precisely, **x** is an array of three member objects, each of which is an array of five **ints**. In the expression **x [i]**, which is equivalent to **(\* (x+(i)))**,

First, **x** is converted to a pointer to the initial array of five **ints**.

Next, **i** is adjusted according to the type of **x**, which conceptually entails multiplying **i** by the size of the object to which the pointer points: an array of five **int** objects.

The compiler adds the results and applies indirection to yield an array of five **ints**.

When used in the expression **x[i][j]**, that in turn is converted to a pointer to the first of the **ints**, so **x[i][j]** yields an **int**.

## A.5.2.2 Function calls

### Constraints

A function does not return a value of an array or function.

If a function prototype declarator is in scope, the number of arguments must agree with the number of formal parameters. The types must be such that each formal parameter may be assigned the value of the corresponding argument.

### Semantics

A postfix expression followed by parentheses ( ) containing a possibly empty, comma-separated list of expressions. The list of expressions specifies the arguments to the function. The postfix expression denotes the function called, and is declared as a pointer to a function returning *type* (which could be the result of converting a function locator). The result is of *type*.

If no declaration is in scope for an identifier used as the first expression in a function call, the identifier is implicitly declared exactly as if, in the innermost block containing the function call, the declaration

**extern int *identifier*( );**

appeared. This function has external linkage and no information about its parameters, and returns an **int**.

An argument may be any expression other than a void expression. In preparing for the call to a function, each argument is evaluated, and each formal parameter is assigned the value of the corresponding argument. An argument or a formal parameter that has an array or function type is converted to a pointer.

If no function prototype declarator is in scope, the integral promotions are performed and arguments that have type **float** are promoted to **double**. These are called the *default argument conversions*. If the number of arguments or their types after conversion do not agree with those of the formal parameters, the behavior is undefined.

If a function prototype declarator is in scope, the arguments are compared with the formal parameters and are converted accordingly. The ellipsis notation (...) in a function prototype declarator stops argument type checking and conversion stop after the last declared formal parameter. The default argument conversions are performed on unchecked arguments.

No other conversions are performed automatically. In particular, the number and types of arguments are not compared with those of the formal parameters in a function definition that does not include a function prototype declarator.

The order of evaluation of arguments and of subexpressions within arguments is unspecified, but the call itself is a sequence point before which all side effects are completed.

Recursive function calls are permitted, both directly and indirectly through any chain of other functions.

### **A.5.2.3 Structure and union members**

#### **Constraints**

The first operand of the `.` operator has a structure or union type, and the second operand names a member of that type.

The first operand of the `->` operator is a pointer to a structure or union. The second operand names a member of the type pointed to.

## Semantics

A postfix expression followed by a dot (.) and an identifier designates a member of a structure or union entity. The value is that of the named member, and is an lvalue unless the first expression is the value returned by a function call.

A postfix expression followed by an arrow -> and an identifier designates a member of a structure or union object. The value is that of the named member of the object to which the first expression points. This value is an lvalue.

With one exception, if a member of a union is inspected when the value of the object has been assigned using a different member, the behavior is indeterminate. One special guarantee is made in order to simplify the use of unions; if a union contains several structures that share a *common initial sequence*, and if the union object currently contains one of these structures, the user may inspect the common initial part of any of them.

## Example

The following is a valid fragment:

```
union {
    struct {
        int      type;
    } n;
    struct {
        int      type;
        int      intnode;
    } ni;
    struct {
        int      type;
        long     longnode;
    } nf;
} u;
u.nf.type = 1;
u.nf.longnode = 3;
/*...*/
if (u.n.type == 1)
    /*...*/
```



### A.5.2.4 Postfix increment and decrement operators

#### Constraints

The operand of the postfix increment or decrement operator has scalar type and is a modifiable lvalue.

#### Semantics

The result of the postfix ++ operator is the value of the operand. After noting the result, the compiler increments (adds 1 to) the value of the operand. See the discussions of additive operators and compound assignment for information of types and conversions and the effects of operations on pointers. The side effect of updating the stored value of the operand may be delayed until the next sequence point is reached.

The postfix -- operator is analogous to the postfix ++ operator, except that the compiler decrements (subtracts 1 from) the value of the operand.

## A.5.3 Unary operators

### Syntax

*unary-expression:*  
*postfix-expression*  
 ++ *unary-expression*  
 -- *unary-expression*  
*unary-operator* *cast-expression*  
**sizeof** *unary-expression*  
**sizeof** ( *type-name* )

*unary-operator:* one of  
 & \* + - ~ !

### A.5.3.1 Prefix increment and decrement operators

#### Constraints

The operand of the prefix increment or decrement operator has scalar type and is a modifiable lvalue.

#### Semantics

The compiler increments the value of the operand of the prefix ++ operator and stores the incremented value before using the value. The expression ++**E** is equivalent to (**E**+=1). See the discussions of additive operators and compound

assignment for information on types and conversions and the effects of operations on pointers.

The prefix `--` operator is analogous to the prefix `++` operator, except that the compiler decrements (subtracts 1 from) the value.

### A.5.3.2 Address and indirection operators

#### Constraints

The operand of the unary `&` operator is either a function locator or an lvalue that designates an object other than a bit-field of an object declared with the **register** storage-class specifier.

The operand of the unary `*` operator has pointer type, other than pointer to void.

#### Semantics

The result of the unary `&` (address-of) operator is a pointer to the object designated by its operand. If the operand has *type*, the result is a pointer to *type*.

The unary `*` operator denotes indirection. If the operand points to a function, the result is a function locator; if it points to an object, the result is an lvalue designating the object. If the operand is a pointer to *type*, the result is of that type. If an invalid value is assigned to the pointer, the behavior of the unary `*` operator is undefined.

### A.5.3.3 Unary arithmetic operators

#### Constraints

The operand of the unary `+` and unary `-` operator has arithmetic type. The operand of the `~` operator has integral type. The operand of the `!` operator has scalar type.

#### Semantics

The result of the `+` operator is the value of its operand. The compiler performs integral promotion on the operand, and the result has the promoted type. Except that it inhibits regrouping of subexpressions of **E** with subexpressions outside of **E**, the expression `+E` is equivalent to `(0+E)`.

The result of the unary `-` operator is the negative of its operand. The integral promotion is performed on the operand, and the result has the promoted type. The expression `-E` is equivalent to `(0-E)`.

The result of the `~` operator is the bit-wise complement of its operand. In other words, each bit in the result is set if, and only if, the corresponding bit in the

converted operand is not set. The integral promotion is performed on the operand, and the result has the promoted type.

The expression `~E` is equivalent to `(ULONG_MAX-E)` if `E` has type **unsigned long**, or to `(UINT_MAX-E)` if `E` has any other unsigned type. (The constants **ULONG\_MAX** and **UINT\_MAX** are defined in the header `<limits.h>`.)

The result of the logical negation operator `!` is 0 if the value of its operand is non-zero, or 1 if the value of its operand is 0. The result has type **int**. The expression `!E` is equivalent to `(0==E)`.

#### A.5.3.4 The `sizeof` operator

##### Constraints

The **sizeof** operator may not be applied to an expression that has type function, bit-field, or void, or to the parenthesized name of the type of such an expression.

##### Semantics

The **sizeof** operator yields the size (in bytes) of its operand, which may be an expression or the parenthesized name of a type.

When applied to an operand that has type **char**, **unsigned char**, or **signed char**, the result is 1.

When applied to an operand that has array type, the result is the total number of bytes in the array.

When applied to an operand that has structure or union type, the result is the total number of bytes in such an object considered as a member of an array, including whatever internal and trailing padding might be needed to align each member in such an array properly.

The size is determined from the type of the operand, which is not itself evaluated. The result is an integer constant. The size of the result is determined by the code generator, and its type (an unsigned integral type) **size\_t**. **size\_t** is defined in the `<stddef.h>` header.

##### Examples

A principal use of the **sizeof** operator is in communication with routines such as storage allocators and I/O systems. A storage-allocation function might accept

a size (in bytes) of an object to allocate and return a pointer to **void**. For example:

```
extern void *alloc(size_t);  
double *dp = alloc(sizeof *dp);
```

The **alloc** function ensures that its return value is aligned suitably for conversion to a pointer to **double**.

Another use of the **sizeof** operator is to compute the number of members in an array:

```
sizeof array / sizeof array[0]
```

## A.5.4 Cast operators

### Syntax

```
cast-expression:  
    unary-expression  
    ( type-name ) cast-expression
```

### Constraints

If the type name specifies **void** type, the operand may be any expression other than a void expression. Otherwise, the type name specifies scalar type and the operand has scalar type.

### Semantics

Preceding an expression by a parenthesized type name converts the value of the expression to the named type. This construction is called a *cast*. A cast that specifies an implicit conversion or no conversion has no effect on the type or value of an expression.

Conversions involving pointers (other than from a pointer to void or from a pointer to an object) can be specified by an explicit cast.

A pointer may be converted to an integer. If the space provided is not long enough, the behavior is undefined.

An arbitrary integer may be converted to a pointer.

A pointer to an object of one type may be converted to a pointer to an object of another type. The resulting pointer might not be valid if it is improperly aligned for the type of object pointed to. However, a pointer to an object of a given alignment may be converted to a pointer to an object of a less strict alignment and back again; the result compares equal to the original pointer. (An object that has type **char** has the least strict alignment.)

A pointer to a function of one type may be converted to a pointer to a function of another type and back again; the result compares equal to the original pointer. If a converted pointer is used to call a function of other than the original type, the behavior is undefined.

### A.5.5 Multiplicative operators

#### Syntax

*multiplicative-expression:*  
*cast-expression*  
*multiplicative-expression* \* *cast-expression*  
*multiplicative-expression* / *cast-expression*  
*multiplicative-expression* % *cast-expression*

#### Constraints

Each operand has arithmetic type. The operands of the % operator have integral type.

#### Semantics

The compiler performs the usual arithmetic conversions on the operands.

The result of the binary \* operator is the product of the operands. The binary \* operator is commutative and associative, and expressions involving several multiplications at the same level may be regrouped.

The result of the / operator is the quotient of the operands.

The result of the % (modulo) operator is the remainder from the division of the first operand by the second.

When integers are divided and the division is inexact, and if both operands are positive, the result of the / operator is the largest integer less than the true quotient and the result of the % operator is positive. If either operand is negative, the result of the / operator is the largest integer less than the true quotient.

### Example

The expression  $(a/b)*b + a\%b$  equals  $a$  (if  $b$  is not 0).

## A.5.6 Additive operators

### Syntax

*additive-expression:*

*multiplicative-expression*

*additive-expression* + *multiplicative-expression*

*additive-expression* - *multiplicative-expression*

### Constraints

Both of the operands may have arithmetic type. Additional type possibilities for each operator involve pointer operands: one operand may be a pointer to an object and the other an expression that has integral type. (Incrementing or decrementing is equivalent to adding or subtracting 1.) The user may subtract two pointers to objects that have the same type.

### Semantics

If both operands have arithmetic type, the compiler performs the usual arithmetic conversions.

The result of the binary + operator is the sum of the operands. The binary + operator is commutative and associative, and expressions involving several additions at the same level may be regrouped.

The result of the binary - operator is the difference of the operands.

When an expression that has integral type is added to or subtracted from a pointer, the integral value is first multiplied by the size of the object pointed to. The result is a pointer of the same type as the original pointer. If the original pointer points to a member of an array object, and the array object is large enough, the result points to another member of the same array object.

For instance, if  $P$  points to a member of an array object, the expression  $P+1$  points to the next member of the array object. If the result is used as the operand of a unary \* operator, the behavior is undefined unless both the pointer operand and the result points to a member of the same array object.

When two pointers to members of the same array object are subtracted, the difference is divided by the size of a member. The result represents the difference of the subscripts of the two array members. The type of the result is a signed

integral type, **ptrdiff\_t**, defined in the `<stddef.h>` header. As with any other arithmetic overflow, if the result does not fit in the space provided, the behavior is undefined.

If two pointers that do not point to members of the same array object are subtracted, the behavior is undefined. However, if **P** points to the last member of the same array object, the expression **(P+1) P** has the value 1, even though **P+1** does not point to a member of the same array object as **P**.

### Example

The compiler may compute the expression **a + (b + c)** as **(a + b) + c**. This feature allows the compiler to sum **a** and **b** during program translation if they are constant expressions. However, this regrouping might lead to overflow or loss of precision. One way to force the desired grouping is to write the expression as **a + + (b + c)**, where white space is required between the first two plus signs.

## A.5.7 Bitwise shift operators

### Syntax

*shift-expression:*  
*additive-expression*  
*shift-expression << additive-expression*  
*shift-expression >> additive-expression*

### Constraints

Each of the operands has integral type.

### Semantics

The compiler performs integral promotions on each of the operands. Then the right operand is converted to **int**; the type of the result is that of the promoted left operand. If the value of the right operand is negative or is greater than or equal to the width in bits of the promoted left operand, the behavior is undefined.

The result of **E1 << E2** is **E1** left-shifted **E2** bit positions; vacated bits are filled with zeros. If **E1** has an unsigned type, the value of the result is

$$(\mathbf{E1} * 2^{\mathbf{E2}}) - \text{modulo}(\mathbf{ULONG\_MAX} + 1)$$

if **E1** has type **unsigned long**, or

$$(\mathbf{E1} * 2^{\mathbf{E2}}) - \text{modulo}(\mathbf{UINT\_MAX} + 1)$$

otherwise.

The constants **ULONG\_MAX** and **UINT\_MAX** are defined in the header **<limits.h>**.

The result of **E1 >> E2** is **E1** right-shifted **E2** bit positions. If **E1** has a signed type, an arithmetic shift is performed. If **E1** has an unsigned type, the right shift is logical; the value of the result is the integral part of the quotient of **E1** divided by (2 raised to the power of **E2**).

## A.5.8 Relational operators

### Syntax

*relational-expression:*  
*shift-expression*  
*relational-expression < shift-expression*  
*relational-expression > shift-expression*  
*relational-expression <= shift-expression*  
*relational-expression >= shift-expression*

### Constraints

Both of the operands may have arithmetic type, or both may be pointers to objects that have the same type.

### Semantics

If both of the operands have arithmetic type, the compiler performs the usual arithmetic conversions.

When two pointers are compared, the result depends on the relative locations in the address space of the objects pointed to; pointers are compared as if they were unsigned integers of the appropriate length. If the objects pointed to are not members of the same aggregate object, the pointer expression **P=1** is greater than **P**, even though **P+1** does not point to a member of the same array object as **P**.

Each of the operators **<** (less than), **>** (greater than), **<=** (less than or equal to), and **>=** (greater than or equal to) yield 1 if the specified relation is true and 0 if it is false. The result has type **int**.

## A.5.9 Equality operator

### Syntax

*equality-expression:*  
*relational-expression*  
*equality-expression == relational-expression*



*equality-expression != relational-expression*

### Constraints

Both of the operands may have arithmetic type, or both may be pointers that have the same type. In addition, one may be an object pointer and the other a pointer to **void**, or one may be a pointer and the other a null pointer constant.

### Semantics

The == (equal to) and the != (not equal to) operators are analogous to the relational operators except for their precedence.

If two pointers to objects or functions compare equal, they point to the same object or function, respectively. If one of the operands is a pointer to an object and the other has type pointer to **void**, the object pointer is converted to that type.

## A.5.10 Bitwise AND operator (&)

### Syntax

*AND-expression:*  
*equality-expression*  
*AND-expression & equality-expression*

### Constraints

Each of the operands has integral type.

### Semantics

The compiler performs the usual arithmetic conversions on the operands.

The result of the binary & operator is the bitwise AND of the operands: each bit in the result is set if, and only if, each of the corresponding bits in the converted operands is set. The binary & operator is commutative and associative, and an expression involving several binary & operations at the same level may be regrouped.

## A.5.11 Bitwise exclusive OR operator (^)

### Syntax

*exclusive-OR-expression:*  
*AND-expression*

*exclusive-OR-expression ^AND-expression*

### Constraints

Each operand has integral type.

### Semantics

The compiler performs the usual arithmetic conversions on the operands.

The result of the ^ operator is the bitwise exclusive OR of the operands: each bit in the result is set if, and only if, exactly one of the corresponding bits in the converted operands is set. The ^ operator is commutative and associative, and an expression involving several ^ operations at the same level may be regrouped.

## A.5.12 Bitwise inclusive OR operator (|)

### Syntax

*inclusive-OR-expression:*  
*exclusive-OR-expression*  
*inclusive-OR-expression | exclusive-OR-expression*

### Constraints

Each operand has integral type.

### Semantics

The compiler performs the usual arithmetic conversions on the operands.

The result of the | operator is the bitwise inclusive OR of the operands: each bit in the result is set if, and only if, at least one of the corresponding bits in the converted operands is set. The | operator is commutative and associative, and an expression involving several | operations at the same level may be regrouped.

## A.5.13 Logical AND operator (&&)

### Syntax

*logical-AND-expression:*  
*inclusive-OR-expression*  
*logical-AND-expression && inclusive-OR-expression*

**Constraints**

Each operand has scalar type.

**Semantics**

The **&&** operator yields 1 if both of its operands evaluate to non-zero, or 0 otherwise. The result has type **int**.

Unlike the bitwise binary **&** operator, the **&&** operator guarantees left-to-right evaluation and is a sequence point. If the value of the first operand is 0, the second operand is not evaluated.

**A.5.14 Logical OR operator (||)****Syntax**

*logical-OR-expression:*  
*logical-AND-expression*  
*logical-OR-expression* || *logical-AND-expression*

**Constraints**

Each operand has scalar type.

**Semantics**

The **||** operator yields 1 if either of its operands evaluates to non-zero, or 0 otherwise. The result has type **int**.

Unlike the bitwise **|** operator, the **||** operator guarantees left-to-right evaluation and is a sequence point. If the value of the first operand is non-zero, the second operand is not evaluated.

**A.5.15 Conditional operator (?:)****Syntax**

*conditional-expression:*  
*logical-OR-expression*  
*logical-OR-expression* ? *expression* : *conditional-expression*

**Constraints**

The first operand has scalar type. The second and third operands can be

both arithmetic type,

both structure, union, or pointer type,

both void expressions,

one object pointer and one pointer to **void**, or

one pointer and one null pointer constant.

### Semantics

The **?** operator is a sequence point. It evaluates the first operand. If its value is non-zero the result is the value of the second operand. Otherwise, the result is the value of the third operand. The types of the operands determine the result.

If both the second and third operands have arithmetic type, the usual arithmetic conversions are performed to bring them to a common type. The result has that type.

If both the operands have structure, union, or pointer type, the result has that type.

If both the operands are void expressions, the result is a void expression.

If one of the operands is a pointer to **void**, the other operand is converted to that type and the result has that type.

If one operand is a pointer and the other operand a null pointer constant, the result has the type of the pointer.

## A.5.16 Assignment operators

### Syntax

*assignment-expression:*  
*conditional-expression*  
*unary-expression assignment-operator assignment-expression*

*assignment-operator*: one of  
`= *= /= %= += -= <<= >>= &= ^= |=`

### Constraints

An assignment operator has a modifiable lvalue as its left operand.

### Semantics

An assignment operator stores a value in the object designated by the left operand. An assignment expression has the type of the left operand and the value of the left operand after the assignment, but is not an lvalue. The compiler stores the value of the assignment expression before using the value.

The order of evaluation of the operands is unspecified.

## A.5.16.1 Simple assignment

### Constraints

Both the operands have arithmetic type or the same structure, union, or pointer type. In addition, if the left operand is a pointer, either operand may be a pointer to **void**, or the right operand may be a null pointer constant.

### Semantics

In the *simple assignment* with `=`, the value of the right operand is converted to the type of the left operand and replaces the value of the object designated by the left operand.

If an object is assigned to another object that overlaps in storage with any part of the object being assigned, the behavior is undefined.

### Example

In the program fragment

```
int f(void) ;
char c ;
/*...*/
/*...*/ ((c = f()) == -1) /*...*/
```

the **int** value returned by the function may be truncated when stored in the **char**, and then converted back to **int** width prior to the comparison. In an implementation in which “plain” **char** behaves the same as **unsigned char**, the result of the conversion cannot be negative, so the operands of the comparison can never

compare equal. Therefore, for full portability, the user must declare the variable **c** as **int**.

### A.5.16.2 Compound assignment

#### Constraints

Each operand has arithmetic type consistent with those allowed by the corresponding binary operator; in addition, for **+=** and **-=** only, the left operand may be a pointer, in which case the right operand has integral type.

#### Semantics

A *compound assignment* of the form **E1 op = E2** differs from the *simple assignment expression* **E1 = E1 op (E2)** only in that the lvalue **E1** is evaluated only once.

### A.5.17 Comma operator

#### Syntax

*expression:*

*assignment-expression*

*expression , assignment-expression*

### Semantics

The comma operator is a sequence point. The compiler first evaluates the left operand of a comma operator as a void expression. Then it evaluates the right operand; the result has its type and value.

### Example

As indicated by the syntax, in contexts where a comma is a punctuator (in lists of argument to functions and lists of initializers), the comma operator as described in this section appears only in parentheses. In the function call:

**f(a, (t=3, t+2), c)**

the function has three arguments, the second of which has the value 5.

## A.6 Constant Expressions

### Syntax

*constant-expression:*  
*conditional-expression*

### Constraints

The compiler does not evaluate these operators in any constant expression:

a function-call operator ( ),

an increment or decrement operator ( ++ or -- ),

an assignment operator, or

a comma operator.

An *integral constant expression* involves only integer, enumeration, and character constants, and casts to integral types. The compiler does not evaluate

array-subscripting operators ( [ ] ),

member-access operators ( . and - ),

the *address-of* (&) and indirection (\*) unary operators, and arbitrary casts for integral constant expressions.

Further restrictions that apply to the integral constant expressions used in preprocessing conditional-inclusion directives are discussed later.

For constant expressions in initializers, in addition to integral constant expressions, floating constants and arbitrary casts may be used. Lvalues of objects that have static storage duration or function identifiers may also specify addresses, either explicitly with the unary & operator or implicitly for unsubscribed array identifiers or function identifiers.

### Semantics

An expression that evaluates to a constant is required in several contexts. To specify the size of a bit-field member of a structure, the value of an enumeration constant, the size of an array, or the value of a case constant, the expression is an integral constant expression. More latitude is permitted for initializers.

If the expression is evaluated in the translation environment, the arithmetic precision and range are at least as great as if the expression were being evaluated in the execution environment.

## A.7 Declarations

### Syntax

*declaration:*

*declaration-specifiers init-declarator-list opt ;*

*declaration-specifiers:*

*storage-class-specifier declaration-specifiers opt*

*type specifier declaration-specifiers opt*

*init-declarator-list:*

*init-declarator*

*init-declarator-list , init-declarator*

*init-declarator:*

*declarator*

*declarator = initializer*

### Constraints



A declaration declares at least a declarator, a tag, or the members of an enumeration.

### Semantics

A *declaration* specifies the interpretation and attributes of a set of identifiers. A declaration that also causes storage to be reserved for an object or function named by an identifier is a *definition*.

The declaration specifiers consist of a sequence of specifiers that indicate the scope, storage duration, and type of the entities that the declarators denote. The init-declarator-list is a comma-separated sequence of declarators, each of which may have an initializer. The declarators contain the identifiers (if any) being declared.

## A.7.1 Storage-class specifiers

### Syntax

*storage-class-specifier:*

**typedef**  
**extern**  
**static**  
**auto**  
**register**

### Constraints

The user can declare one storage-class specifier at most.

### Semantics

The **typedef** specifier is called a “storage-class specifier” for syntactic convenience only; it is discussed in the "Declarations" section of Appendix A under "Type definition and type equivalence." The meanings of the various scopes and storage durations were discussed in the "Lexical Elements" section titled, “Scopes of identifiers,” and the section titled, “Storage durations of objects.”

A declaration with storage-class specifier **extern** indicates that, if the identifier declared is referred to, then somewhere in the set of source files that constitutes the entire program, an external object definition for the given identifier (a definition with file scope) exists. If the declaration with storage-class specifier **extern** occurs outside a function, the identifier is declared with file scope and external linkage, and may itself serve as the required definition as discussed later.

A declaration of an object with storage-class specifier **static**, **auto**, or **register** also serves as a definition, as it causes an appropriate amount of storage to be

reserved. The storage-class specifier **static** specifies static storage duration; the storage-class specifiers **auto** and **register** specify automatic storage duration.

A declaration with storage-class specifier **register** is an auto declaration, with a suggestion that the objects declared be stored in fast-access machine registers if possible. The types of objects that are stored in such registers and the number of such declarations in each block that are effective are described in subsection "Operators" in the "Lexical Elements" section of this Appendix A.

The following rules apply to a declaration without a storage-class specifier:

1. For a function, the meaning is the same as if the storage-class specifier were **extern**.
2. For an object declared inside a function or among its formal parameters, the meaning is the same as if the storage-class specifier were **auto**.
3. For an object declared outside a function, the declaration is an external object definition.

## A.7.2 Type specifiers

### Syntax

*type-specifier:*

**char**  
**short**  
**int**  
**long**  
**signed**  
**unsigned**  
**const**  
**volatile**  
**void**

**near**  
**far**  
**huge**  
**float**  
*struct-or-union-specifier*  
*enum-specifier*  
*typedef-name*

### Constraints

At most one of the keywords **long** or **short** may be specified in conjunction with **int**; the meaning is the same if the keyword **int** is not mentioned.

The keywords **signed** or **unsigned** may be specified alone (in which case the presence of **int** is implied), or in conjunction with **int** or its short or long varieties, or with **char**.

The keyword **long** may be specified in conjunction with **double**.

The keywords **const** and **volatile** may be specified alone (in which case the presence of **int** is implied), or in conjunction with other types of specifiers.

Only **const** and **volatile** may be specified in conjunction with a structure, union, or enumeration specifier, or with a **typedef** name.

Otherwise, at most one type specifier may be given in a declaration.

### Semantics

The characteristics of the **const** and **volatile** type modifiers are discussed in the subsection, “const and volatile” in section "Conversions." Specifiers for structures, unions, and enumerations are discussed in the subsection "Type definition and type equivalence" under the "Declarations" section. The characteristics of the other types are discussed in the subsection “Types” under section "Lexical Elements" in this Appendix.

If no type specifiers exist in a declaration, the type is taken to be **int**.

## A.7.2.1 Structure and union specifiers

### Syntax

*struct-or-union-specifier:*  
*struct-or-union identifier* *opt* { *struct-declaration-list* }  
*struct-or-union identifier*

*struct-or-union:*

**struct**  
**union**

*struct-declaration-list:*

*struct-declaration*  
*struct-declaration-list struct-declaration*

*struct-declaration:*

*type-specifier-list struct-declarator-list ;*  
*type-specifier-list:*  
*type-specifier*  
*type-specifier-list type-specifier*

*struct-declarator-list:*

*struct-declarator*  
*struct-declarator-list , struct-declarator*

*struct-declarator:*

*declarator*  
*declarator<sub>opt</sub> : constant-expression*

## Constraints

A structure or union does not contain an instance of itself, but may contain a pointer to an instance of itself.

The constant expression that specifies the width of a bit-field has integral type and non-negative value. The width of a bit-field does not declare more storage than an ordinary object of the same type.

## Semantics

As discussed in subsection, "Types" under section "Lexical Elements" of the Appendix A, a structure is a type consisting of an ordered sequence of named members, and a union is a type consisting of an overlapping sequence of named members. A union is similar to a structure whose members overlap and whose size is sufficient to contain the largest of its members. At most one of the members can be stored in a union object at any time.

Structure and union specifiers have the same form.

The presence of a *struct-declaration-list* in a *struct-or-union-specifier* declares a new type. The *struct-declaration-list* is a sequence of declarations for the members of the structure or union.

A member of a structure or union may have any previously declared type of an object, except the type being declared. In addition, a member may be declared to consist of a specified number of bits, including a sign bit, if any. Such a member is called a *bit-field*; its width is preceded by a colon.

A bit-field may have type **int**, **unsigned int**, or **signed int**. The compiler treats the high-order bit position of a “plain” **int** bit-field as a sign bit.

The compiler may allocate any bit-field small enough to fit in an **int**. If enough space remains, a bit-field that follows another bit-field is packed into adjacent bits of the same **int**. If insufficient space remains, a bit-field that does not fit is put into the next **int**. The order of allocation of bit-fields within an **int** is high-order to low-order.

A bit-field declaration with no declarator, but only a colon and a width, indicates an unnamed bit-field. As a special case of this, a bit-field with a width of 0 indicates that no further bit-field is to be packed into the unit in which the previous bit-field, if any, was placed.

Within a structure object, the bit-field members and the units in which bit-fields reside have addresses that increase in the order in which they are declared. A pointer to a structure object, suitably cast, points to its initial member or, if it is a bit-field, to the unit in which it resides. There may therefore be unnamed holes within or at the end of a structure, but not at its beginning, nor more than necessary to achieve the appropriate alignment.

A pointer to a union object, suitably cast, points to each of its members, or if a member is a bit-field, to the unit in which it resides.

#### A.7.2.2 Structure and union tags

A complete structure or union specifier of the form

*struct-or-union identifier { struct-declaration-list }*

declares the identifier to be the *tag* of the structure or union specified by the list. A subsequent declaration in the same scope may then use the tag, but the bracketed declaration list is omitted.

An incomplete structure or union specifier of the form

*struct-or-union identifier*

declares a tag that may be used only when the size of an object of the specified type is not needed:

when a **typedef** name is declared to be a synonym for a structure or union, or

when a pointer to or a function returning a structure or union is being declared.

The specification must be complete before such a function is called or defined. Another declaration of the tag in the same scope (but not in an enclosed block, which would declare a new type known only within that block) may give the complete specification. A vacuous structure or union specifier of the form

*struct-or-union identifier ;*

supercedes any prior declaration in an enclosing scope.

### Examples

This mechanism allows the user to declare a self-referential structure

```
struct tnode {  
    char tword [20];  
    int count;  
    struct tnode *left;  
    struct tnode *right;  
};
```

declares a structure that contains an array of 20 characters, an integer, and two pointers to objects of the same type. Once this structure is declared, the declaration

struct tnode s, \*sp;

defines **s** to be the object of the given type and **sp** to be a pointer to an object of the given type. With these declarations,

**sp-count** refers to the count member of the object to which **sp** points;

**s.left** refers to the left **struct tnode** pointer of **s**; and

**s.right-tword[0]** refers to the initial character of the **tword** member of the right **struct tnode** pointed to from **s**.

To illustrate the use of prior declaration of a tag to declare a pair of mutually-referential structures, the declarations

```
struct s1 { struct s2 *s2p; /*...*/ }; /* D1 */  
struct s2 { struct s1 *s1p; /*...*/ }; /* D2 */
```

specify a pair of structures that contain pointers to each other. Note, however, that if **s2** were already declared as a tag in an enclosing scope, the declaration **D1** would refer to *it*, not to the tag **s2** declared in **D2**. To eliminate this context sensitivity, the vacuous declaration

```
struct s2;
```

may be inserted ahead of **D1**. This declares a new tag **s2** in the inner scope; the declaration **D2** then completes the declaration of the new type.

### A.7.2.3 Enumeration specifiers

#### Syntax

*enum-specifier:*

```
enum identifier opt { enumerator-list }  
enum identifier
```

*enumerator-list:*

```
enumerator  
enumerator-list , enumerator
```

*enumerator:*

```
enumeration-constant  
enumeration-constant = constant-expression
```

#### Semantics

The identifiers in an enumerator list are declared as constants of type **int** and may appear wherever constants are permitted. If the first enumerator has no =, the values of the corresponding constants begin at 0 and increase by 1 in the order in which they are declared, until an enumerator with = appears. An enumerator with = gives the associated enumeration constant the value indicated; subsequent enumeration constants with no = continue the progression from the earlier ascribed value (and may duplicate other values in the same enumeration).

The scope of an enumeration constant begins after its declaration and ends with the scope of the enumeration of which it is a member. The identifiers of



enumeration constants in the same scope must all be distinct from each other and from other identifiers declared in ordinary declarators.

The role of the identifier in the *enum-specifier* is analogous to that of the tag in a *struct-or-union-specifier*: it names a particular enumeration.

An object that has an enumeration type behaves like an **int** in an expression. The compiler may use the set of values in the enumeration to determine whether to allocate less storage than an **int**.

### Example

```
enum hue { chartreuse, burgundy, claret=20, winedark };
/*...*/
enum hue col, *cp;
/*...*/
col = claret;
cp = &col;
/*...*/
/*...*/ (*cp != burgundy) /*...*/
```

makes **hue** the tag of an enumeration, and then declares **col** as an object that has that type and **cp** as a pointer to an object that has that type. The enumerated values are in the set { 0, 1, 20, 21 }.

## A.7.3 Declarators

### Syntax

```
declarator:
    pointer opt direct-declarator

direct-declarator:
    identifier
    ( declarator )
    direct-declarator [ constant-expression opt ]
    direct-declarator ( parameter-type-list )
    direct-declarator ( identifier-list opt )

pointer:
    * type-specifier-list opt
    * type-specifier-list opt pointer

parameter-type-list:
    parameter-list
    parameter-list , ...
```

*parameter-list:*  
*parameter-declaration*  
*parameter-list , parameter-declaration*

*parameter-declaration:*  
*declaration-specifiers declarator*  
*type-name*

*identifier-list:*  
*identifier*  
*identifier-list , identifier*

### Semantics

Each declarator declares one identifier, and asserts that when a construction of the same form as the declarator appears in an expression, it yields an entity of the indicated scope, storage duration, and type.

If an unadorned identifier appears as a declarator, it has the type indicated by the type specifiers heading the declaration.

A declarator in parentheses is identical to the unadorned declarator, but the binding of complex declarators may be altered by parentheses.

In the following subsections, consider a declaration

**T D1**

where **T** is a type specifier (such as **int**) and **D1** is a declarator. Suppose this declaration makes the identifier have type *type-specifier T*, where the *type-specifier* is empty if **D1** is just a plain identifier (so that the type of **x** in “**int x**” is just **int**).

#### A.7.3.1 Pointer declarators

##### Constraints

Only **const** and **volatile** may appear in the list of type specifiers.

##### Semantics

If **D1** has the form

\* *type-specifier-list* *opt* **D**

the contained identifier is a *type-specifier* pointer to *T*. If the type specifier list includes **const**, the identifier is a constant pointer. If the type specifier list includes **volatile**, the identifier is a volatile pointer.

### Examples

The following pair of declarations demonstrates the difference between a variable pointer to a constant value and a constant pointer to a variable value.

```
const int *ptr_to_constant;
int *const constant_ptr;
```

The contents of the array pointed to by **ptr\_to\_constant** should not be modified, but **ptr\_to\_constant** itself may be changed to point to another **const int**. Similarly, the contents of the array pointed to by **constant\_ptr** may be modified, but **constant\_ptr** itself always points to the same location.

The declaration of the constant pointer **constant\_ptr** may be clarified by including a definition for a pointer to **int**.

```
typedef int *int_ptr;
const int_ptr constant_ptr;
```

## A.7.3.2 Array declarators

### Constraints

The constant expression that specifies the size of an array has integral type and value greater than zero. It is present, except that the first size may be omitted when an array is being declared as a formal parameter of a function, or when the array declaration has storage-class specifier **extern** and the definition that actually allocates storage is given elsewhere. It may also be omitted when the declarator is followed by initialization; in this case, the size is determined by the number of initializers supplied.

### Semantics

If **D1** has the form

$$D \text{ [} \textit{constant-expression opt} \text{ ]}$$

the contained identifier is an array of *type-specifier* with *T* elements. When several *array of* specifications are adjacent, a multi-dimensional array is declared.

Two declarations of an array with file **scope**, one of which has a missing first constant expression and the other of which has a specified size, are compatible; the size of the array is known from the completion of the declarator that specifies the size to the end of the source file.

### Examples

```
long fa [11], *afp[17];
```

declares an array of **long** numbers and an array of pointers to **long** numbers.

Note the distinction between the declarations

```
extern int *x;  
extern int y[];
```

The first declares **x** to be a pointer to **int**; the second declares **y** to be an array of **int** of unspecified size, the storage for which is defined elsewhere.

### A.7.3.3 Function declarators (including prototypes)

#### Constraints

All declarators of a particular function must agree in the type returned.

#### Semantics

If **D1** has the form

$$D(\textit{parameter-type-list})$$

or

$$D(\textit{identifier-list}_{opt})$$

the contained identifier is a *type-specifier* function returning *T*.

A parameter type list declares the types of, and optional identifiers for, the formal parameters of the function. If the list terminates with an ellipsis (*, ...*), no information about the number of types of the parameters after the comma is supplied.

In a declaration that is a function definition, identifiers are present for every parameter. The only storage-class specifier allowed is **register**, which suggests that the corresponding parameter be stored in a machine register if possible. If the list is empty, the function has no parameters.

In a declaration that is not a function, any identifier declared in the list has function prototype scope, which extends to the end of the declaration. The only storage-class specifier allowed is **register**, which is ignored. The special case of **void** as the only item in the list specifies that the function has no parameters.

The definition and every prototype of a function must agree in the number and types of the parameters and in the use of the ellipsis terminator. The parameter type checking includes agreement on the number of dimensions of arrays and on the bounds for each dimension except the first. Function declarators with empty lists may also appear.

An identifier list declares only the identifiers of the formal parameters of the function. If the list is empty in a function declaration that is not part of a function definition, no information about the number or types of the parameters is supplied. If the list is empty in a function declaration that is part of a function definition, the function has no parameters.

### Examples

The declaration

```
static int f(void), *fip(), (*pfi) ();
```

declares a function **f** with no parameters returning an **int**, a function **fip** with no parameter-type information returning a pointer to an **int**, and a pointer **pfi** to a function with no parameter-type information returning an **int**. It is especially useful to compare the last two.

The binding of **\*fip()** is **\*(fip())**, so that the declaration suggests, and the same construction in an expression requires, the calling of a function **fip**, and then using indirection through the pointer result to yield an **int**. In the declarator **(\*pfi)()**, the extra parentheses are necessary to indicate that indirection through a pointer to a function yields a function locator, which is then used to call the function; it returns an **int**.

The identifiers of the functions **f** and **fip** are declared as having file scope (not program scope) and internal linkage. The pointer **pfi** is declared as having file scope and internal linkage if the declaration is outside of any function, block scope and no linkage if the declaration is inside a function.

Here are two more intricate examples.

```
int (*apfi[])(int *x, int *y);
```

declares an array **apfi** of pointers to functions returning **int**. Each of these functions has two parameters that are pointers to **int**. The identifiers **x** and **y** are

declared for descriptive purposes only and go out of scope at the end of the declaration **apfi**. The declaration

```
int (*fpfi(int (*)(long), int))(int, ...);
```

declares a function **fpfi** that returns a pointer to a function returning an **int**. The function **fpfi** has two parameters; a pointer to a function returning an **int** (with one parameter of type **long**), and an **int**. The pointer returned by **fpfi** points to a function that has at least one parameter, which has type **int**.

## A.7.4 Type names

### Syntax

*type-name:*

*type-specifier-list abstract-declarator opt*

*abstract-declarator:*

*pointer*

*pointer opt direct-abstract-declarator*

*direct-abstract-declarator:*

*( abstract-declarator )*

*direct-abstract-declarator opt [ constant-expression opt ]*

*direct-abstract-declarator opt ( parameter-type-list opt )*

### Semantics

The user may need to specify a type in several situations. Examples include specifying

a type conversion with **cast** (Appendix A, section "Expressions", subsection, "Multiplicative Operators"),

the type of the operand of the **sizeof** operator (Appendix A, section "Expressions", subsection "Unary operators"), and

the type of a formal parameter in a function declarator (section "Declarations", subsection "Function Declarations").

The user can specify a type with a *type name*, which is syntactically a declaration for an object of that type that omits the identifier of the object.

### Examples

Declaration	Type
<b>int</b>	<b>int</b>
<b>int *</b>	Pointer to <b>int</b>
<b>int *[3]</b>	Array of three pointers to <b>int</b>
<b>int (*) [3]</b>	Pointer to an array of three <b>ints</b>
<b>int *()</b>	Function with no parameter type information returning a pointer to <b>int</b>
<b>int (*)(void)</b>	Pointer to a function with no parameters returning an <b>int</b>
<b>int (*const [])(unsigned int, ...)</b>	Array of an unspecified number of constant pointers to functions, each with one parameter that has type <b>unsigned int</b> and an unspecified number of other parameters, returning an <b>int</b> .

### A.7.5 Type definitions and type equivalence

#### Syntax

*typedef-name:*  
*identifier*

#### Semantics

With the scope of a declaration whose storage-class specifier is **typedef**, each identifier declared becomes equivalent to a type specifier naming the type associated with the identifier as described in section "Declarations", subsection "Declarators."

A **typedef** declaration does not introduce a brand-new type, but a synonym for a type that could be specified in another way. Two types are the same if they have the same ordered set of type specifiers and abstract declarators, either directly or through **typedefs**. Two structures or unions are different if they have

no names or different names, taking tags and typedef synonyms into account, even if their members are identical.

A **typedef** name shares the same name space as other identifiers declared in ordinary declarators. It may be redeclared in an inner block, but the type may not be omitted in the inner declaration.

### Examples

After

```
typedef int MILES, KLICKSP();  
typedef struct { double re, im; } complex;
```

the constructions

```
MILES distance;  
extern KLICKSP *metricp;  
complex x;  
complex z, *zp;
```

are all valid declarations. The type of **distance** is **int**, that of **metricp** is “pointer to function with no parameter-type information returning **int**”, and that of **x** and **z** is the specified structure; **zp** is a pointer to such a structure. The object **distance** is considered to have exactly the same type as any other **int** object.

After the declarations

```
typedef struct s1 { int x; } t1, *tp1;  
typedef struct s2 { int x; } t2, *tp2;
```

type **t1** and the type pointed to by **tp1** are equivalent to each other and to the type **struct s1**, but different from the types **struct s2** and **t2**, the type pointed to by **tp2**, and type **int**.

## A.7.6 Initialization

### Syntax

*initializer:*

```
assignment-expression  
{ initializer-list }  
{ initializer-list , }
```

*initializer-list:*

```
initializer  
initializer-list , initializer
```

### Semantics



Any declaration that defines an object may include an initializer that specifies its initial value. The initializers in a declaration list are evaluated in the order they appear in the source file.

For a union object, the initializer initializes the member that appears first in the declaration list of the union type.

An object that has static storage duration may be initialized by constant expressions only. If such an object is not initialized explicitly, it is initialized implicitly as if every member that has arithmetic type were assigned 0 and every member that has pointer type were assigned a null pointer constant.

Except as noted below for aggregates, an object that has automatic storage duration may be initialized by arbitrary expressions involving constants, previously declared identifiers, function calls, and assignments. Such an object is not initialized implicitly; if the value is used before one is assigned, the behavior is undefined.

If the declared object is a scalar, the initializer is a single expression, optionally in braces. The initial value of the object is that of the expression; the same conversions as for assignment are performed.

#### **A.7.6.1 Initialization of aggregate objects**

If the declared object is a structure object that has automatic storage duration, the initializer may be a single expression that has the same type. The initial value of the object is that of the expression.

If the declared object is an array of **chars**, the initializer may be a string literal. Successive characters of the string literal (including the terminating null character if there is room or if no size for the array is specified) initialize the members of the array.

Otherwise, the initializer is a brace-enclosed list of initializers for the members of the aggregate, written in increasing subscript or member order. All expressions in such a list are constant expressions.

If the aggregate contains members that are aggregates, these rules apply recursively to the subaggregates. If the initializer of a subaggregate begins with a left brace, the succeeding initializers initialize the members of the subaggregate. Otherwise, only enough initializers from the list are taken to account for the members of the first subaggregate; any remaining initializers are left to initialize the next member of the subaggregate of which the current aggregate is a part.

No more initializers exist in a list than there are members of an aggregate. If fewer initializers exist in a list than there are members of an aggregate, the remainder of the aggregate is initialized as if it had storage duration.

### Examples

The declaration

```
int x[] = { 1, 3, 5 };
```

defines and initializes **x** as a one-dimensional array object that has three members, since no size was specified and three initializers exist.

```
int y[4][3] = {
    { 1, 3, 5 },
    { 2, 4, 6 },
    { 3, 5, 7 },
};
```

is a definition with a fully bracketed initialization: 1, 3, and 5 initialize the first row of the array object **y[0]**, namely **y[0][0]**, **y[0][1]**, **y[0][2]**. Likewise the next two lines initialize **y[1]** and **y[2]**. The initializer ends early, so **y[3]** is initialized with zeros. The code below achieves the same effect.

```
int y[4][3] = {
    1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

The initializer for **y[0]** does not begin with a left brace, so three items from the list are used. Likewise the next three are taken successively for **y[1]** and **y[2]**. Also,

```
int z[4][3] = {
    { 1 }, { 2 }, { 3 }, { 4 }
};
```

initializes the first column of **z** as specified and initializes the rest with zeros.

```
struct { int a[3], b; } w[] = { { 1 }, 2 };
```

is a definition with a partly bracketed initialization, which is not recommended programming practice. It defines an array with two member structures: **w[0].a[0]** is **1** and **w[1].a[0]** is **2**. All the other elements are zero.

Finally, the declaration

```
char s[] = "abc", t[3] = "abc";
```

defines character array objects **s** and **t** whose members are initialized with string literals. This declaration is identical to

```
char s[] = { 'a', 'b', 'c', '\0' },
          t[] = { 'a', 'b', 'c' };
```

The contents of the arrays are modifiable. On the other hand, the declaration

```
char *p = "abc";
```

defines a character pointer **p** that is initialized to point to a character array object whose members are initialized with a string literal. If an attempt is made to use **p** to modify the contents of the array, the behavior is undefined.

## A.8 Statements

### Syntax

```
statement:
    labeled-statement
    compound-statement
    expression-statement
    selection-statement
    iteration-statement
    jump-statement
```

### Semantics

A *statement* specifies an action to perform. Except as indicated, statements are executed in sequence.

A *full expression* is an expression that is not part of another expression. Each of the following is a full expression:

an initializer of an object that has automatic storage duration;

the expression in an expression statement;

the controlling expression of an selection statement (**if** or **switch**); and

the controlling expression of an iteration statement (**while**, **do**, or **for**).

Completion of the evaluation of a full expression is a sequence point.

### A.8.1 Labeled statements

#### Syntax

*labeled-statement:*  
*identifier : statement*  
**case** *constant-expression : statement*  
**default** : *statement*

#### Constraints

A named label may be used only as a target of a **goto** statement. A case or default label may appear only in a **switch** statement. The constraints on such labels are discussed under the **switch** statement.

#### Semantics

Any statement may be preceded by a prefix that declares an identifier as a label name. Labels in themselves do not alter the flow of control, which continues unimpeded across them.

### A.8.2 Compound statement, or block

#### Syntax

*compound-statement:*  
*{ declaration-list opt statement-list opt }*

*declaration-list:*  
*declaration*  
*declaration-list declaration*

*statement-list:*  
*statement*  
*statement-list statement*

#### Constraints

An object declared **extern** inside a block is not initialized in the declaration as storage since it is defined elsewhere.

### Semantics

A *compound statement* (also called a *block*) allows a set of statements to be grouped into one syntactic unit, which may have its own set of declarations and initializers (as discussed in section "Lexical Elements", subsection "Storage durations of objects.")

## A.8.3 Expression and null statements

### Syntax

*expression-statement:*  
*expression* *opt* ;

### Semantics

An expression statement, consisting of an expression followed by a semicolon, is evaluated for its side effects, such as assignments and function calls.

A *null statement*, consisting of just a semicolon, has no expression and performs no operations.

### Examples

If a function call is evaluated as an expression statement for its side effects only, the discarding of its value may be made explicit by converting the expression to a void expression with a cast:

```
int p(int);
/*...*/
(void) p(0);
```

In the program fragment

```
char *s;
/*...*/
while (*s++ != '\0')
    ;
```

a null statement supplies an empty loop body to the iteration statement.

A null statement may also carry a label just before the closing } of a compound statement.

```
while (loop1) {
    /*...*/
```

```
        while (loop2) {  
            /*...*/  
            if (want_out)  
                goto end_loop1;  
            /*...*/  
        }  
        /*...*/  
end_loop1: ;  
}
```

## A.8.4 Selection statements

### Syntax

*selection-statement:*  
**if** ( *expression* ) *statement*  
**if** ( *expression* ) *statement* **else** *statement*  
**switch** ( *expression* ) *statement*

### Semantics

A selection statement selects among a set of statements depending on the value of a controlling expression.

#### A.8.4.1 The if statement

##### Constraints

The controlling expression of an **if** statement has scalar type.

##### Semantics

In both forms, the first substatement is executed if the expression evaluates to non-zero. In the **else** form, the second substatement is executed if the expression evaluates to zero. If the first substatement is reached through a label, the second substatement is not executed.

An **else** is associated with the lexically immediately preceding **else**-less **if** that is in the same block (but not in the enclosed block).

#### A.8.4.2 The switch statement

##### Constraints

The controlling expression of a **switch** statement and the constant expression of each **case** label has integral type. No two of the **case** constants in the same **switch**

statement may have the same value after conversion. There may be at most one **default** label in a **switch** statement (but any enclosed **switch** statement may also have a **default** label).

### Semantics

A **switch** statement causes control to jump to, into, or past the statement that is the *switch body*, depending on the value of a controlling expression and the values of any **case** prefixes on or in the switch body. If there are nested **switch** statements within the switch body, any **default** or **case** labels in them are ignored.

The integral promotions are performed on the controlling expression. The constant expression in each **case** label is converted to the type of the promoted controlling expression. If a converted value matches that of the promoted controlling expression, control jumps to the statement following the matched **case** prefix. Otherwise, if there is a **default** label, control jumps to the labeled statement. If no converted **case** constant matches and there is no **default** label, none of the statements in the switch body are executed.

## A.8.5 Iteration statements

### Syntax

*iteration-statement:*

**while** ( *expression* ) *statement*

**do** *statement* **while** ( *expression* ) ;

**for** ( *expression* *opt* ; *expression* *opt* ; *expression* *opt* ) *statement*

### Constraints

The controlling expression of an iteration statement has scalar type.

### Semantics

An iteration statement repeatedly executes a statement called the *loop body* until the controlling expression evaluates to zero.

#### A.8.5.1 The while statement

The evaluation of the controlling expression takes place before each execution of the loop body.

#### A.8.5.2 The do statement

The evaluation of the controlling expression takes place after each execution of the loop body.

### A.8.5.3 The for statement

Except for the behavior of a **continue** statement in the loop body, the statement

**for** (*expression-1* ; *expression-2* ; *expression-3* ) *statement*

is equivalent to

```
expression-1 ;  
while (expression-2) {  
    statement  
    expression-3 ;  
}
```

Thus *expression-1* specifies initialization for the loop;

*expression-2* specifies an evaluation made before each iteration, such that execution of the loop continues until the expression evaluates to zero; and

*expression-3* specifies an operation (such as incrementing) that is performed after each iteration.

Both *expression-1* and *expression-3* may be omitted, or may have any type, or may be void expressions. An omitted *expression-2* is replaced by a non-zero constant.

## A.8.6 Jump statements

### Syntax

```
jump-statement:  
    goto identifier ;  
    continue ;  
    break ;  
    return expression opt ;
```

### Semantics

A jump statement causes an unconditional jump to another place.

### A.8.6.1 The goto statement

#### Constraints

The identifier in a **goto** statement names a label located somewhere in the current function.



## Semantics

A **goto** statement causes an unconditional jump to the named label in the current function.

### A.8.6.2 The continue statement

#### Constraints

A **continue** statement may appear only in a loop body.

#### Semantics

A **continue** statement causes a jump to the loop-continuation portion of the smallest enclosing iteration statement; that is, to the end of the loop body. More precisely, in each of the statements

while ( <i>/*...*/</i> ) {	do {	for ( <i>/*...*/</i> ) {
<i>/*...*/</i>	<i>/*...*/</i>	
continue;	continue;	continue;
<i>/*...*/</i>	<i>/*...*/</i>	<i>/*...*/</i>
contin: ;	contin: ;	contin: ;
}	} while ( <i>/*...*/</i> );	}

unless the **continue** statement shown is in an enclosed iteration statement (in which case it is interpreted within that statement), it is equivalent to **goto contin:**. (Following the **contin:** is a null statement.)

### A.8.6.3 The break statement

#### Constraints

A **break** statement may appear only in a switch body or loop body.

#### Semantics

A **break** statement terminates execution of the smallest enclosing **switch** or iteration statement.

### A.8.6.4 The return statement

#### Constraints

A **return** statement with an expression may not appear in a function declared as returning type **void**.

#### Semantics

A **return** statement terminates execution of the current function and returns control to its caller. A function may have any number of **return** statements, with and without expressions.

If a **return** statement with an expression is executed, the value of the expression is returned to the caller. If the expression has a type different from that of the function in which it appears, it is converted as if it were assigned to an object of that type.

If a **return** statement without an expression is executed, and the value of the function call is used by the caller, the behavior is undefined. Reaching the **}** that terminates a function is equivalent to executing a **return** statement without an expression.

## A.9 External Definitions

### Syntax

*file:*  
    *external-definition*  
    *file external-definition*

*external-definition:*  
    *function-definition*  
    *declaration*

### Constraints

If present, the storage-class specifier in an external definition must be **extern** (which is the default and declares external linkage) or **static** (which declares internal linkage). As with other declarations, if there are no type specifiers the type must be taken to be **int**.

### Semantics

The unit of program text is a file, which consists of a sequence of external definitions of functions and objects (and other declarations, such as type declarations) described as “external” because they appear outside of any function. The syntax of an external definition is the same as that of a declaration, except that the code for a function (the function body) appears outside of any other function.

An *external definition* implicitly declares its identifier to have file scope and static storage duration.

## A.9.1 Function definitions

### Syntax

*function-definition:*  
*declaration-specifiers* *opt* *declarator* *declaration-list* *opt*  
*compound-statement*

### Constraints

As specified only by the declarator portion of a function definition, the first identifier (which is the name of the function) has type *function*.

If the declarator is a function prototype declarator, no declaration list may follow. If the declarator is an identifier list, only the identifiers it names may be declared in the declaration list; an identifier declared as a typedef name must not be redeclared as a formal parameter. The only storage-class specifier allowed in the declaration of a formal parameter is **register**.

### Semantics

The declarator in a function definition specifies the name of the function being defined and lists the identifiers of its formal parameters. If the declarator has the form of a function prototype declarator, the list also declares the types of all the formal parameters. A declarator also serves as a function prototype. If the declarator is an identifier list, the types of the formal parameters may be declared

in a following declaration list. Any formal parameter whose type is not declared in the declaration list is assumed to have type **int**.

If a function that accepts a variable number of arguments is defined without the ellipsis notation in its prototype, the behavior is undefined.

If any formal parameter is declared with a type that is affected by the default argument conversions, the following applies:

If a function prototype is in scope when the function is defined, and a call is executed for which no semantically equivalent function prototype is in scope, the behavior is undefined.

If no function prototype is in scope when the function is defined, and a call is executed for which a function prototype is in scope that declares other than the appropriate widened types (according to the default argument conversions), the behavior is undefined.

On entry to the function the value of the argument expression is converted to the type of the formal parameter, as if by assignment to the formal parameter.

Because array expressions and function identifiers as arguments are always converted to pointers before the call, a declaration of a formal parameter as an array of *type* always is adjusted to a pointer to *type*, and a declaration of a formal parameter as a function returning *type* is always adjusted to a pointer to a function returning *type*.

Each formal parameter is treated as having automatic storage duration and is an lvalue. The layout of the storage for formal parameters is unspecified.

### Examples

```
extern int max (int a, int b)
{
    return a > b ? a : b;
}
```

Here **extern** is the storage-class specifier and **int** is the type specifier (each of which may be omitted as those are the defaults); **max(int a, int b)** is the function declarator; and

```
{ return a > b ? a : b; }
```

is the function body. The following equivalent definition uses the identifier-list form for the parameter declarations:

```
extern int max(a, b)
int a, b;
{
    return a > b ? a : b;
}
```

Here **int a, b;** is the declaration list for the formal parameters, which may be omitted as those are the defaults.

The reason for this constraint is so that the type in a function definition cannot be inherited from a typedef:

```
typedef int F(void);
/* type F is "function of no arguments returning int" */
F f { /*...*/ }
/* WRONG: implies that f returns a function */
int f(void) { /*...*/ }
/* RIGHT: f has the same type as F */
int g() { /*...*/ }
/* RIGHT: g has the same type as F */
F *f(void) { /*...*/ }
/* f returns a pointer to a function */
F * ((f))(void) { /*...*/ }
/* same: parentheses irrelevant */
int (*fp)(void);
/* fp points to a function that has type F */
F *Fp;
/* Fp points to a function that has type F */
```

To pass one function to another, the user might say

```
int f(void);
/*...*/
g(f);
```

Note that **f** is declared explicitly in the calling function, as its appearance in the expression **g(f)** was not followed by **(**. Then the definition of **g** might read

```
g(int (*funct)(void))
{
    /*...*/ (*funct)() /*...*/
}
```

or, equivalently,

```
g(int func(void))
{
    /*...*/ func() /*...*/
}
```

## A.9.2 External object definitions

### Constraints

One external definition should exist for each object declared with the storage-class specifier **extern** that is referred to in any way.

### Semantics

A declaration of an identifier of an object outside of any function that includes an initializer constitutes the definition of the object.

A declaration of an identifier of an object outside of any function without an initializer, and without a storage-class specifier or with the storage-class specifier **static**, constitutes a *tentative definition*. If a definition is encountered later in the source file, all tentative definitions are taken to be declarations of the same object. If no subsequent definition is encountered, the first tentative definition is taken to be a definition with initializer equal to 0.

### Examples

```
int i1 = 1;           /* definition, external linkage */
static int i2 = 2;    /* definition, internal linkage */
extern int i3 = 3;    /* definition, external linkage */
int i4;               /* tentative definition, external linkage */
static int i5;        /* tentative definition, internal linkage */
```

```
int i1;          /* valid tentative definition, refers to previous */
int i2;          /* violates "linkages of identifiers" in section
                  "Lexical Elements", subsection " Identifiers"
                  on linkage disagreement */

int i3;          /* valid tentative definition, refers to previous */
int i4;          /* valid tentative definition, refers to previous */
int i5;          /* violates "linkages of identifiers" in section
                  "Lexical Elements", subsection " Identifiers"
                  on linkage disagreement */

extern int i1;    /* refers to previous, linkage is external */
extern int i2;    /* refers to previous, linkage is internal */
extern int i3;    /* refers to previous, linkage is external */
extern int i4;    /* refers to previous, linkage is external */
extern int i5;    /* refers to previous, linkage is internal */
```

# Language Syntax Summary

## Appendix B

The notation is described in the introduction to Appendix A, “Language.”

## B.1 Lexical Grammar

### B.1.1 Tokens

*token:*

*keyword*  
*identifier*  
*constant*  
*string-literal*  
*operator*  
*punctuator*

### B.1.2 Keywords

*keyword:* one of

auto	do	fract	reentrant*	switch
bit**	double	goto	register	typedef
break	else	huge*	return	union
case	enum	if	short	unsigned
char	extern	int	signed	void
const	far*	interrupt*	sizeof	volatile
continue	float	long	static	while
default	for	near*	struct	

\* Language extension for embedded application



### B.1.3 Identifiers

*identifier:*

*nondigit*

*identifier nondigit*

*identifier digit*

*nondigit:* one of

a b c d e f g h i j k l m

n o p q r s t u v w x y z

A B C D E F G H I J K L M

N O P Q R S T U V W X Y Z

*digit:* one of

0 1 2 3 4 5 6 7 8 9

### B.1.4 Constants

*constant:*

*floating-constant*

*integer-constant*

*enumeration-constant*

*character-constant*

*floating-constant:*

*fractional-constant exponent-part opt floating-suffix opt*

*digit-sequence exponent-part floating-suffix opt*

*fractional-constant:*

*digit-sequence opt . digit-sequence*

*digit-sequence .*

*exponent-part:*

**e** *sign opt digit-sequence*

**E** *sign opt digit-sequence*

*sign:* one of

+ -

*digit-sequence:*

*digit*

*digit-sequence digit*

*floating-suffix*: one of  
**f l F L**

*integer-constant*:  
    *decimal-constant integer-suffix opt*  
    *octal-constant integer-suffix opt*  
    *hexadecimal-constant integer-suffix opt*

*decimal-constant*:  
    *nonzero-digit*  
    *decimal-constant digit*

*octal-constant*:  
    **0**  
    *octal-constant octal-digit*

*hexadecimal-constant*:  
    **0x** *hexadecimal-digit*  
    **0X** *hexadecimal-digit*  
    *hexadecimal-constant hexadecimal-digit*

*nonzero-digit*: one of  
    1 2 3 4 5 6 7 8 9

*octal-digit*: one of  
    0 1 2 3 4 5 6 7

*hexadecimal-digit*: one of  
    0 1 2 3 4 5 6 7 8 9  
    a b c d e f  
    A B C D E F

*integer-suffix*:  
    *unsigned-suffix long-suffix opt*  
    *long-suffix unsigned-suffix opt*

*unsigned-suffix*: one of  
    **u U**

*long-suffix*: one of  
    **l L**

*enumeration-constant*:  
    *identifier*

*character-constant:*

**L** *opt* 'c-char-sequence'

*c-char-sequence:*

*c-char*

*c-char-sequence c-char*

*c-char:*

any character in the source character set except  
the single quote ('), backslash (\), or new-line character

*escape-sequence*

*escape-sequence:* one of

|' |" |? ||

|o |oo |ooo

|xh |xhh |xhhh

|a |b |f |n |r |t |v

## B.1.5 String literals

*string-literal:*

**L** *opt* "s-char-sequence" *opt*

*s-char-sequence:*

*s-char*

*s-char-sequence s-char*

*s-char:*

any character in the source character set except  
the double-quote ("), backslash (\), or new-line character

*escape-sequence*

## B.1.6 Operators

*operator:* one of

[ ] ( ) . ->

++ -- & \* + - ~ ! sizeof

/ % << >> < > <= >= == != ^ | && ||

? :

= \*= /= %= += -= <<= >>= &= ^= |=

, # ##

## B.1.7 Punctuators

*punctuator*: one of  
 [ ] ( ) { } \* , : = ; ... #

## B.2 Phrase Structure Grammar

### B.2.1 Expressions

*primary-expression*:  
*identifier*  
*constant*  
*string-literal*  
 ( *expression* )

*postfix-expression*:  
*primary-expression*  
*postfix-expression* [ *expression* ]  
*postfix-expression* ( *argument-expression-list* *opt* )  
*postfix-expression* . *identifier*  
*postfix-expression* -> *identifier*  
*postfix-expression* ++  
*postfix-expression* --

*argument-expression-list*:  
*assignment-expression*  
*argument-expression-list* , *assignment-expression*

*unary-expression*:  
*postfix-expression*  
 ++ *unary-expression*  
 -- *unary-expression*  
*unary-operator* *cast-expression*  
**sizeof** *unary-expression*  
**sizeof** ( *type-name* )

*unary-operator*: one of  
 \* & + - ~ !

*cast-expression*:  
*unary-expression*  
 ( *type-name* ) *cast-expression*

*multiplicative-expression:*

*cast-expression*  
*multiplicative-expression* \* *cast-expression*  
*multiplicative-expression* / *cast-expression*  
*multiplicative-expression* % *cast-expression*

*additive-expression:*

*multiplicative-expression*  
*additive-expression* + *multiplicative-expression*  
*additive-expression* - *multiplicative-expression*

*shift-expression:*

*additive-expression*  
*shift-expression* << *additive-expression*  
*shift-expression* >> *additive-expression*

*relational-expression:*

*shift-expression*  
*relational-expression* < *shift-expression*  
*relational-expression* > *shift-expression*  
*relational-expression* <= *shift-expression*  
*relational-expression* >= *shift-expression*

*equality-expression:*

*relational-expression*  
*equality-expression* == *relational-expression*  
*equality-expression* != *relational-expression*

*AND-expression:*

*equality-expression*  
*AND-expression* & *equality-expression*

*exclusive-OR-expression:*

*AND-expression*  
*exclusive-OR-expression* ^ *AND-expression*

*inclusive-OR-expression:*

*exclusive-OR-expression*  
*inclusive-OR-expression* | *exclusive-OR-expression*

*logical-AND-expression:*

*inclusive-OR-expression*  
*logical-AND-expression* && *inclusive-OR-expression*

*logical-OR-expression:*

*logical-AND-expression*

*logical-OR-expression* *||* *logical-AND-expression*

*conditional-expression:*

*logical-OR-expression*

*logical-OR-expression* *?* *expression* *:* *conditional expression*

*assignment-expression:*

*conditional-expression*

*unary-expression* *assignment-operator* *assignment-expression*

*assignment-operator:* one of

*=* *\*=* */=* *%=* *+=* *-=* *<<=* *>>=* *&=* *^=* *|=*

*expression:*

*assignment-expression*

*expression* *,* *assignment-expression*

*constant-expression:*

*conditional-expression*

## B.2.2 Declarations

*declaration:*

*declaration-specifiers* *init-declarator-list* *opt* *;*

*declaration-specifiers:*

*storage-class-specifier* *declaration-specifiers* *opt*

*type-specifier* *declaration-specifiers* *opt*

*init-declarator-list:*

*init-declarator*

*init-declarator-list* *,* *init-declarator*

*init-declarator:*

*declarator*

*declarator* *=* *initializer*

*storage-class-specifier:*

*typedef*

*extern*

*static*

*auto*

*register*

*type-specifier:*

char  
short  
int  
long  
signed  
unsigned  
float  
double  
const  
volatile  
void  
huge  
far  
near  
*struct-or-union-specifier*  
*enum-specifier*  
*typedef-name*

*struct-or-union-specifier:*

*struct-or-union identifier* *opt* { *struct-declaration-list* }  
*struct-or-union identifier*

*struct-or-union:*

struct  
union

*struct-declaration-list:*

*struct-declaration*  
*struct-declaration-list struct-declaration*

*struct-declaration:*

*type-specifier-list struct-declaration ;*

*type-specifier-list:*

*type-specifier*  
*type-specifier-list type-specifier*

*struct-declarator-list:*

*struct-declarator*  
*struct-declarator-list , struct-declarator*

*struct-declarator:*

*declarator*  
*declarator* *opt* : *constant-expression*

*enum-specifier:*

*enum identifier* *opt* { *enumerator-list* }  
*enum identifier*

*enumerator-list:*

*enumerator*  
*enumerator-list* , *enumerator*

*enumerator:*

*enumeration-constant*  
*enumeration-constant* = *constant-expression*

*declarator:*

*pointer* *opt* *direct-declarator*

*direct-declarator:*

*identifier*  
(*declarator*)  
*direct-declarator* [ *constant-expression* *opt* ]  
*direct-declarator* ( *parameter-type-list* )  
*direct-declarator* ( *identifier-list* *opt* )

*pointer:*

\* *type-specifier-list* *opt*  
\* *type-specifier-list* *opt* *pointer*

*parameter-type-list:*

*parameter-list*  
*parameter-list* , ...

*parameter-list:*

*parameter-declaration*  
*parameter-list* , *parameter-declaration*

*parameter-declaration:*

*declaration-specifiers* *declarator*  
*type-name*

*identifier-list:*

*identifier*  
*identifier-list* , *identifier*

*type-name:*

*type-specifier-list* *abstract-declarator* *opt*



*abstract-declarator:*

*pointer*

*pointer* *opt* *direct-abstract-declarator*

*direct-abstract-declarator:*

( *abstract-declarator* )

*direct-abstract-declarator* *opt* [ *constant-expression* *opt* ]

*direct-abstract-declarator* *opt* ( *parameter-type-list* *opt* )

*typedef-name:*

*identifier*

*initializer:*

*assignment-expression*

{ *initializer-list* }

{ *initializer-list* , }

*initializer-list:*

*initializer*

*initializer-list* , *initializer*

### B.2.3 Statements

*statement:*

*labeled-statement*

*compound-statement*

*expression-statement*

*selection-statement*

*iteration-statement*

*jump-statement*

*labeled-statement:*

*identifier* : *statement*

**case** *constant-expression* : *statement*

**default** : *statement*

*compound-statement:*

{ *declaration-list* *opt* *statement-list* *opt* }

*declaration-list:*

*declaration*

*declaration-list* *declaration*

*statement-list:*

*statement*  
*statement-list statement*

*expression-statement:*

*expression opt ;*

*selection-statement:*

**if** ( *expression* ) *statement*  
**if** ( *expression* ) *statement* **else** *statement*  
**switch** ( *expression* ) *statement*

*iteration-statement:*

**while** ( *expression* ) *statement*  
**do** *statement* **while** ( *expression* ) ;  
**for** ( *expression opt ; expression opt ; expression opt* ) *statement*

*jump-statement:*

**goto** *identifier* ;  
**continue** ;  
**break** ;  
**return** *expression opt* ;

## B.2.4 External definitions

*file:*

*external-definition*  
*file external-definition*

*external-definition:*

*function-definition*  
*declaration*

*function-definition:*

*declaration-specifiers opt declarator function-body*

*function-body:*

*declaration-list opt compound-statement*

## B.3 Preprocessing Directives

*preprocessing-file:*

*group*  
*preprocessing-file--group*

*group:*

*group-part*  
*group group-part*

*group-part:*

*tokens opt new-line*  
*if-section*  
*control-line*

*if-section:*

*if-group elif-groups opt else-group opt endif-line*

*if-group:*

**# if** *constant-expression new-line group opt*  
**# ifdef** *identifier new-line group opt*  
**# ifndef** *identifier new-line group opt*

*elif-groups:*

*elif-group*  
*elif-groups elif-group*

*elif-group:*

**# elif** *constant-expression new-line group opt*

*else-group:*

**# else** *new-line group opt*

*endif-line:*

**# endif***new-line*

*control-line:*

**# include** *<tokens> new-line*  
**# include** *file-name new-line*  
**# include** *tokens new-line*  
**# define** *identifier tokens opt new-line*  
**# define** *identifier lparen identifier-list opt )tokens opt new-line*  
**# undef** *identifier new-line*  
**# line** *digit-sequence file-name opt new-line*  
**# error** *tokens opt new-line*  
**# pragma** *tokens opt new-line*  
**#** *new-line*

*file-name:*

*“tokens”*

*lparen:*

the left-parenthesis character without preceding white-space

*tokens:*

*token*

*tokens token*

*new-line:*

the new-line character

This page is left blank intentionally.

# Compiler Preprocessor

## Appendix C

### C.1 Preprocessing Directives

#### Syntax

*preprocessing-file:*  
*group opt*

*group:*  
*group-part*  
*group group-part*

*group-part:*  
*pp-tokens opt new-line*  
*if-section*  
*control-line*

*if-section:*  
*if-group elif-group opt else-group opt endif-line*

*if-group:*  
**# if** *constant-expression new-line group<sub>opt</sub>*  
**# ifdef** *identifier new-line group<sub>opt</sub>*  
**# ifndef** *identifier new-line group<sub>opt</sub>*

*elif-groups:*  
*elif-group*  
*elif-groups elif-group*

*elif-group:*  
**# elif** *constant-expression new-line group<sub>opt</sub>*

*else-group:*  
**# else** *new-line group<sub>opt</sub>*

*endif-line:*  
**# endif** *new-line*

*control-line:*

<b># include</b>	<i>pp-tokens new-line</i>
<b># define</b>	<i>identifier replacement-list new-line</i>
<b># define</b>	<i>identifier lparen identifier-list <i>opt</i> )</i>
	<i>replacement-list new-line</i>
<b># undef</b>	<i>identifier new-line</i>
<b># line</b>	<i>pp-tokens new-line</i>
<b># error</b>	<i>pp-tokens <i>opt</i>-new-line</i>
<b># pragma</b>	<i>pp-tokens <i>opt</i>-new-line</i>
<b>#</b>	<i>new-line</i>

*lparen:*

the left-parenthesis character without preceding white space

*replacement-list:*

*pp-tokens *opt**

*pp-tokens:*

*preprocessing-token*  
*pp-tokens preprocessing-token*

*new-line:*

the new-line character

## Descriptions

A preprocessing directive consists of a sequence of preprocessing tokens that begins with a **#** preprocessing token that is the first character in the source file (after any white space containing no new-line characters) or that follows white space containing at least one new-line character, and ended by the next new-line character.

## Constraints

The only white-space characters that should appear between preprocessing tokens within a preprocessing directive are space and horizontal-tabs.

### Semantics

The implementation can process and skip sections of source files conditionally, include other source files, and replace macros. These capabilities are called *preprocessing*, because conceptually they occur before translation of the resulting translation unit.

The preprocessing tokens within a preprocessing directive are not subject to macro expansion unless otherwise stated.

## C.2 Conditional Inclusion

### Constraints

Additional restrictions apply to a constant expression that controls conditional inclusion. The expression must be an integral constant expression that must not contain a `sizeof` operator, a cast, or an enumeration constant. It may contain unary expressions of the form

`defined identifier`

or

`defined ( identifier )`

which evaluate to 1 if the identifier is currently defined as a macro name (that is, if it is predefined or if it has been the subject of a **#define** preprocessing directive without an intervening **#undef** directive). If not, the expression evaluates to 0.

Each preprocessing token that remains after all macro replacements have occurred is in the lexical form of a token.

### Semantics

Preprocessing directives of the forms

**# if**     *constant-expression new-line group opt*  
**# elif**     *constant-expression new-line group opt*

check whether the controlling constant expression evaluates to nonzero.

Prior to evaluation, macro invocations in the list of preprocessing tokens that will become the controlling constant expression are replaced (except for those macro names modified by **defined**), just as in normal text. The **defined** operator explicitly appears in the original list of preprocessing tokens. After all replace-



ments are finished, the resulting preprocessing tokens are converted into tokens. Then, before the controlling constant expression is evaluated, all remaining identifiers are replaced with **OL** and each integer constant not already suffixed with **I** or **L** is considered to be additionally suffixed with **L**. This includes interpreting character constants, which may involve converting escape sequences into characters.

Preprocessing directives of the forms

```
# ifdef identifier new-line group opt  
# ifndef identifier new-line group opt
```

check whether the identifier is or is not currently defined as a macro name. Their conditions are equivalent to **#if defined** *identifier* and **#if !defined** *identifier* respectively.

Each directive's condition is checked in order. If it evaluates to false (zero), the group that it controls is skipped. Directives are processed only through the name that determines the directive in order to keep track of the level of nested conditionals; the rest of the directives' preprocessing tokens are ignored, as are the other preprocessing tokens in the group. Only the first group whose control condition evaluates to true (non-zero) is processed. If none of the conditions evaluates to true, and there is a **#else** directive, the group controlled by the **#else** is processed; lacking a **#else** directive, all the groups until the **#endif** are skipped.

## C.3 Source File Inclusion

### Constraints

A **#include** directive identifies a header or source file that the implementation can process.

### Semantics

A preprocessing directive of the form

```
# include <h-char-sequence> new-line
```

searches a sequence of places for a header identified uniquely by the specified character sequence between the **<** and **>** delimiters, and causes the replacement of that directive by the entire contents of the header.

A preprocessing directive of the form

```
# include "q-char-sequence" new-line  
causes the replacement of that directive by the entire contents of the source file identified by the specified character sequence between the " delimiters. The
```

named source file is searched for. If this search fails, the directive is reprocessed as if it read

**# include** *<h-char-sequence> new-line*

with the identical contained character sequence (including > characters, if any) from the original directive.

A preprocessing directive of the form

**# include** *pp-token new-line*

(that does not match one of the two previous forms) is permitted. The preprocessing tokens after **#include** in the directive are processed just as in normal text. (Each identifier currently defined as a macro name is replaced by its replacement list of preprocessing tokens.) The directive resulting after all replacements matches one of the two previous forms.

A **#include** preprocessing directive may appear in a source file that has been read because of a **#include** directive in another file.

## Examples

The most common uses of **#include** processing directives are as in the following:

```
#include <stdio.h>
#include "myprog.h"
```

This example illustrates a macro-replaced **#include** directive:

```
#if VERSION == 1
    #define INCFILE "vers1.h"
#elif VERSION == 2
    #define INCFILE "vers2.h"    /* and so on */
#else
    #define INCFILE "versN.h"
#endif
#include INCFILE
```

## C.4 Macro Replacement

### Constraint

Two replacement lists are identical if, and only if, the preprocessing tokens in both have the same number, ordering, spelling, and white-space separation, where all white-space separations are considered identical.

An identifier currently defined as a macro without use of **lparen** (an *object-like* macro) may be redefined by another **#define** preprocessing directive provided, that the second definition is an object-like macro definition and the two replacement lists are identical.

An identifier currently defined as a macro using **lparen** (a *function-like* macro) may be redefined by another **#define** preprocessing directive provided, that the second definition is a function-like macro definition that has the same number and spelling of parameters, and the two replacement lists are identical.

The number of arguments in an invocation of a function-like macro agrees with the number of parameters in the macro definition, and a ‘)’ preprocessing token exists that terminates the invocation.

A parameter identifier in a function-like macro must be uniquely declared within its scope.

### Semantics

The identifier immediately following the **define** is called the *macro name*. Any white-space characters preceding or following the replacement list of preprocessing tokens is not considered part of the replacement list for either form of macro.

If a **#** preprocessing token, followed by a name, occurs lexically at the point at which a preprocessing directive could begin, the name is not subject to macro replacement.

A preprocessing directive of the form

**# define** *identifier replacement-list new-line*

defines a function-like macro that causes each subsequent instance of the macro name to be replaced by the replacement list of preprocessing tokens that constitute the remainder of the directive.

A preprocessing directive of the form

**# define** *identifier lparen identifier-list opt ) replacement-list new-line*

defines a function-like macro with arguments, similar syntactically to a function call. The parameters are specified by the optional list of identifiers, whose scope extends from their declaration in the identifier list until the new-line character that terminates the **#define** preprocessing directive. Each subsequent instance of the function-like macro name followed by a “(” as the next preprocessing token introduces the sequence of preprocessing tokens that is replaced by the replacement list in the definition (an invocation of the macro). The replaced sequence of preprocessing tokens is terminated by the matching “)” preprocessing token, skipping intervening matched pairs of left and right parentheses tokens.

Within the sequence of preprocessing tokens making up an invocation of a function-like macro, new-line is considered a normal white-space character.

The sequence of preprocessing tokens bounded by the outermost matching parentheses forms the list of arguments for the function-like macro. The individual arguments within the list are separated by comma preprocessing tokens, but comma preprocessing tokens bounded by nested parentheses do not separate arguments. If any argument consists of no preprocessing tokens, the behavior is undefined. If there are sequences of preprocessing tokens within the list of arguments that would otherwise act as preprocessing directives, the behavior is undefined.

### C.4.1 Argument substitution

After the arguments for the invocation of a function-like macro have been identified, argument substitution takes place. A parameter in the replacement list, unless preceded by a **#** or **##** preprocessing token or followed by a **##** preprocessing token, is replaced by the corresponding argument after all macros have been expanded. Before being substituted, each argument’s preprocessing tokens are completely macro-replaced as if they formed the rest of the source file; no other preprocessing tokens are available.

## C.4.2 The # operator

### Constraints

Each # preprocessing token in the replacement list for a function-like macro should be followed by a parameter as the next preprocessing token in the replacement list.

### Semantics

If, in the replacement list, a parameter is immediately preceded by a # preprocessing token, both are replaced by a single character string literal preprocessing token that contains the spelling of the preprocessing token sequence for the corresponding argument. Each occurrence of white space between the argument's preprocessing tokens becomes a single space character in the character string literal. White space before the first preprocessing token and after the last preprocessing token comprising the argument is deleted. Otherwise, the original spelling of each preprocessing token in the argument is retained in the character string literal.

This process requires special handling for producing the spelling of string literals and character constants: a \ character is inserted before each " and \ character of a character constant or string literal (including the delimiting " characters). The order of evaluation of # operators is unspecified.

## C.4.3 The ## operator

### Constraints

A ## preprocessing token does not occur at the beginning or at the end of a replacement list for either form of macro definition.

### Semantics

If, in the replacement list, a parameter is immediately preceded or followed by a ## preprocessing token, the parameter is replaced by the corresponding argument's preprocessing token sequence.

For both object-like and function-like macro invocations, before the replacement list is reexamined for more macro names to replace, each instance of a ## preprocessing token in the replacement list (not from an argument) is deleted and the preceding preprocessing token is concatenated with the following preprocessing token. If the result is not a valid preprocessing token, the behavior is undefined. The resulting token is available for further macro replacement. The order of evaluation of ## operators is unspecified.

### C.4.4 Rescanning and further replacement

After all parameters in the replacement list have been substituted, the resulting preprocessing token sequence is rescanned with the rest of the source file's preprocessing tokens for more macro names to replace.

If the name of the macro being replaced is found during this scan of the replacement list (not including the rest of the source file's preprocessing tokens), it is not replaced. Further, if any nested replacements encounter the name of the macro being replaced, it is not replaced. These nonreplaced macro name preprocessing tokens are no longer available for further replacement, even if they are later reexamined in contexts in which that macro name preprocessing token would otherwise be replaced.

The resulting completely macro-replaced preprocessing token sequence is not processed as a preprocessing directive even if it resembles one.

### C.4.5 Scope of macro definitions

A macro definition lasts, independent of block structure, until either a corresponding **#undef** directive is encountered or, the end of the translation unit is reached.

A preprocessing directive of the form

```
# undef identifier new-line
```

deletes the macro definition of the specified identifier. The **#undef** directive is ignored if the specified identifier is not currently defined as a macro name.

#### Examples

The simplest use of this facility is to define a “manifest constant,” as in

```
#define TABSIZE 100
```

```
int table[TABSIZE];
```

The following defines a function-like macro whose value is the maximum of its arguments. Advantages include

working for any compatible types of the arguments, and

generating in-line code without the overhead of function calling.

Disadvantages include

evaluating one argument a second time (including side effects), and  
generating more code than a function if invoked several times.

**#define max(a, b) ((a) > (b) ? (a) : (b))**

The parentheses insure that the arguments and the resulting expression are bound properly.

To illustrate the rules for redefinition and reexamination, the sequence

```
#define      x      3
#define     f(a)    f(x * (a))
#undef      x
#define      x      2
#define      g      f
#define      z      z[0]
#define      h      g(~
#define     m(a)    a(w)
#define      w      0,1
#define     t(a)    a

f(y+1) + f(f(z)) % t(t(g) (0) +t) (1);
g(x+(3,4)-w) | h 5) & m
(f)^m(m) ;
```

results in

```
f(2 * (y+1)) + f(2 * (f(2 * (z[0])))) % f(2 * (0)) + t(1) ;
f(2 * (2+(3,4)-0,1)) | f(2 * (~ 5)) & f(2 * (0,1))^m(0,1) ;
```

To illustrate the rules for creating character string literals and concatenating tokens, the sequence

```
#define str(s)      # s
#define xstr(s)     str(s)
#define debug(s, t) printf(“‘x’” # s “‘= %d, x’” # t “‘= %s’”, \
                        x ## s, x ## t)
#define INCFILE(n)  vers ## n /* from previous #include example */
#define glue(a, b)  a ## b
#define xglue(a, b) glue(a, b)
#define HIGHLOW     “‘hello’”
```

```

#define LOW          LOW “, world”
debug(1, 2) ;
fputs(str(strncmp(“abc \0d”, “abc”, ‘ \4’) /* this goes away */
      == 0) str(: @ \n), s) ;
#include xstr(INCFIL(2) .h)
glue(HIGH, LOW) ;
xglue(HIGH, LOW)

results in

printf(“x” “1” “= %d, x” “2” “= %s”, x1, x2) ;
fputs(“strcmp( \“abc \0d \”, \“abc \”, ‘ \4’) == 0” “: @\n”, s) ;
#include “vers2.h” (after macro replacement, before file access)
“hello”;
“hello” “, world”

```

or, after concatenation of the character string literals,

```

printf(“x1= %d, x2= %s”, x1, x2) ;
fputs(“strcmp(\“abc \0d \”, \“abc \”, ‘ \4’) == 0: @\n”, s) ;
#include “vers2.h” (after macro replacement, before file access)
“hello”;
“hello, world”

```

Space around the # and ## tokens in the macro definition is optional.

And finally, to demonstrate the redefinition rules, the following sequence is valid.

```

#define OBJ_LIKE      (1-1)
#define OBJ_LIKE      /* white space */ (1-1) /* other */
#define FTN_LIKE(a)( a )
#define FTN_LIKE( a ) /* note the white space */ \
                      a /* other items on this line
                      */ )

```

However, the following redefinitions are invalid:

```

#define OBJ_LIKE      (0)          /* different token sequence */
#define OBJ_LIKE      (1 - 1)     /* different white space */
#define FTN_LIKE(b)   ( a )       /* different parameter usage */
#define FTN_LIKE(b)   ( b )       /* different parameter spelling */

```



## C.5 Line Control

### Constraints

The string literal, if present, is a character string literal.

### Semantics

The *line number* of the current source line is one greater than the number of new-line characters read or introduced in translation phase 1 while processing the source file to the current token.

A preprocessing directive of the form

**# line**      *digit-sequence new-line*

causes the compiler to behave as if the following sequence of source lines begins with a source line that has a line number as specified by the digit sequence (interpreted as a decimal integer).

A preprocessing directive of the form

**# line**      *digit-sequence string-literal new-line*

sets the line number similarly and changes the presumed name of the source file to be the characters contained within the character string literal.

A preprocessing directive of the form

**# line**      *pp-tokens new-line*

is permitted if it does not match one of the two previous forms. The preprocessing tokens after **# line** on the directive are processed just as in normal text (each identifier currently defined as a macro name is replaced by its replacement list of preprocessing tokens). After the compiler makes all replacements, the directive that results matches one of the two previous forms and is then processed as appropriate.

## C.6 Error Directive

### Semantics

A **#error** preprocessing directive causes the compiler to produce a diagnostic message that includes the specified sequence of tokens.

## C.7 Pragma Directive

### Semantics

The compiler ignores **#pragma** preprocessing directives.

## C.8 Null Directive

### Semantics

If there is no token between the **#** and the next new-line character, the directive has no effect.

## C.9 Predefined macro names

The compiler defines the following names:

- \_\_LINE\_\_** The line number of the current source line (a decimal constant).
- \_\_FILE\_\_** The presumed name of the source file (a string literal).
- \_\_DATE\_\_** The date of translation of the source file (a character string literal of the form “**mm dd yyyy**”), where the names of the months are the same as those generated by the **asctime** function, and the first character of the **dd** is a space character if the value is less than 10.
- \_\_TIME\_\_** The time of translation of the source file (a character string literal of the form “**hh:mm:ss**”) as in the time generated by the **asctime** function).
- \_\_STDC\_\_** The decimal constant 1.

The values of the predefined macros (except for **\_\_LINE\_\_** and **\_\_FILE\_\_**) remain constant throughout the translation unit.

None of these macro names, nor the identifier **defined**, should be the subject of a **#define** or a **#undef** preprocessing directive. All predefined macro names should begin with a leading underscore followed by an uppercase letter or a second underscore.

This page is left blank intentionally.

# Sequence Points

## Appendix D

---

Sequence points are points in the execution sequence at which all side effects of previous evaluations are complete and no side effects of subsequent evaluations have taken place.

The following is the list of sequence points:

1. The call to a function, after the arguments have been evaluated ("Function calls" in Appendix A, section "Expressions", subsection "Postfix operators").
2. The logical AND operator `&&` ("Logical AND operator" in Appendix A, section "Expressions").
3. The logical OR operator `||` ("Logical OR operator" in Appendix A, section "Expressions").
4. The conditional operator `?` ("Conditional operator" in Appendix A, section "Expressions").
5. The comma operator `,` ("Comma operator" in Appendix A, section "Expressions").
6. Completion of the evaluation of a full expression:

an initializer of an object that has automatic storage duration ("Initialization" in Appendix A, section "Declarations");

the **expression** statement ("Expression and null statements" in Appendix A, section "Statements");

the controlling expression of a selection statement such as **if** or **switch** ("Selection statements" in Appendix A, section "Statements"); and

the controlling expression of an iteration statement such as **while**, **do**, or **for** (“Iteration statements” in Appendix A, section "Statements").

## E.1 Common Definitions <stddef.h>

ptrdiff\_t  
size\_t  
wchar\_t  
NULL  
offsetof (*type*, *identifier*)

## E.2 Diagnostics <assert.h>

NDEBUG  
void assert(int expression);

## E.3 Character Handling <ctype.h>

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);  
int tolower(int c);  
int toupper(int c);

## E.4 Limits <float.h> and <limits.h>

The contents of a header <limits.h> are given below, in alphabetical order. The values will all be constant expressions suitable for use in **#if** preprocessing directives.

CHAR_BIT	8
CHAR_MAX	127
CHAR_MIN	0 or SCHAR_MIN
INT_MAX	32767
INT_MIN	-32767
LONG_MAX	2147483647
LONG_MIN	-2147483647
SCHAR_MAX	127
SCHAR_MIN	-127
SHRT_MAX	32767
SHRT_MIN	-32767
UCHAR_MAX	0xFF
UINT_MAX	0xFFFF
ULONG_MAX	0xFFFFFFFF
USHRT_MAX	0xFFFF

The contents of a header <float.h> are given below, in alphabetical order. The value of **FLT\_RADIX** will be a constant expression suitable for use in **#if** preprocessing directives. Values that don't need to be constant expressions will be supplied for all other components.

DBL_DIG	15
DBL_EPSILON	1E-9
DBL_MANT-DIG	52
DBL_MAX	1E+307
DBL_MAX_10_EXP	+307
DBL_MAX_EXP	
DBL_MIN	1E-307
DBL_MIN_10_EXP	-308
DBL_MIN_EXP	
FLT_DIG	6
FLT_EPSILON	1E-5
FLT_MANT_DIG	23
FLT_MAX	3.4E+38
FLT_MAX_10_EXP	+38
FLT_MAX_EXP	
FLT_MIN	1.2E-38
FLT_MIN_10_EXP	-38

FLT_MIN_EXP	
FLT_RADIX	2
FLT_ROUNDS	0
LDBL_DIG	15
LDBL_EPSILON	1E-9
LDBL_MANT_DIG	52
LDBL_MAX	1E+307
LDBL_MAX_10_EXP	+307
LDBL_MAX_EXP	
LDBL_MIN	1E-307
LDBL_MIN_10_EXP	-308
LDBL_MIN_EXP	

## E.5 Mathematics <math.h>

EDOM  
ERANGE  
HUGE\_VAL  
double acos(double x);  
double asin(double x);  
double atan(double x);  
double atan2 (double y, double x);  
double ceil(double x);  
double cos(double x);  
double cosh(double x);  
double exp(double x);  
double fabs(double x);  
double floor(double x);  
double fmod(double x, double y);  
double frexp(double value, int \*exp);  
double ldexp(double x, int exp);  
double log(double x);  
double log10(double x);  
double modf(double value, double \*iptr);  
double pow(double x, double y);  
double sin(double x);  
double sinh(double x);  
double sqrt(double x);  
double tan(double x);  
double tanh(double x);



## E.6 Non-Local Jumps <setjmp.h>

```
jmp_buf  
int setjmp(jmp_buf env);  
void longjmp(jmp_buf env, int val);
```

## E.7 Variable Arguments <stdarg.h>

```
va_list  
void va_start(va_list ap, parmN);  
type va_arg(va_list ap, type);  
void va_end(va_list ap);
```

## E.8 Input/Output <stdio.h>

```
size_t  
int printf(const char *format, ...);  
int scanf(const char *format, ...);  
int sprintf(char *s, const char *format, ...);  
int sscanf(const char *s, const char *format, ...);  
int vprintf(const char *format, va_list arg);  
int vsprintf(char *s, const char *format, va_list arg);  
int getchar(void);  
char *gets(char *s);  
int putchar(int c);  
int puts(const char *s);
```

## E.9 NonStandard IO <nonstdio.h>

```
int getch();  
int getche(void);  
int kbhit();  
void init_uart(void);  
void putch(char);
```

## E.10 General Utilities <stdlib.h>

```

MB_CUR_MAX
NULL
RAND_MAX
div_t
ldiv_t
size_t
wchar_t
double atof(const char *nptr);
int atoi(const char *nptr);
long int atol(const char *nptr);
double strtod(const char *nptr, char **endptr);
long int strtol(const char *nptr, char **endptr, int base);
int rand(void);
void srand(unsigned int seed);
void *calloc(size_t nmemb, size_t size);
void free(void *ptr);
void *malloc(size_t size);
void *realloc(void *ptr, size_t size);
void abort(void); void exit(int status);
char *getenv(const char *name);
void *bsearch(const void *key, const void *base,
    size_t nmemb, size_t size, int (*compar)(const void *, const void *));
void qsort(void *base, size_t nmemb, size_t size, int (*compar)
    (const void *, const void *));
int abs(int j);
div_t div(int numer, int denom);
long int labs(long int j);
ldiv_t ldiv(long int numer, long int denom);

```

## E.11 Character Handling<string.h>

```

NULL
size_t
void *memcpy(void *s1, const void *s2, size_t n);
void *memmove(void *s1, const void *s2, size_t n);
char *strcpy(char *s1, const char *s2);
char *strncpy(char *s1, const char *s2, size_t n);
char *strcat(char *s1, const char *s2);
char *strncat(char *s1, const char *s2, size_t n);
int memcmp(const void *s1, const void *s2, size_t n);
int strcmp(const char *s1, const char *s2);
int strncmp(const char *s1, const char *s2, size_t n);

```

```
void *memchr(const void *s, int c, size_t n);
char *strchr(const char *s, int c);
size_t strcspn(const char *s1, const char *s2);
char *strpbrk(const char *s1, const char *s2);
char *strchr(const char *s, int c);
size_t strspn(const char *s1, const char *s2);
char *strstr(const char *s1, const char *s2) char *strtok (char *s1,
    const char *s2);
void *memset(void *s, int c, size_t n);
char *strerror(int errnum); size_t strlen(const char *s);
```

# Error Messages

## Appendix | | |---| | F | |---|

---

### F.1 Preprocessor Error Messages

000    “Illegal constant expression in directive”

A constant expression made up of constants and macros that evaluate to constants may be the only operands of an expression used in a preprocessor directive.

001    “Concatenation at end-of-file. Ignored”

An attempt was made to concatenate lines with a backslash (when the line is the last line of the file).

002    “Illegal token. Ignored”

An unrecognizable token or non-ASCII character was encountered.

003    “Illegal macro re-definition <name>”

An attempt was made to redefine a macro and the tokens in the macro definition do not match those of the previous definition.

004    “Incorrect number of arguments for macro <name>”

An attempt was made to call a macro, but too few or too many arguments were given.

005    “Unbalanced parentheses in macro call”

An attempt was made to call a macro with a parenthesis embedded in the argument list that did not match up.

006 “Cannot redefine <name> keyword”

An attempt was made to redefine a keyword as a macro.

007 “Illegal directive syntax”

The syntax of a preprocessor directive is incorrect.

008 “Illegal “#if” directive syntax”

The syntax of a #if preprocessor directive is incorrect.

009 “Bad preprocessor file. Aborted”

An unrecognizable source file was given to the compiler.

010 “Illegal macro call syntax”

An attempt was made to call a macro that does not conform to the syntax rules of the language.

011 “Integer constant too large”

An integer constant that has a binary value too large to be stored in 32 bits was encountered.

013 “Illegal #include argument”

The argument to a #include directive must be of the form “pathname” or <filename>.

014 “Macro “<name>” requires arguments”

An attempt was made to call a macro defined to have arguments and was given none.

017 “Unterminated quoted string”

Within a quoted string an end of line was encountered.

018    “Escape sequence ASCII code too large to fit in char”

The binary value of an escape sequence requires more than 8 bits of storage.

019    “Character not within radix”

An integer constant was encountered with a character greater than the radix of the constant.

020    “More than one character in string constant”

A string constant was encountered having more than one ASCII character.

027    “End-of-file found before #endif directive”

Within a #if or #else block end of file was encountered.

## F.2 Front-End Error Messages

100    “Syntax error”

A syntactically incorrect statement, declaration or expression was encountered.

101    “Function “<name>” already declared”

An attempt was made to define two functions with the same name.

102    “Constant integer expression expected”

A non integral expression was encountered where only an integral expression may be.

103    “Constant expression overflow”

In the process of evaluating a constant expression, value became too large to be stored in 32 bits.

## 104 “Function return type mismatch for “&lt;name&gt;”

A function prototype or function declaration was encountered that has a different result from a previous declaration.

## 105 “Argument type mismatch for argument &lt;name&gt;”

The type of an actual parameter does not match the type of the formal parameter of the function called.

## 106 “Cannot take address of un-subscripted array”

An attempt was made to take the address of an array with no index. The address of the array is already implicitly calculated.

## 107 “Function call argument cannot be void type”

An attempt was made to pass an argument to a function that has type **void**.

## 108 “Identifier “&lt;name&gt;” is not a variable or enumeration constant name”

In a declaration, a reference to an identifier was made that was not a variable name or an enumeration constant name.

## 109 “Cannot return a value from a function returning “void”

An attempt was made to use a function defined as returning void in an expression.

## 110 “Expression must be arithmetic, structure, union or pointer type”

The type of an operand to a conditional expression was not arithmetic, structure, union or pointer type.

## 112 “Expression not compatible with function return type”

An attempt was made to return a value from function that could not be promoted to the type defined by the function declaration.

## 113 “Function cannot return value of type array or function”

An attempt was made to return a value of type array or function.

## 114 “Structure or union member may not be of function type”

An attempt was made to define a member of structure or union that has type function.

## 115 “Cannot declare a typedef within a structure or union”

An attempt was made to declare a typedef within a structure or union.

## 116 “Illegal bit field declaration”

An attempt was made to declare a structure or union member that is a bit field and is syntactically incorrect.

## 121 “Illegal declaration specifier”

An attempt was made to declare an object with an illegal declaration specifier.

## 122 “Only “const” and “volatile” may be specified with a struct, union, enum, or typedef”

An attempt was made to declare a struct, union, enum, or typedef with a declaration specifier other than const and volatile.

## 123 “Cannot specify both long and short in declaration specifier”

An attempt was made to specify both long and short in the declaration of an object.

## 124 “Only “const” and “volatile” may be specified within pointer declarations”

An attempt was made to declare a pointer with a declaration specifier other than const and volatile.



125    “Identifier “<name>” already declared within current scope”

An attempt was made to declare two objects of the same name in the same scope.

126    “Identifier “<name>” not in function argument list, ignored”

An attempt was made to declare an argument that is not in the list of arguments when using the old style argument declaration syntax.

127    “Name of formal parameter not given”

The type of a formal parameter was given in the new style of argument declarations without giving an identifier name.

128    “Identifier “<name>” not defined within current scope”

An identifier was encountered that is not defined within the current scope.

129    “Cannot have more than one default per switch statement”

More than one default statements were found in a single switch statement.

130    “Label “<name>” is already declared”

An attempt was made to define two labels of the same name in the same scope.

131    “Label “<name>” not declared”

A goto statement was encountered with an undefined label.

132    “ “continue” statement not within loop body”

A continued statement was found outside a body of any loop.

133    “ “break” statement not within switch body or loop body”

A break statement was found outside the body of any loop.

134    “ ‘‘case’’ statement must be within switch body’’

A case statement was found outside the body of any switch statement.

135    “ ‘‘default’’ statement must be within switch body’’

A default statement was found outside the body of any switch statement.

136    “‘Case value ‘‘<name>’’ already declared’’

An attempt was made to declare two cases with the same value.

137    “‘Expression is not a pointer’’

An attempt was made to dereference value of an expression whose type is not pointer.

138    “‘Expression is not a function locator’’

An attempt was made to use an expression as the address of a function call that does not have a type pointer to function.

139    “‘Expression to left of ‘‘.’’ or ‘‘->’’ is not a structure or union’’

An attempt was made to use an expression as a structure or union, or a pointer to a structure or union; whose type was neither a structure or union, or a pointer to a structure or union.

140    “‘Identifier ‘‘<name>’’ is not a member of <name> structure’’

An attempt was made to reference a member of a structure that does not belong to the structure.

141    “‘Object cannot be subscripted’’

An attempt was made to use an expression as the address of an array or a pointer that was not an array or pointer.

142    “Array subscript must be of integral type”

An attempt was made to subscript an array with a non integral expression.

143    “Cannot dereference a pointer to “void” ”

An attempt was made to dereference a pointer to void.

144    “Cannot compare a pointer to a non-pointer”

An attempt was made to compare a pointer to a non-pointer.

145    “Pointers to different types may not be compared”

An attempt was made to compare pointers to different types.

146    “Pointers may not be added”

It is not legal to add two pointers.

147    “A Pointer and a non-integral may not be subtracted”

It is not legal to subtract a non-integral expression from a pointer.

148    “Pointers to different types may not be subtracted”

It is not legal to subtract two pointers of different types.

149    “Unexpected end of file encountered”

In the process of parsing the input file, end of file was reached during the evaluation of an expression, statement or declaration.

150    “Unrecoverable error encountered”

The compiler became confused beyond the point of recovery.

## 151 “Operand must be a modifiable lvalue”

An attempt was made to assign a value to an expression that was not modifiable.

## 152 “Operands are not assignment compatible”

An attempt was made to assign a value whose type could not be promoted to the type of the destination.

## 153 “&lt;name&gt; must be arithmetic type”

An expression was encountered whose type was not arithmetic where only arithmetic types are allowed.

## 154 “&lt;name&gt; must be integral type”

An expression was encountered whose type was not integral where only integral types are allowed.

## 155 “&lt;name&gt; must be arithmetic or pointer type”

An expression was encountered whose type was not pointer or arithmetic where only pointer and arithmetic types are allowed.

## 156 “Expression must be an lvalue”

An expression was encountered that is not an lvalue where only an lvalue is allowed.

## 157 “Cannot assign to an object of constant type”

An attempt was made to assign a value to an object defined as having constant type.

## 158 “Cannot subtract a pointer from an arithmetic expression”

An attempt was made to subtract a pointer from an arithmetic expression.

159    “An array is not a legal lvalue”

Cannot assign an array to an array.

160    “Cannot take address of a bit field”

An attempt was made to take the address of a bit field.

161    “Cannot take address of variable with “register” class”

An attempt was made to take the address of a variable with “register” class.

162    “Conditional expression operands are not compatible”

One operand of a conditional expression cannot be promoted to the type of the other operand.

163    “Casting a non-pointer to a pointer”

An attempt was made to promote a non-pointer to a pointer.

164    “Type name of cast must be scalar type”

An attempt was made to cast an expression to a non-scalar type.

165    “Operand to cast must be scalar type”

An attempt was made to cast an expression whose type was not scalar.

166    “Expression is not a structure or union”

An expression was encountered whose type was not structure or union where only a structure or union is allowed.

167    “Expression is not a pointer to a structure or union”

An attempt was made to dereference a pointer with the arrow operator, and the expression's type was not pointer to a structure or union.

168 “Cannot take size of void, function, or bit field types”

An attempt was made to take the size of an expression whose type is void, function, or bit field.

169 “Actual parameter has no corresponding formal parameter”

An attempt was made to call a function whose formal parameter list has fewer elements than the number of arguments in the call.

170 “Formal parameter has no corresponding actual parameter”

An attempt was made to call a function whose formal parameter list has more elements than the number of arguments in the call.

171 “Argument type is not compatible with formal parameter”

An attempt was made to call a function with an argument whose type is not compatible with the type of the corresponding formal parameter.

172 “Identifier “<name>” is not a structure or union tag”

An attempt was made to use the dot operator on an expression whose type was not structure or union.

173 “Identifier “<name>” is not a structure tag”

The tag of a declaration of a structure object does not have type structure.

174 “Identifier “<name>” is not a union tag”

The tag of a declaration of a union object does not have type union.

175 “Structure or union tag “<name>” is not defined”

The tag of a declaration of a structure or union object is not defined.

176 “Only one storage class may be given in a declaration”

An attempt was made to give more than one storage class in a declaration.

177 “Type specifier cannot have both ‘unsigned’ and ‘signed’ ”

An attempt was made to give both unsigned and signed type specifiers in a declaration.

178 “ ‘unsigned’ and ‘signed’ may be used in conjunction only with ‘int’, ‘long’ or ‘char’ ”

An attempt was made to use signed or unsigned in conjunction with a type specifier other than int, long, or char.

179 “ ‘long’ may be used in conjunction only with ‘int’ or ‘double’ ”

An attempt was made to use long in conjunction with a type specifier other than int or double.

180 “Illegal bit field length”

The length of a bit field was outside of the range 0-32.

181 “Too many initializers for object”

An attempt was made to initialize an object with more elements than the object contains.

182 “Static objects can be initialized with constant expressions only”

An attempt was made to initialize a static object with a non-constant expression.

183 “Array ‘<name>’ has too many initializers”

An attempt was made to initialize an array with more elements than the array contains.

184 “Structure ‘<name>’ has too many initializers”

An attempt was made to initialize a structure with more elements than the structure has members.

## 185 “Dimension size may not be omitted”

An attempt was made to omit the dimension of an array which is not the rightmost dimension.

## 186 “First dimension of ‘&lt;name&gt;’ may not be omitted”

An attempt was made to omit the first dimension of an array which is not external and is not initialized.

## 187 “Dimension size must be greater than zero”

An attempt was made to declare an array with a dimension size of zero.

## 188 “Only ‘register’ storage class is allowed for formal parameter”

An attempt was made to declare a formal parameter with storage class other than register.

## 189 “Cannot take size of array with missing dimension size”

An attempt was made to take the size of an array with an omitted dimension.

## 190 “Identifier ‘&lt;name&gt;’ already declared with different type or linkage”

An attempt was made to declare a tentative declaration with a different type than a declaration of the same name; or, an attempt was made to declare an object with a different type from a previous tentative declaration.

## 191 “Cannot perform pointer arithmetic on pointer to void”

An attempt was made to perform pointer arithmetic on pointer to void.



## F.3 Code Generator Error/Warning Messages

303 “Case value <number> already defined”

If a case value consists of an expression containing a **sizeof**, its value is not known until code generation time. Thus, it is possible to have two cases with the same value not caught by the front end. Review the **switch** statement closely.

308 “Excessive Registers required at line <num> of function <func>”

Excessive Page 0 registers are required at line number <num>. The compiler does not perform register spilling, so complex expressions that generate this error must be factored into two or more expressions.

309 “Interrupt function <name> cannot have arguments”

A function declared as an interrupt function cannot have function arguments.

310 “Index out of range, truncating index of <num> to 255”

The 51CC compiler detected an array access outside of the 0..255 byte range. Use a temporary variable if you need to access an array element outside of this range.

311 “<feature> is not supported by the C19 compiler”

The feature is not supported by the compiler. Unsupported features include floating point values, 32-bit multiplication and division, and functions returning a structure.

312 “Aggregate Copy out of range, truncating copy size of <num>”  
to 255”

The compiler limits structure sizes to 255 bytes. An attempt was made to copy structures greater than 255 bytes. Use the **memcpy** library function if structures greater than 255 bytes are required.

## 313 “Bitfield Length exceeds 8 bits”

The compiler only accepts bitfield lengths of 8 bits or less.

This page is left blank intentionally.

# Index

---

## A

abs 5-51  
acos 5-52  
address A-196  
aggregate types A-176  
alignment 3-21  
argument C-269  
arithmetic conversions A-185  
arithmetic types A-176, A-185  
array type A-186  
arrays A-175  
asctime C-275  
asin 5-53  
asm statement 3-19  
assembly language 6-156  
assert 4-30, 5-54  
assignment operators A-206, A-209  
atan 5-55  
atan2 5-56  
atof 4-39, 5-57  
atoi 4-27, 4-39, 5-58  
atol 4-39, 5-59  
automatic storage duration A-174

## B

basic types A-174  
bsearch 5-60  
binary operator A-189  
bit-field A-216  
block A-170, A-230 - A-231

block scope A-171  
break statement A-236

## C

call frame layouts 6-155  
call frames 3-17  
calloc 4-39, 5-61  
cast A-198  
ceil 5-62  
char A-174, A-184, A-214  
CHAR\_BIT 4-31  
CHAR\_MAX 4-31  
CHAR\_MIN 4-31  
character A-174  
character constant A-165, A-180  
character string literal A-182  
comma operator A-209  
comma operators A-189  
comments A-184  
common initial sequence A-194  
compound assignment A-208  
compound statement A-231  
const A-187, A-214, A-220  
const pointers 3-18  
const-qualified type A-186  
constants A-167, A-176, A-190  
continue statement A-234 - A-235  
control character 4-30  
conversions A-184 - A-185  
cos 5-63  
cosh 5-64

**D**

data alignment 3-21  
DATE C-275  
DBL\_DIG 4-32  
DBL\_MANT\_DIG 4-32  
DBL\_MAX 4-32  
DBL\_MAX\_10\_EXP 4-32  
DBL\_MAX\_EXP 4-32  
DBL\_MIN 4-32  
DBL\_MIN\_10\_EXP 4-32  
DBL\_MIN\_EXP 4-32  
declarations A-210, A-231  
declarator A-220 - A-221  
decrement operator A-209  
default argument conversions A-192  
#define C-265, C-268 - C-269, C-275  
definition A-212  
derived types A-175  
DI 5-65  
distance A-226  
div 3-19, 4-38, 5-66  
do statement A-233  
domain error 4-35  
double A-214  
dynamic call frame 3-20  
dynamic call frames 3-15, 6-156

**E**

EI 5-67  
8051.h 4-43  
#else C-266  
#endif C-266  
enumerated types A-175  
enumeration A-170  
EOF 4-30  
errno 4-29  
#error A-184, C-274  
escape sequences A-181  
exception A-189  
exp5-68  
expression A-189 - A-190

expression statement A-229, A-231  
extern A-171, A-173, A-238  
external definition A-237  
external linkage A-171, A-192

**F**

fabs 5-69  
far pointers 3-18  
.FCALL directive 6-155  
file A-170, C-275  
file scope A-171  
floating types 4-32, A-176  
floor 5-70  
FLT\_DIG 4-32  
FLT\_MANT\_DIG 4-33  
FLT\_MAX 4-33  
FLT\_MAX\_10\_EXP 4-33  
FLT\_MAX\_EXP 4-33  
FLT\_MIN 4-33  
FLT\_MIN\_EXP 4-33  
FLT\_RADIX 4-32 - 4-33  
FLT\_ROUND 4-33  
fmod 5-71  
for statement A-234  
.FRAME directive 6-155  
free 5-72  
frexp 5-73  
full expression A-229  
function call A-231  
function locator A-187  
function prototype A-170  
function prototype scope A-171  
function scope A-170  
function-call operator A-189, A-209  
functions A-170, A-175

**G**

getch 5-74  
getchar 5-76  
getc 4-43, 5-75

gets 5-77  
 gmtime 3-19  
 goto statement A-230, A-234

## H

header 4-26  
 hexadecimal digits A-181  
 hexadecimal integer A-181  
 HUGE\_VAL 4-34

## I

identifiers A-167, A-168, A-170, A-190  
 if statement A-232  
 implicit conversion A-184  
 #include 2-8, 4-26, A-184  
 incomplete type A-186  
 init\_uart 4-43  
 initialization A-226  
 int A-174, A-184, A-214  
 INT\_MAX 4-31, A-175  
 INT\_MIN 4-31, A-175  
 integer constant A-176, A-178 - A-179  
 integral constant expression A-209  
 integral promotions A-184  
 integral types 4-31, A-176  
 internal linkage A-172  
 interrupt 3-16, 5-120  
 interrupt vectors 5-120  
 isalnum 5-78  
 isalpha 5-79  
 iscntrl 5-80  
 isdigit 5-81  
 isgraph 5-82  
 islower 5-83  
 isprint 5-84  
 ispunct 5-85  
 isspace 5-86  
 isupper 5-87  
 isxdigit 5-88

iteration statement  
 A-230 - A-231, A-233, A-235 - A-236

## J

jump statement A-234

## K

kbhit 5-89  
 keywords A-167, A-168, A-214

## L

label names A-170, A-173  
 labs 5-90  
 large model  
   See memory models  
 LDBL\_DIG 4-33  
 LDBL\_MANT\_DIG 4-33  
 LDBL\_MAX 4-33  
 LDBL\_MAX\_10\_EXP 4-34  
 LDBL\_MAX\_EXP 4-33  
 LDBL\_MIN 4-34  
 LDBL\_MIN\_10\_EXP 4-34  
 LDBL\_MIN\_EXP 4-34  
 ldexp 5-91  
 ldiv 3-19, 4-38, 5-92  
 ldiv\_t 4-38  
 #line A-184, C-275  
 line number C-274  
 linkage A-171  
 localtime 3-19  
 log 5-93  
 log10 5-94  
 long A-214  
 long int A-174  
 LONG\_MAX 4-31  
 LONG\_MIN 4-31  
 longjmp 5-95

loop body A-233  
lvalue A-186

## M

macro C-268, C-275  
macro name A-170, C-268  
malloc 4-39, 5-96  
members A-173  
memchr 5-97  
memcmp 5-98  
memcpy 5-99  
memmove 5-100  
memory models  
    models 3-15  
memory models  
    large model 3-16  
memset 5-101  
modf 5-102

## N

name spaces A-173  
NDEBUG 4-30  
near 3-17  
near pointers 3-18  
NULL 4-29  
null character A-165, A-182  
null pointer A-187  
null pointer constant A-187  
null statement A-231, A-235

## O

object A-170  
octal digits A-181  
# operator C-270  
operators A-167, A-183  
ordinary identifiers A-173

## P

### PCG

See program call graph

pointers A-175  
pow 5-103  
#pragma A-184, C-275  
    #pragma asm 3-19  
Pragmas 2-9  
    #pragma alias 2-9  
    #pragma fpcp 2-9  
    #pragma noalias 2-9  
    #pragma nobss 2-9  
    #pragma nofpcp 2-9  
    #pragma noopt 2-9  
    #pragma nooptlink 2-9  
    #pragma nosdipt 2-9  
    #pragma nostkck 2-9  
    #pragma nostrict 2-9  
    #pragma optlink 2-9  
    #pragma optsize 2-9  
    #pragma optspeed 2-9  
    #pragma sdiopt 2-9  
    #pragma stkck 2-10  
    #pragma strict 2-10  
preprocessing C-265  
primary expression A-189  
printf 5-104  
printing character 4-30  
program call graph 3-15, 6-155  
ptrdiff\_t 4-29  
punctuators A-167, A-183  
putch 4-43, 5-108  
putchar 5-109  
puts 5-110

## Q

qsort 5-111

**R**

rand	3-19, 4-39, 5-112	STDC	C-275
range error	4-35	storage duration	A-173
realloc	4-39, 5-113	storage-class	A-212
reentrancy	3-19	strcat	5-128
Requirements	1-1	strchr	5-129
rescanning	C-271	strcmp	5-130
return statement	A-236	strcpy	5-131
Runtime Library Funnel	3-21	strcspn	5-132
		string literal	A-190
		string literals	A-167, A-182, C-272, C-274
		strlen	5-133
		strncat	5-134
		strncmp	5-135
		strncpy	5-136
		strpbrk	5-137
		strrchr	5-138
		strspn	5-139
		strstr	5-140
		strtod	5-141
		strtok	3-19, 5-142
		strtol	5-142
		structures	A-170, A-175
		switch body	A-233
		switch statement	A-230, A-232

**S**

scalar types	A-176	SWITCHES	
scanf	5-114	-ALIAS	2-3
SCHAR_MAX	4-31	-ASM	2-3
SCHAR_MIN	4-31	-ASMSW	2-3
scope	A-170, C-271	-DEBUG	2-4
segments	6-162	-DEFALL	2-4
selection statements	A-229, A-232	-DEFINE	2-4
SET_VECTOR	5-118 - 5-121	-EMBEDDED	2-4
setjmp	5-118	-EXPMAC	2-4
short	A-214	-FPLIB	2-4
short int	A-174, A-184	-HEXFILE	2-4
SHRT_MAX	4-31	-INTSRC	2-4
SHRT_MIN	4-31	-KEEPASM	2-3, 2-5
signed	A-214	-KEEPLNK	2-5
signed char	A-174	-KEEPLST	2-5
signed integer types	A-174	-LINK	2-5
simple assignment	A-207	-LINKSW	2-5
sin	5-122	-LIST	2-5
sinh	5-123	-LISTINC	2-5
sizeof	A-197		
small model	3-17		
sprintf	5-124		
sqrt	5-125		
srand	3-19, 5-126		
sscanf	5-127		
statement	A-229, A-232 - A-233		
static	6-162, A-171		
static call frame	3-20		
static call frames	3-15, 6-155		
static storage duration	A-174		



-MAP	2-5	token	A-167
-MAXERRS	2-5, 2-10	tolower	5-146
-MODEL	2-5	toupper	5-147
-NOALIAS	2-5	trigraph sequences	A-166
-NOASM	2-6	type equivalence	A-225
-NOBSS	2-6	type name	5-148, A-224
-NODEFALL	2-6	typedef	A-212, A-214, A-225
-NOEXPMAC	2-6	typedef name	A-170
-NOFPCP	2-6	types	A-174
-NOFPLIB	2-6		
-NOINTSRC	2-6		
-NOKEEPASM	2-6	<b>U</b>	
-NOLINK	2-6	UCHAR_MAX	4-31
-NOLIST	2-6	UINT_MAX	4-31, A-197, A-202
-NOLISTINC	2-6	ULONG_MAX	4-32, A-197, A-202
-NOMAP	2-6	unary operator	A-189
-NONINTRINSICS:	2-7	unary plus operators	A-189
-NOOPT	2-7	#undef	4-27, 4-37, C-265, C-271, C-275
-NOOPTLINK	2-7	unions	A-170, A-175
-NOSDIOPT	2-7	unsigned	A-175, A-214
-NOSTKCK	2-7	unsigned integer type	A-175
-NOSTRICT	2-7	USHRT_MAX	4-32
-OF	2-7		
-OPTLINK	2-7	<b>V</b>	
-OPTSIZE	2-7		
-OPTSPEED	2-7	va_arg	5-148
-ROM	2-4	va_end	5-149
-SDIOPT	2-7	va_start	4-37, 5-151
-STARTUP	2-7	variable arguments	3-20
-STDINC	2-8	void	A-175, A-187
-STKCK	2-8	volatile	A-187 - A-188, A-214, A-220
-STRICT	2-8	vprintf	5-152
-STRTABSZ	2-8	vsprintf	5-153
-USRINC	2-8		
-WATCH	2-9		

**T**

tags	A-170, A-173, A-216
tan	5-144
tanh	5-145
tentative definition	A-240
TIME	C-275

**W**

while statement	A-233
white space	A-167
wide character	A-182
wide string literal	A-182