A

abort() — General utility (libc)

End program immediately
void abort(void)

abort terminates a program's execution immediately. It is used to "bail out" of a program when a severe, unrecoverable problem occurs. It does not return.

abort terminates the program by calling **exit** with status **EXIT_FAILURE**.

abort prints the relative address from the beginning of the program, so that you can look the location up in the symbol table. See the entry for **nm** for more information on how to extract the symbol table from an executable program.

Example

This example simply aborts itself. For an example that uses **abort** in a more realistic manner, see **signal**.

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

main(void)
{
      puts("...Dave ... I can feel my memory going ...");
      abort();
}
```

Cross-references

Standard, §4.10.4.1
The C Programming Language, ed. 2, p. 252

See Also

atexit, exit, general utilities, getenv, program termination, system

Notes

Some implementations of **abort**, specifically the one included with UNIX system V, permit it to return. The Standard forbids **abort** to return.

abs() — General utility (libc)

Compute the absolute value of an integer
#include <stdlib.h>
int abs(int *n***);**

abs returns the absolute value of integer *n*. The *absolute value* of a number is its distance from zero. This is *n* if *n***>=0**, and -*n* otherwise.

Example

This example checks whether **abs** is defined for all values on your implementation.

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

```
main(void)
{
	if(INT_MAX != abs(INT_MIN))
		printf("abs of %d is undefined\n", INT_MIN);
	return(EXIT_SUCCESS);
}
```

Cross-reference

Standard, §4.10.6.1
The C Programming Language, ed. 2, p. 253

See Also

div, general utilities, labs, ldiv

Notes

On two's complement machines, the absolute value of the most negative number may not be representable.

abs was originally declared in the header **math.h**. The Standard moved this function to **stdlib.h** on the grounds that it does not return **double**. This change may require that some existing code be altered.

access() — Access checking (libc)

Check if a file can be accessed in a given mode
#include <access.h>
int access(char ****filename*, int** *mode***);**

access checks whether a file can be accessed in the mode you wish. *filename* is the full path name of the file you wish to check. *mode* is the mode in which you wish to access *filename*, as follows:

1	AEXEC	Execute the file
2	AWRITE	Write into the file
4	AREAD	Read the file

The header **access.h** defines the manifest constants that are commonly used with **access**.

access returns zero if *filename* can be accessed in the requested mode, and a number greater than zero if it cannot.

Example

The following example checks if a file can be accessed in a particular manner.

```
#include <access.h>
#include <path.h>
#include <stdio.h>
#include <stdlib.h>

void
fatal(char *message)
{
	sprintf(stderr, "%s\n", message);
	exit(EXIT_FAILURE);
}
```

```
main(int argc, char *argv[])
{
      char *env, *pathname;
      extern char *getenv(), *path();
      int mode;
      extern int access();

      if (argc != 3)
            fatal("Usage: access filename mode");

      switch(*argv[2]) {
            case 'e':
            case 'E':
                  mode = AEXEC;
                  break;
            case 'w':
            case 'W':
                  mode = AWRITE;
                  break;
            case 'r':
            case 'R':
                  mode = AREAD;
                  break;
            default:
                  fatal("modes: e=execute, w=write, r=read");
      }

      env = getenv("PATH");
      if ((pathname = path(env,argv[1],mode)) != NULL) {
            printf("PATH = %s\n", env);
            printf("pathname = %s\n", pathname);

            if (access(pathname, mode) == 0)
                  printf("%s accessible in mode %s\n",
                        pathname, argv[2]);
            else
                  printf("%s not accessible in mode %d\n",
                         pathname, mode);
      } else {
            printf("file %s of mode %d not found in path\n",
                   argv[1], mode);
            exit(EXIT_FAILURE);
      }
      return EXIT_SUCCESS;
}
```

See Also

access checking, access.h, path

Notes

access is included mainly for compatibility with the UNIX operating system. The only meaningful test that **access** can perform on the Atari ST is to check if a file is writable.

access.h — Header

Define manifest constants used by access()
#include <access.h>

access.h is a header file that defines the manifest constants used with the function **access**.

See Also

access, access checking, header

access checking — Overview

Let's C includes the following routines to check the access to a given file:

access.h
  access Check if a file can be accessed in a given mode

path.h
  path Build a path name for a file

stat.h
  stat Find file attributes

These routines are not described in the ANSI Standard. Any program that uses any of them does not conform strictly to the Standard, and may not be portable to other compilers or environments.

See Also

Library

acos() — Mathematics (libm)

Calculate inverse cosine
#include <math.h>
double acos(double *arg***);**

acos calculates the inverse cosine of *arg*, which should be in the range of from -1.0 to 1.0. Any other argument will trigger a domain error.

acos returns the result, which is in the range of from zero to π radians.

Cross-references

Standard, §4.5.2.1
The C Programming Language, ed. 2, p. 251

See Also

asin, atan, atan2, cos, mathematics, sin, tan

address — Definition

An *address* designates a location in memory.

Example

The following prints the address and contents of a given byte of memory.

```
#include <stdio.h>
#include <stdlib.h>
main(void)
{
      char byte = 'a';
      /* Note use of the '%p' format specifier */
      printf("Address==%p  Contents==\"%c\"\n",
            &byte, byte);
      return EXIT_SUCCESS;
}
```

Cross-reference

The C Programming Language, ed. 2, p. 94

See Also

&, Definitions, pointer

alias — Definitions

An *alias* for an object is alternative way to access that object.

Because C uses pointers, it can be impossible for the translator to keep track of all possible aliases for an object. Often, the translator must use "worst-case aliasing assumptions" when memory is read. These assumption are explained below.

The Standard refers to aliasing in the section on expressions (3.3). It allows the translator to assume that the only way to reference a given object is by an object of the same type, a pointer to an object of that type, or by a character pointer. Type qualifiers and sign do not count in this situation. The reason a character pointer is assumed to point to any type of object is one of historical practice.

By making use of this information concerning types, a translator is said to make more favorable aliasing assumptions, and produce better code. For example, consider the following code fragment:

```
fn(int *ip, float *fp)
{
	int i;
	float f;

	ip = &i;	/* line 1 */
	*fp = f;	/* line 2 */
}
```

Normally in an assignment to a dereferenced pointer (line 2), the translator must assume that such a statement can overwrite the values of all global variables and the values of all local variables that have had their addresses taken.

Because **fp** is a pointer to **float**, the assignment to ***fp** need not invalidate the value of **i**. The translator must assume only that the current values of other **float**s may have been changed.

Any attempt to trick the translator, such as with a statement of the form

```
*fp = (float) i;
```

generates undefined behavior.

See Also

Definitions, type qualifier

alien — C keyword

Name a non-standard function

The **alien** declaration tells **Let's C** that the following function name is not a standard C function.

With the Mark Williams family of C compilers, **alien** indicates that a function uses the PL/M calling conventions. These differ from C in a number of ways. First, the calling sequence for PL/M pushes the leftmost argument first, whereas the calling sequence for C functions pushes the rightmost argument first. In addition, PL/M arguments are popped by the called function, whereas C arguments are popped by the calling function. Finally, when **Let's C** compiles a C function, it appends an underbar '_' to the end of the function's name.

Use of the **alien** keyword allows direct calls of most PL/M procedures and functions; that is, it can generate PL/M calls as well as C calls. For example,

```
extern alien plmfn();
```

declares **plmfn** to be a function that uses PL/M calling conventions. Of course, the types of the arguments to **plmfn** must correspond to the types of the arguments the PL/M functions expects.

To use the **alien** keyword in a program compiled with **Let's C**, you must compile the program using the **-VALIEN** option to the **cc** command.

See Also

C keywords, Language, statements

alignment — Definition

The term *alignment* refers to the fact that some environments require the addresses of certain data types to be evenly divisible by a certain integer. Different processors have different alignment requirements. For example, the Motorola 68000 requires that every **int** have an address that is even (i.e., that is evenly divisible by two). The translator must ensure that data objects are aligned properly so that fetches to memory will be performed efficiently and on the correct data types.

The environment may require that empty bytes of "padding" be inserted into structures to ensure that every type is aligned properly. For example, on the M68000 the following structure

```
struct example {
      char member1;
      int member2;
};
```

will actually consist of four bytes: one byte to hold the **char**, two bytes to hold the **int**, and between them, one byte of padding to ensure that the **int** is aligned properly. Often, the alignment of a **struct** member will be the maximum alignment required to align any of its members' data types.

Because different environments require different forms of alignment, a program that is intended to be portable should not assume that the members of a structure abut each other.

An object of type **char** * has the least strict alignment.

Cross-references

Standard, §1.6
The C Programming Language, ed. 2, p. 185

See Also

char, Definitions, struct

arena — Definition

An **arena** is the area of memory that is available for a program to allocate dynamically at run time. It consists of an area of memory that is divided into *allocated* and *unallocated* blocks. Normally, SMALL model programs cannot increase the size of the arena at run time; however, LARGE model programs can do so to a limited extent. The unallocated blocks together form the "free memory pool."

Portions of the arena can be allocated using the functions **malloc**, **calloc**, or **realloc**; returned to the free memory pool with **free**; or checked to see if they are allocated or not with **notmem**.

See Also

Definitions, extended STDIO, LARGE model, SMALL model, STDIO

argc — Definition

argc is the conventional name for the first argument to the function **main**. It is of type **int**. It gives the number of strings in the array pointed to by **argv**, which is the second argument to **main**.

By definition, the value of **argc** is never negative.

Cross-references

Standard, §2.1.2.2
The C Programming Language, ed. 2, p. 114

See Also

argv, Environment, envp, main

argument — Definition

An *argument* is an expression that appears between the parentheses of a function call or invocation of a function-like macro. Multiple arguments are separated by commas. For example, the following function call

```
example(arg1, arg2, arg3);
```

has three arguments.

Cross-references

Standard, §1.6
The C Programming Language, ed. 2, p. 201

See Also

conversions, Definitions, parameter

Notes

The Standard uses the term “argument” when it refers to the actual arguments of a function call or macro invocation. It uses the term “parameter” to refer to the formal parameters given in the definition of the function or macro.

argv — Definition

char *argv[];
argv is the conventional name for the second argument to the function **main**. It points to an array of pointers to type **char**. The strings to which **argv** points are passed by the host environment. Each may change the behavior of the program, and each may be modified by the program. Thus, the strings are called *program parameters*.

The number of pointers in the **argv** array is given by **argc**, which is the first argument to **main**. By definition, **argv[0]** always points to the name of the program. If the name is not available from the environment, then ***argv[0]** must be a null character. **argv[1]** through **argv[argc-1]** point to the set of program parameters; **argv[argc]** must be a null pointer.

Cross-references

Standard, §2.1.2.2
The C Programming Language, ed. 2, p. 114

See Also

argc, Environment, envp, main

array declarators — Definition

An *array declarator* declares an array. It can also establish the size of the array and cause storage to be allocated for it.

For example, consider the declaration:

```
int example[10];
```

The brackets '[ ]' establish that **example** is an array; the constant **10** establishes that **example** has ten elements. Thus, **example** is established to be an array of ten **int**s; memory is reserved for the ten members.

The constant expression that sets the size of an array must be an integral constant greater than zero. It must be known by translation phase 7 so the appropriate amount of storage can be allocated.

An array declarator may be empty; for example:

```
int example[];
```

In this case, **example** is an incomplete type. It will be completed when it is initialized.

Cross-references

Standard, §3.5.4.2
The C Programming Language, ed. 2, p. 216

See Also

[], declarators, initialization

Notes

For two array types to be compatible, the type of element in each, the number of dimensions in each, and the size of each corresponding dimension (except the first) must be identical.

as — Command

i8086 assembler
as [-bglx] [-o*file***]** *filename***.s ...**

as is a multipass assembler that will assemble functions written in i8086 assembly language. **as** will assemble programs into either SMALL or LARGE model, and will generate an object module in MS-DOS object format. It also supports i8087 opcodes, and it allows you to write functions in a model-independent manner.

as is not intended to be used for full-scale assembly-language programming; therefore, it does not include some of the more elaborate features found in full-fledged assemblers. For example, it has no facility for conditional compilation or user-defined macros. However, **Let's C** allows you to use preprocessor instructions to perform conditional assembly and expand macros. In addition, **as** optimizes branches to take advantage of short addressing forms, where the span of the branch permits.

File Names

All files of assembly language must have the suffix **.s** or **.m**. A **.s** file contains only assembly language, and may be assembled either directly by **as** using the command line shown below, or through the **cc** command. If you ask **as** to assemble a file that does not have the suffix **.s**, it will refuse to do so.

A file with the suffix **.m** is one that is passed through the C preprocessor **cpp** before it is assembled. These files *cannot* be assembled directly by **as**, but must be passed to the compiler controller **cc**, which will first invoke **cpp** and then **as**. For example, to assemble the file **foo.m**, use the instruction

```
cc foo.m
```

This allows you to use preprocessor instructions that conditionalize code within a file; for example, the same file can contain code for SMALL model and LARGE model, with **cpp** selecting the correct code when you assemble the file. An example of a **.m** file is given below. For more information on **.m** files, see the Lexicon entry for **larges.h**.

Usage

To invoke **as** directly through MS-DOS, use the following command:

> **as [-bglsx] [-o** *file***]** *filename***.s ...**

The named *files* are concatenated and the resulting object code is written either into the file specified by the **-o** option, or into the file **l.out** if the **-o** option is not used.

The other options are as follows:

-b Create a LARGE-model object module. This module hs two segments: *modname*_**code** and *modname*_**data**. By default, **as** creates an object module that is in SMALL model. See the Lexicon entry for **model** for more information on how these differ.

-g Give all symbols that are undefined at the end of the first pass the type **undefined external**, as though they had been declared with a **.globl** directive.

-l Generate a listing of your program. The listing is written to the standard output device; you can redirect it to a file or to the printer by using the '>' operator after the **as** command line.

-s Strip all non-global symbols from the symbol table. This option should be used with programs whose symbol tables are large enough to cause the linker **ld** to fail.

-x Strip all non-global symbols that begin with the character 'L' from the symbol table of the object module. This is a limited version of the **-s** option described above. It speeds up the linking of files by removing compiler-generated labels from the symbol table.

Lexical Conventions

Assembler tokens consist of identifiers (also called "symbols" or "names"), constants, and operators.

An *identifier* is a sequence of alphanumeric characters (including the period '.' and the underscore '_'). The first character must *not* be numeric. Only the first 16 characters of the name are significant; the remainder are quietly thrown away. Upper case and lower case are considered different. The machine instructions, assembly directives, and frequently used built-in symbols are in lower case.

The following lists the identifiers that represent the i8086 machine registers, which are predefined:

```
ax   sp   al   ah   cs
bx   bp   bl   bh   ds
cx   si   cl   ch   es
dx   di   dl   dh   ss
```

With regard to *constants*, the assembler uses the same syntax as the C compiler: A sequence of digits with a leading '0' is taken to be an octal constant. A sequence of digits with a leading '0x' is taken to be a hexadecimal constant; in this base, the letters 'A' through 'F' have the decimal values 10 through 15. Any strings of digits that do not begin with '0' are taken to be decimal constants.

A *character constant* consists of an apostrophe followed by an ASCII character. The constant's value is the ASCII code for the character, which is right-justified in the machine word. For example, an instruction to move the letter 'A' to the register **al** could be expressed in any of four equivalent ways:

```
movb    al $0x41      / hexadecimal
movb    al $0101      / octal
movb    al $'A        / character
movb    al $65        / decimal
```

The dollar sign indicates an immediate operand.

A blank space can be represented either as 0x20 (its ASCII value in hexadecimal), or as an apostrophe followed by a space ('), which on the page or screen resembles an apostrophe alone.

as represents character constants with the following escape sequences:

\b	backspace	(0010)
\f	form feed	(0014)
\n	newline	(0012)
\r	carriage return	(0015)
\t	tab	(0011)
\v	vertical tab	(0013)
*****nnn*	octal value	(0*nnn*)

The semicolon character ';' indicates a line break. This character must be used at the end of a line in a **.m** file, because the ANSI definition of the C preprocessor assumes that multi-line macro definitions are always a single logical line.

In the ANSI preprocessor, a macro expansion always occupies no more than one line, no matter how many lines the definition or the actual parameters to the macro span; therefore, you must embed semicolons in macros that you want to expand to more than one line. For example,

```
#define enter(n) .globl n;n: push si; push di
```

will be treated by **as** as if it read

```
.globl n
n:    push si
      push di
```

The following gives a more readable form of the macro **enter**:

```
#define enter(n)  .globl n;\
n:  push si;\
    push di
```

Blanks and Tabs

Blanks and tab characters may be used freely between tokens, but not within identifiers. A blank or a tabulation character is required to separate adjacent tokens not otherwise separated, e.g., between an instruction opcode and its first operand.

Comments

Comments are introduced by a slash ('/') and continue until the end of the line. All characters in comments are ignored by the assembler.

Program Sections

as permits you to divide programs into sections, each corresponding to a functional area of the address space. **as** gives each program section its own location counter during assembly.

Under SMALL model, a program can have up to eight program sections, which are organized into three groups, as shown below:

code:	**shri**	shared instruction
	prvi	private instruction
	bssi	uninitialized instruction
data:	**prvd**	private data
	bssd	uninitialized data
	shrd	shared data
	strn	strings
tables:	**symt**	symbol table

All Mark Williams assemblers use the same set of sections. This contributes to the portability of programs between operating systems. Not all the sections are distinct under MS-DOS, however; the meanings of the sections under MS-DOS are as follows:

shri (shared instruction) is the same as **prvi** (private instruction); *shared* refers to the sharing of physical memory between two or more concurrent processes, and this capability does not exist under MS-DOS. **prvi** is used for all code generated by the C compiler.

There is no distinction between **shrd** and **prvd**. The latter is used by the compiler for all external and static data that are explicitly initialized in a C program.

bssi and **bssd** are initialized to zero. **Let's C** uses the **bssd** section for external or static data that are not initialized: the C language guarantees that these data are in fact initialized to zeros. **Let's C** does not use the **bssi** section.

The **strn** (strings) section is actually a special part of the data section, that **Let's C** uses to store string constants. It is synonymous with **prvd** under MS-DOS.

The **symt** section contains the symbol table used by the linker. Both the C compiler and the assembler generate symbol tables that go in this section.

In most cases, you need not worry about what all these program sections are, and can simply write code under the keywords **.prvi** or **.shri**, and write data under the keywords **.prvd** or **.shrd**. Do not to place items in the **symt** section, because the C compiler, the assembler, and the linker use it to communicate among themselves.

Under LARGE model, the assembled module has two sections: *filename*_**code** and *filename*_**data**. The former contains all code, that is,w hat goes into the **shri**, **prvi**, and **bssi** sections in SMALL model. The latter contains all data, that is, what goes into the **shrd**, **prvd**, **bssd**, and **strn** sections under SMALL model.

When a program is assembled, the sections of a program are concatenated so that in the assembly listing the whole program looks like a solid block of code and data. All code sections are combined into the i8086 **code** segment, and all data sections into the i8086 **data** segment. The symbol table is not actually linked when the program is executed, and so is not assigned to any i8086 segment

The Current Location

The special symbol '.' (dot) is a counter that represents the current location. The current location can be changed by an assignment; for example:

```
. = .+START
```

The assignment must not cause the value to decrease, and it must not change the program section, i.e., the right-hand operand must be defined in the same section as the current section.

Expressions

An expression is a sequence of symbols representing a value and a program section. Expressions are made up of identifiers, constants, operators, and brackets. All binary operators have equal precedence and are executed in a strict left-to-right order (unless altered by brackets).

Notice that brackets '[' and ']' group expression elements, because parentheses are used for indexed register addressing.

Types

Every expression has a type determined by its operands. The simplest operands are *symbols*. The following names the types of symbols available:

Undefined — A symbol is *defined* if it is a constant or a label, or if it is assigned a defined value; otherwise, it is *undefined*. A symbol may become undefined if it is assigned the value of an undefined expression. It is an error to assemble an undefined expression in pass 2. Pass 1 allows assembly of undefined expressions, but phase errors may be produced if undefined expressions are used in certain contexts, such as in a **.blkw** or **.blkb**.

Absolute — An *absolute* symbol is one defined ultimately from a constant or from the difference of two relocatable values.

Register — These are the machine registers.

Relocatable — All other user symbols are relocatable symbols in some program section. Each program section is a different relocatable type.

Any keyword may be used in an expression to obtain the basic value of the keyword. This may be useful when employing the keywords that define machine instructions. The basic value of a machine operation by default has the highest opcode associated with it; for example

```
.word push
```

yields **FF**.

Note that the type of an expression does not include such attributes as length (word or byte), so the assembler will not remember whether you defined a particular variable to be a word or a byte. Addresses and constants have different types, but the assembler does not treat a constant as an immediate value unless it is preceded by a dollar sign '$'. If you use a constant where an address is expected, **as** will treat the constant like an address (and vice versa). You must distinguish between variables and addresses or immediate values.

Operators

The following lists the operators that **as** recognizes:

+	addition
-	subtraction
*	multiplication
-	unary negation
~	unary complement
^	type transfer
|	segment construction

Expressions may be grouped with brackets. Parentheses are reserved for use in address mode descriptions.

Type propagation

When operands are combined in expressions, the resulting type is a function of both the operator and the types of the operands. The '*', '~', and unary '-' operators can only manipulate absolute operands and always yield an absolute result.

The '+' operator signifies the addition of two absolute operands to yield an absolute result, and the addition of an absolute to a relocatable operand to yield a result with the same type as the relocatable operand.

The binary ‘-’ operator allows two operands of the same type, including relocatable, to be subtracted to yield an absolute result; it also allows an absolute to be subtracted from a relocatable, to yield a result with the same type as the relocatable operand.

The binary operator ‘^’ yields a result with the value of its left operand and the type of its right operand. It can be used to create expressions, usually used in an assignment statement, with any desired type.

Statements

A program consists of a sequence of statements separated by newlines or by semicolons. There are four kinds of statements: null statements, assignment statements, keyword statements, and machine instructions.

A statement can be proceded by any number of labels. There are two kinds of labels: *name* and *temporary*.

A *name* label consists of an identifier followed by a colon (**:**). The program section and value of the label are set to that of the current location counter. It is an error for the value of a label to change during an assembly. This most often happens when an undefined symbol is used to control a location counter adjustment.

A *temporary* label consists of a digit (“ toQ ’) followed by a colon ‘:’. It defines temporary symbols of the form ‘*n*f’ and ‘*n*b’, where ‘*n*’ is the digit of the label. References of the form ‘*n*f’ refer to the first temporary label ‘*n*:’ forward from the reference; those of the form ‘*n*b’ refer to the first temporary label ‘*n*:’ backward from the reference. Such labels conserve symbol table space in the assembler.

A *null statement* is an empty line, or a line containing only labels or a comment. It can occur anywhere. **as** ignores it, except in the case of a label, which **as** gives the current value of the location counter.

An *assignment statement* consists of an identifier followed by an equal sign ‘=’ and an expression. The value and program section of the identifier are set to that of the expression. Any symbol defined by an assignment statement may be redefined, either by another assignment statement or by a label. An assignment statement is equivalent to the **equ** keyword statement found in many assemblers.

Assembler directives

Assembler directives allow you to pass instructions directly to **as**. Each directive begins with a period, and most are followed by operands.

The following describes the directives that **as** recognizes:

.ascii *string*

The first non-white space character, typically a quotation mark, that appears after the keyword is taken as a delimiter. Successive characters are assembled into successive bytes until until the delimiter appears again. To include a quotation mark within a string, use another character for the delimiter.

It is an error if a newline is encountered before reaching the second delimiter. To insert a newline into a string, use the character constant ‘\n’, a described above.

.blkb/.blkw

Assemble blocks of bytes or words that are filled with zeroes. The size of the block is *expression* bytes or words.

.bssd Change the current program section to **bssd**. The current location is reset to the value of the **bssd** location counter.

.bssi Change the current program section to **bssi**. The current location is reset to the value of the **bssi** location counter.

.byte The *expression*s in the list are truncated to byte size and assembled into successive bytes. Expressions in the list are separated by commas.

.even/.odd
These insert a NULL byte, if necessary, to set the location counter to the next even or odd location, respectively. They are used to force alignment.

.globl The identifiers in the comma-separated list are marked as global. If they are defined in the current assembly, they may be referenced by other object modules; if they are undefined, they must be resolved by the linker before execution.

.page Force the printed listing of your assembly-language program to skip to the top of a new page by inserting a form-feed character into the file. The title is printed at the top of the page.

.prvd Change the current program section to **prvd**. The current location is reset to the value of the **prvd** location counter.

.prvi Change the current program section to **prvi**. The current location is reset to the value of the **prvi** location counter.

.shrd Change the current program section to **shrd**. The current location is reset to the value of the **shrd** location counter.

.shri Change the current program section to **shri**. The current location is reset to the value of the **shri** location counter.

.strn Change the current program section to **strn**. The current location is reset to the value of the **strn** location counter.

.title *string*
Print *string* at the top of every page in the listing. This directive also causes the listing to skip to a new page.

.word *expression* **[,** *expression* **]**
Truncate *expression*s to word length and assemble the resulting data into successive words. Expressions in the list are separated by commas.

Address descriptors

The source and destination descriptors use the following syntax. *r* refers to a register and the symbol *e* to an expression, as follows:

r: register
al, cl, dl, bl, ah, ch, dh, bh
ax, cx, dx, bx, sp, bp, si, di

e: direct address|
Any eight- or 16-bit number. Eight-bit numbers are sign extended.

(*r*): indexing
(si) (di) (bx)

f*e*(*r*): index displacement
e(si) *e*(di) *e*(bx): default segment is **ds**
e(bp): default segment is **ss**

(*r*,*r*): double index
(bx), si) (bx, di): default segment is **ds**
(bp, si) (bp, di): default segment is **ss**

e(*r*,*r*): double index with displacement
e(bx, si) *e*(bx, di): default segment is **ds**
e(bp, si) *e*(bp, di): default segment is **ss**

R*e*: immediate

s: segment register|
ss, ds, es, cs: allowed only where explicitliy stated.

Note that the dollar sign is always used to indicate an immediate value, even if the expression is a constant.

A direct address is interpreted as either a direct address or a PC-relative displacement, depending on the requirements of the instruction.

If an address descriptor indicates an indexing mode and the base expression is of type absolute, **as** uses the shortest displacement length (zero, one, or two bytes) that can hold the expression's value. Relocatable base expressions, whose values cannot be completely determined until the program is linked, are always assigned two-byte displacements.

Any address descriptor may be modified by a segment escape prefix. A segment escape prefix consists of a segment register name followed by a colon ':'. The escape causes **as** to produce a segment override prefix that uses the specified segment register as an operand. **as** does not produce segment override prefixes unless explicitly required by an instruction.

Instructions

The following machine instructions are defined. The examples illustrate the general syntax of the operands. Combinations that are syntactically valid may be forbidden for semantic reasons.

The examples use the following references:

a	general address
al	al register
ax	ax register
cl	cl register
d	direct address
dx	dx register
e	expression
$e	immediate expression
m	memory address (not an immediate)
p	port address

as treats as ordinary one-byte machine operations some operations that the Intel assembler ASM86 handles with special syntax; these include the *lock* and *repeat* prefixes. **as** makes no attempt to prevent the generation of incorrect sequences of these prefix bytes.

Although every machine operation has a type and value associated with it, in most cases the value was chosen to help **as** format the machine instructions.

For more information on these instructions, see the Intel *ASM86 Assembly Language Reference Manual*.

aaa	ASCII adjust AL after addition
aad	ASCII adjust AX before division
aam	ASCII adjust AX after multiply
aas	ASCII adjust AL after subtraction

adcb	*r, a*	Add with carry, byte
adc	*r, a*	Add with carry, word
adcb	*a, r*	Add with carry, byte
adc	*a, r*	Add with carry, word
adcb	*a, $e*	Add with carry, byte
adc	*a, $e*	Add with carry, word
addb	*r, a*	Add, byte
add	*r, a*	Add, word
addb	*a, r*	Add, byte
add	*a, r*	Add, word
addb	*a, $e*	Add, byte
add	*a, $e*	Add, word
andb	*r, a*	Logical and, byte
and	*r, a*	Logical and, word
andb	*a, r*	Logical and, byte
and	*a, r*	Logical and, word
andb	*a, $e*	Logical and, byte
and	*a, $e*	Logical and, word
call	*d*	Near call, PC-relative
cbw		Convert byte into word
clc		Clear carry flag
cld		Clear direction flag
cli		Clear interrupt flag
cmc		Complement carry flag
cmpb	*r, a*	Compare two operands, byte
cmp	*r, a*	Compare two operands, word
cmpb	*a, r*	Compare two operands, byte
cmp	*a, r*	Compare two operands, word
cmpb	*a, $e*	Compare two operands, byte
cmp	*a, $e*	Compare two operands, word
cmps		Compare string operands, bytes
cmpsb		Compare string operands, bytes
cmpsw		Compare string operands, words
cwd		Convert word to double
daa		Decimal adjust AL after addition
das		Decimal adjust AL after subtraction
decb	*a*	Decrement by one, byte
dec	*a*	Decrement by one, word
divb	*m*	Unsigned divide, byte
div	*m*	Unsigned divide, word
esc	*a*	Escape **0xD8**
hlt		Halt
icall	*a*	Near call, absolute offset at EA word
idivb	*m*	Signed divide, byte
idiv	*m*	Signed divide, word
ijmp	*a*	Jump short, absolute offset at EA word
imulb	*m*	Signed multiply, byte
imul	*m*	Signed multiply, word
inb	*al, p*	Input, byte
in	*ax, p*	Input, word
inb	*al, dx*	Input, byte
in	*ax, dx*	Input, word
incb	*a*	Increment by one, byte
inc	*a*	Increment by one, word

int	*e*	Call to interrupt
into		Call to interrupt, overflow
iret		Interrupt return
ja	*d*	Jump short if greater
jae	*d*	Jump short if greater or equal
jb	*d*	Jump short if less
jbe	*d*	Jump short if less or equal
jc	*d*	Jump short if carry
jcxz	*d*	Jump short if CX equals zero
je	*d*	Jump short if equal to
jg	*d*	Jump short if greater
jge	*d*	Jump short if greater or equal
jl	*d*	Jump short if less
jle	*d*	Jump short if less or equal
jmp	*d*	Jump short, PC-relative word offset
jmpb	*d*	Jump short, PC-relative byte offset
jmpl	*d*	Jump long
jna	*d*	Jump short if not above
jnae	*d*	Jump short if not above or equal
jnb	*d*	Jump short if not below
jnbe	*d*	Jump short if not below or equal
jnc	*d*	Jump short if not carry
jne	*d*	Jump short if not equal
jng	*d*	Jump short if not greater
jnge	*d*	Jump short if not greater or equal
jnl	*d*	Jump short if not less
jnle	*d*	Jump short if not less or equal
jno	*d*	Jump short if not overflow
jnp	*d*	Jump short if not parity
jns	*d*	Jump short if not sign
jnz	*d*	Jump short if not zero
jo	*d*	Jump short if overflow
jp	*d*	Jump short if parity
jpe	*d*	Jump short if parity even
jpo	*d*	Jump short if parity odd
js	*d*	Jump short if sign
jz	*d*	Jump short if zero
lahf		Load flags into AH register
lds	*r, a*	Load double pointer into DS
lea	*r, a*	Load effective address offset
les	*r, a*	Load double pointer into ES
lock		Assert BUS LOCK signal
lodsb		Load byte into AL
lods		Load byte into AL
lodsw		Load byte into AL
loop	*d*	Loop; decrement CX, jump short if CX less than zero
loope	*d*	Loop; decrement CX, jump short if CZ not zero and equal
loopne	*d*	Loop; decrement CX, jump short if CX not zero and not equal
loopnz	*d*	Loop; decrement CX, jump short if CZ not zero and ZF equals zero
loopz	*d*	Loop; decrement CX, jump short if CX not zero and zero
movb	*r, a*	Move, byte
mov	*r, a*	Move, word
movb	*a, r*	Move, byte
mov	*a, r*	Move, word

movb	*a, $e*	Move, byte
mov	*a, $e*	Move, word
movb	*a, s*	Move, byte
mov	*a, s*	Move, word
movb	*s, a*	Move, byte
mov	*s, a*	Move, word
movsb		Move string byte-by-byte
movs		Move string word-by-word
movsw		Move string word-by-word
mulb	*m*	Multiply, byte
mul	*m*	Multiply, word
negb	*a*	Two's complement negation, byte
neg	*a*	Two's complement negation, word
nop		No operation
notb	*a*	One's complement negation, byte
not	*a*	One's complement negation, word
orb	*r, a*	Logical inclusive OR, byte
or	*r, a*	Logical inclusive OR, word
orb	*a, r*	Logical inclusive OR, byte
or	*a, r*	Logical inclusive OR, word
orb	*a, $e*	Logical inclusive OR, byte
or	*a, $e*	Logical inclusive OR, word
outb	*p, al*	Output to port, byte
out	*p, ax*	Output to port, word
outb	*dx, al*	Output to port, byte
out	*dx, ax*	Output to port, word
pop	*m*	Pop a word from the stack
pop	*s*	Pop a word from the stack
popf		Pop fom stack into flags register
push	*m*	Push a word onto the stack
push	*s*	Push a word onto the stack
pushf		Push flags register onto the stack
rclb	*a, $1*	Rotate left $1 times, byte
rclb	*a, cl*	Rotate left CL times, byte
rcl	*a, $1*	Rotate left $1 times, word
rcl	*a, cl*	Rotate left CL times, word
rcrb	*a, $1*	Rotate right $1 times, byte
rcrb	*a, cl*	Rotate right CL times, byte
rcr	*a, $1*	Rotate right $1 times, word
rcr	*a, cl*	Rotate right CL times, word
rep		Repeat following string operation
repe		Find nonmatching bytes
repne		Repeat, not equal
repnz		Repeat, not equal
repz		Repeat, equal
ret		Return from procedure
rolb	*a, $1*	Rotate left, byte
rolb	*a, cl*	Rotate left, byte
rol	*a, $1*	Rotate left, word
rol	*a, cl*	Rotate left, word
rorb	*a, $1*	Rotate right, byte
rorb	*a, cl*	Rotate right, byte
ror	*a, $1*	Rotate right, word
ror	*a, cl*	Rotate right, word

sahf		Store AH into flags
salb	*a, $1*	Shift left, byte
salb	*a, cl*	Shift left, byte
sal	*a, $1*	Shift left, word
sal	*a, cl*	Shift left, word
sarb	*a, $1*	Shift right, byte
sarb	*a, cl*	Shift right, byte
sar	*a, $1*	Shift right, word
sar	*a, cl*	Shift right, word
sbbb	*r, a*	Integer subtract with borrow, byte
sbb	*r, a*	Integer subtract with borrow, word
sbbb	*a, r*	Integer subtract with borrow, byte
sbb	*a, r*	Integer subtract with borrow, word
sbbb	*a, $e*	Integer subtract with borrow, byte
sbb	*a, $e*	Integer subtract with borrow, word
scasb		Compare string data, byte
scas		Compare string data, word
shlb	*a, $1*	Shift left, byte
shlb	*a, cl*	Shift left, byte
shl	*a, $1*	Shift left, word
shl	*a, cl*	Shift left, word
shrb	*a, $1*	Shift right, byte
shrb	*a, cl*	Shift right, byte
shr	*a, $1*	Shift right, word
shr	*a, cl*	Shift right, word
stc		Set carry flag
std		Set direction flag
sti		Set interrupt enable flag
stosb		Store string data, byte
stos		Store string data, byte or word
stosw		Store string data, word
subb	*r, a*	Integer subtraction, byte
sub	*r, a*	Integer subtraction, word
subb	*a, r*	Integer subtraction, byte
sub	*a, r*	Integer subtraction, word
subb	*a, $e*	Integer subtraction, byte
sub	*a, $e*	Integer subtraction, word
testb	*r, a*	Logical compare, byte
test	*r, a*	Logical compare, word
testb	*a, r*	Logical compare, byte
test	*a, r*	Logical compare, word
testb	*a, $e*	Logical compare, byte
test	*a, $e*	Logical compare, word
wait		Wait until BUSY pin is inactive
xcall	*d, d*	Far call, immediate four-byte address
xchgb	*r, a*	Exchange memory, byte
xchg	*r, a*	Exchange memory, word
xicall		Far call, address at EA double word
xijmp		Jump far, address at memory double word
xjmp	*d, d*	Jump far, immediate four-byte address
xlat		Table look-up translation
xorb	*r, a*	Logical exclusive OR, byte
xor	*r, a*	Logical exclusive OR, word
xorb	*a, r*	Logical exclusive OR, byte

xor	*a, r*	Logical exclusive OR, word
xorb	*a, $e*	Logical exclusive OR, byte
xor	*a, $e*	Logical exclusive OR, word
xret		Return, intersegment

i8087 instructions

as can also generate object files that use the i8087 mathematics co-processor. The example instructions use the following references:

d	direct address
st0	floating point register 0
st1	any floating point register *except* 0

The following lists the i8087 instructions:

fabs		Absolute value
fadd	*st0, st1*	Add real
fadd	*st1, st0*	Add real
ffadd	*d*	Add real, float
fdadd	*d*	Add real, double
faddp		Add real and pop
faddp	*st, st0*	Add real and pop
fbld	*d*	Load packed decimal (BCD)
fbstp	*d*	Store packed decimal (BCD) and pop
fchs		Change sign
fclex		Clear exception
fnclex		Clear exception
fcom		Compare real
ffcom	*d*	Compare real, float
fdcom	*d*	Compare real, double
fcomp		Compare real and pop
fcomp	st1	Compare real and pop
ffcomp	*d*	Compare real and pop, float
fdcomp	*d*	Compare real and pop, double
fcompp		Compare real and pop twice
fdecstp		Decrement stack pointer
fdisi		Disable interrupts
fndisi		Disable interrupts, no operands
fdiv	*st0, st1*	Divide real
fdiv	*st1, st0*	Divide real
ffdiv	*d*	Divide real, float
fddiv	*d*	Divide real, double
fdivp		Divide real and pop
fdivp	st1	Divide real and pop
fdivr	*st0, st1*	Divide real reversed
fdivr	*st1, st0*	Divide real reversed
ffdivr	*d*	Divide real reversed, float
fddivr	*d*	Divide real reversed, double
fdivrp		Divide real reversed and pop
fdivrp	*st1*	Divide real reversed and pop
feni		Enable interrupts
fneni		Enable interrupts, no operands
ffree	*st1*	Free register
fiadd	*d*	Integer add
fladd	*d*	Integer add, long
ficom	*d*	Integer compare

flcom	*d*	Integer compare, long
ficomp	*d*	Integer compare and pop
flcomp	*d*	Integer compare and pop, long
fidiv	*d*	Integer divide
fldiv	*d*	Integer divide, long
fidivr	*d*	Integer divide reversed
fldivr	*d*	Integer divide, long reversed
fild	*d*	Integer load
flld	*d*	Integer load, long
fqld	*d*	Integer load, quad
fimul	*d*	Integer multiply
flmul	*d*	Integer multiply, long
fincstp		Increment stack pointer
finit		Initialize processor
fninit		Initialize processor
fist	*d*	Integer store
flst	*d*	Integer store, long
fistp	*d*	Integer store and pop
flstp	*d*	Integer store and pop, long
fqstp	*d*	Integer store and pop, quad
fisub	*d*	Integer subtract
flsub	*d*	Integer subtract, long
fisubr	*d*	Integer subtract reversed
flsubr	*d*	Integer subtract reversed, long
fld	*st1*	Load real
ffld	*d*	Load real, float
fdld	*d*	Load real, double
ftld	*d*	Load real, temp
fldcw	*d*	Load control word
fldenv	*d*	Load environment
fldlg2		Load log(10)2
fldln2		Load log(e)2
fldl2e		Load log(2)e
fldl2t		Load log(2)10
fldpi		Load pi
fldz		Load +0.0
fld1		Load +1.0
fmul		Multiply real
fmul	*st0, st1*	Multiply real
ffmul	*st1, st0*	Multiply real, float
fdmul	*d*	Multiply real, double
fmulp	*d*	Multiply real and pop
fnop	st1	No operation
fpatan		Partial arctangent
fprem		Partial remainder
fptan		Partial tangent
frndint		Round to integer
frstor	*d*	Restore saved state
fsave	*d*	Save state
fnsave	*d*	Save state
fscale		Scale
fsetpm		Set protected mode
fsqrt		Square root
fst	*st1*	Store real

ffst	*d*	Store real, float
fdst	*d*	Store real, double
fstcw	*d*	Store control word
fnstcw	*d*	Store control word
fstenv	*d*	Store environment
fnstenv	*d*	Store environment
fstp	*st1*	Store real and pop
ffstp	*d*	Store real and pop, float
fdstp	*d*	Store real and pop, double
ftstp	*d*	Store real and pop, temp
fstsw	*d*	Store status word
fnstsw	*d*	Store status word
fsub	*st0, st1*	Subtract real
fsub	*st1, st0*	Subtract real
ffsub	*d*	Subtract real, float
fdsub	*d*	Subtract real, double
fsubp		Subtract real and pop
fsubp	*st1*	Subtract real and pop
fsubr	*d*	Subtract real reversed
ffsubr	*d*	Subtract real reversed, float
fdsubr	*d*	Subtract real reversed, double
fsubrp		Subtract real reversed and pop
fsubrp	*st1*	Subtract real reversed and pop
ftst		Test stack top against +0.0
fwait		Wait while 8087 is busy
fxam		Examine stack top
fxch	*st1*	Exchange registers
fxch		Exchange registers
fxtract		Extract exponent and significance
fyl2x		Y*log(2)X
fyl2xp1		Y*log(2)(X+1)

Examples

The first example executes the program **hello.c** in a model-independent assembly language. If executed, it should be placed in a file called **hello.m**, and assembled through the **cc** command, as follows:

```
cc -o hello hello.m
```

The **cc** command will pass the program first to the C preprocessor **cpp**, and then to **as**. For more information, see the Lexicon entry for **larges.h**.

```
#include <larges.h>
     .prvd
Hi:  .ascii     "Hello world.\n"
     .shri
     Enter(main_)      /* Note use of C-style comments */
     mov  ax, $Hi      /* push offset of msg */
     push ax
#ifdef LARGEDATA
     mov  ax, $@Hi     /* push segment of msg */
     push ax
#endif
     Gcall    printf_
     add sp, $RASIZE
     Leave
```

The next example program, **strchar.s** defines a function **strchar** that returns the number of occurrences of a character in a string.

FILE: strchar.s

```
/
/
/  Count and return the occurrences
/  of a character in a string.
/
/     int
/     strchar(s, c)
/     char *s;
/     int c;
/
/

  .globl     strchar_   / Make the name known externally.

  strchar_:
       push    si         / Standard C function
       push    di         / linkage. Save the
       push    bp         / si, di, and bp registers
       mov     bp, sp     / and set up new frame pointer.

       mov     si, 8(bp)  / String ptr -> si.
       mov     bx, 10(bp) / Char -> bx (actually bl).
       sub     ax, ax     / Clear ax (count register).
       sub     cx, cx     / Clear cx.

  0:   movb    cl, (si)   / Get character from string.
       jcxz    2f         / End of string?
       cmpb    bl, cl     / No. Do chars match?
       jnz     1f         / No.
       inc     ax         / Yes. Increment count.

  1:   inc     si         / Bump string pointer
       jmp     0b         / and loop again.

  2:   pop     bp         / Standard C return
       pop     di         / linkage. Restore
       pop     si         / saved registers and
       ret                / go home.
```

The following C program, **main.c** uses **strchar** The assembly language listing that follows, **main.s** was produced from **main.c** by the **-VASM** option in **cc**. The listing has been edited, and comments added, to illustrate what is happening.

```
/* FILE: main.c */

        main()
        {
            int n;
            n = strchar("aardvark", ’a’);
        }

              .shri                / ‘‘code’’ program section.

              .globl   main_

        main_:

              .strn                / ‘‘string’’ program section.
```

```
L2:   .byte    0x61        / This is the string
      .byte    0x61        / ''aardvark''
      .byte    0x72
      .byte    0x64
      .byte    0x76
      .byte    0x61
      .byte    0x72
      .byte    0x6B
      .byte    0x00

      .shri                / Back to ''code''

      push  si             / Standard C function
      push  di             / linkage.  Save registers,
      push  bp             / set up new frame pointer (bp),
      mov   bp, sp         / and make room on stack
      sub   sp, $0x02      / for the auto int, ''n''

      mov   ax, $0x61      / Push the
      push  ax             / character 'a'.
      mov   ax, $L2        / Push the address
      push  ax             / of the string ''aardvark''
      call  strchar_       / Function call.
      add   sp, $0x04      / Remove args from stack.
      mov   -0x02(bp), ax  / Assign result to auto 'n'.

      mov   sp, bp         / Standard C return
      pop   bp             / linkage.  Adjust stack
      pop   di             / pointer, then restore
      pop   si             / registers and
      ret                  / go home.
```

See Also

C language, calling conventions, cc, larges.h, memory allocation

ASCII — Definition

ASCII is an acronym for the American Standard Code for Information Interchange. It is a table of seven-bit binary numbers that encode the letters of the alphabet, numerals, punctuation, and the most commonly used control sequences for printers and terminals.

The extended ASCII character set defines eight-bit encodings. The lower 127 characters are those of standard ASCII, and the higher 127 characters are also defined.

Though the standard ASCII character set is used commonly throughout the United States, other countries use the ISO 646 character set, which is an invariant subset of standard ASCII. See the entry on **trigraphs** for a discussion of the representing C characters in environments in which not all of the 127 ASCII characters are available.

The following table gives the lower 127 ASCII characters in octal, decimal, and hexadecimal numbers.

000	0	0x00	NUL	**<ctrl-@>**	Null character
001	1	0x01	SOH	**<ctrl-A>**	Start of header
002	2	0x02	STX	**<ctrl-B>**	Start of text
003	3	0x03	ETX	**<ctrl-C>**	End of text
004	4	0x04	EOT	**<ctrl-D>**	End of transmission
005	5	0x05	ENQ	**<ctrl-E>**	Enquiry
006	6	0x06	ACK	**<ctrl-F>**	Positive acknowledgement
007	7	0x07	BEL	**<ctrl-G>**	Alert
010	8	0x08	BS	**<ctrl-H>**	Backspace

011	9	0x09	HT	**<ctrl-I>**	Horizontal tab
012	10	0x0A	LF	**<ctrl-J>**	Line feed
013	11	0x0B	VT	**<ctrl-K>**	Vertical tab
014	12	0x0C	FF	**<ctrl-L>**	Form feed
015	13	0x0D	CR	**<ctrl-M>**	Carriage return
016	14	0x0E	SO	**<ctrl-N>**	Shift out
017	15	0x0F	SI	**<ctrl-O>**	Shift in
020	16	0x10	DLE	**<ctrl-P>**	Data link escape
021	17	0x11	DC1	**<ctrl-Q>**	Device control 1 (XON)
022	18	0x12	DC2	**<ctrl-R>**	Device control 2 (tape on)
023	19	0x13	DC3	**<ctrl-S>**	Device control 3 (XOFF)
024	20	0x14	DC4	**<ctrl-T>**	Device control 4 (tape off)
025	21	0x15	NAK	**<ctrl-U>**	Negative acknowledgement
026	22	0x16	SYN	**<ctrl-V>**	Synchronize
027	23	0x17	ETB	**<ctrl-W>**	End of transmission block
030	24	0x18	CAN	**<ctrl-X>**	Cancel
031	25	0x19	EM	**<ctrl-Y>**	End of medium
032	26	0x1A	SUB	**<ctrl-Z>**	Substitute
033	27	0x1B	ESC	**<ctrl-[>**	Escape
034	28	0x1C	FS	**<ctrl-\>**	Form separator
035	29	0x1D	GS	**<ctrl-]>**	Group separator
036	30	0x1E	RS	**<ctrl-^>**	Record separator
037	31	0x1F	US	**<ctrl-_>**	Unit separator
040	32	0x20	SP	Space	
041	33	0x21	!	Exclamation point	
042	34	0x22	"	Quotation mark	
043	35	0x23	#	Pound sign (sharp)	
044	36	0x24	$	Dollar sign	
045	37	0x25	%	Percent sign	
046	38	0x26	&	Ampersand	
047	39	0x27	’	Apostrophe	
050	40	0x28	(	Left parenthesis	
051	41	0x29	)	Right parenthesis	
052	42	0x2A	*	Asterisk	
053	43	0x2B	+	Plus sign	
054	44	0x2C	,	Comma	
055	45	0x2D	-	Hyphen (minus sign)	
056	46	0x2E	.	Period	
057	47	0x2F	/	Virgule (slash)	
060	48	0x30	0		
061	49	0x31	1		
062	50	0x32	2		
063	51	0x33	3		
064	52	0x34	4		
065	53	0x35	5		
066	54	0x36	6		
067	55	0x37	7		
070	56	0x38	8		
071	57	0x39	9		
072	58	0x3A	:	Colon	
073	59	0x3B	;	Semicolon	
074	60	0x3C	<	Less-than symbol (left angle bracket)	
075	61	0x3D	=	Equal sign	
076	62	0x3E	>	Greater-than symbol (right angle bracket)	

077	63	0x3F	?	Question mark
0100	64	0x40	@	At sign
0101	65	0x41	A	
0102	66	0x42	B	
0103	67	0x43	C	
0104	68	0x44	D	
0105	69	0x45	E	
0106	70	0x46	F	
0107	71	0x47	G	
0110	72	0x48	H	
0111	73	0x49	I	
0112	74	0x4A	J	
0113	75	0x4B	K	
0114	76	0x4C	L	
0115	77	0x4D	M	
0116	78	0x4E	N	
0117	79	0x4F	O	
0120	80	0x50	P	
0121	81	0x51	Q	
0122	82	0x52	R	
0123	83	0x53	S	
0124	84	0x54	T	
0125	85	0x55	U	
0126	86	0x56	V	
0127	87	0x57	W	
0130	88	0x58	X	
0131	89	0x59	Y	
0132	90	0x5A	Z	
0133	91	0x5B	[	Left bracket (left square bracket)
0134	92	0x5C	\	Backslash
0135	93	0x5D	]	Right bracket (right square bracket)
0136	94	0x5E	^	Circumflex
0137	95	0x5F	_	Underscore (underbar)
0140	96	0x60	‘	Grave
0141	97	0x61	a	
0142	98	0x62	b	
0143	99	0x63	c	
0144	100	0x64	d	
0145	101	0x65	e	
0146	102	0x66	f	
0147	103	0x67	g	
0150	104	0x68	h	
0151	105	0x69	i	
0152	106	0x6A	j	
0153	107	0x6B	k	
0154	108	0x6C	l	
0155	109	0x6D	m	
0156	110	0x6E	n	
0157	111	0x6F	o	
0160	112	0x70	p	
0161	113	0x71	q	
0162	114	0x72	r	
0163	115	0x73	s	
0164	116	0x74	t	

0165	117	0x75	u	
0166	118	0x76	v	
0167	119	0x77	w	
0170	120	0x78	x	
0171	121	0x79	y	
0172	122	0x7A	z	
0173	123	0x7B	{	Left brace (left curly bracket)
0174	124	0x7C	|	Vertical bar
0175	125	0x7D	}	Right brace (right curly bracket)
0176	126	0x7E	~	Tilde
0177	127	0x7F	DEL	Delete

See Also

Definitions, trigraph sequences

asctime() — Time function (libc)

Convert broken-down time to text
#include <time.h>
char *asctime(const struct tm ***timestruct*);**

The function **asctime** converts the data pointed to by *timestruct* into a text string of the form:

```
Wed Dec 10 13:57:33 1987\n\0
```

The structure pointed to by *timestruct* must first be initialized by either the function **gmtime** or the function **localtime** before it can be used by **asctime**. See the entry for **tm** for further information on this structure.

asctime returns a pointer to the string it creates.

Example

This example uses **asctime** to display Universal Coordinated Time.

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

main(void)
{
      printf(asctime(gmtime(NULL)));
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.12.3.1
The C Programming Language, ed. 2, p. 256

See Also

ctime, date and time, gmtime, localtime, strftime, time_t, tm

Notes

asctime writes its string into a static buffer that will be written by another call to either **asctime** or **ctime**.

The name "asctime" is short for "ASCII time"; its use, however, is not limited to implementations on ASCII systems.

The Standard describes the following algorithm with which **asctime** can generate its string:

```
char *
asctime(const struct tm *timeptr)
{
      static const char wday_name[7][3] = {
            "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"
      };
      static const char mon_name[12][3] = {
            "Jan", "Feb", "Mar", "Apr", "May", "Jun",
            "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
      };
      static char result[26];

      sprintf(result, "%.3s %.3s%3d %.2d:%.2d:%.2d %d\n",
            wday_name[timeptr->tm_wday],
            mon_name[timeptr->tm_mon],
            timeptr->tm_mday, timeptr->tm_hour,
            timeptr->tm_min, timeptr->tm_sec,
            1900 + timeptr->tm_year);

      return result;
}
```

asin() — Mathematics (libm)

Calculate inverse sine
#include <math.h>
double asin(double *arg***);**

asin calculates the inverse sine of *arg*, which must be in the range of from -1.0 to 1.0; any other value will trigger a domain error.

asin returns the result, which is in the range $\pi/2$ to π.

Cross-references

Standard, §4.5.2.2
The C Programming Language, ed. 2, p. 251

See Also

acos, atan, atan2, cos, mathematics, sin, tan

assert() — Diagnostics (assert.h)

Check assertion at run time
#include <assert.h>
void assert(int *expression***);**

assert checks the value of *expression*. If *expression* is false (zero), **assert** sends a message into the standard error stream and calls **abort**. It is useful for verifying that a necessary condition is true.

The error message includes the text of the assertion that failed, the name of the source file, and the line within the source file that holds the expression in question. These last two elements consist, respectively, of the values of the preprocessor macros _ _**FILE**_ _ and _ _**LINE**_ _.

Because **assert** calls **abort**, it never returns.

To turn off **assert**, define the macro **NDEBUG** prior to including the header **assert.h**. This forces **assert** to be redefined as

```
#define assert(ignore)
```

Example

This program generates an error if your implementation does not conform to the Standard.

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

main(void)
{
#ifdef STDC
      assert(STDC);
#else
      fprintf(stderr, "Not ANSI C\n");
#endif
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.2.1.1
The C Programming Language, ed. 2, p. 253

See Also

abort, assert.h, diagnostics, NDEBUG

Notes

The Standard requires that **assert** be implemented as a macro, not a library function. If a program suppresses the macro definition in favor of a function call, its behavior is undefined.

Turning off **assert** with the macro **NDEBUG** will affect the behavior of a program if the expression being evaluated normally generates side effects.

assert is useful for debugging, and for testing boundary conditions for which more graceful error recovery has not yet been implemented.

assert.h — Header

Header for assertions
#include <assert.h>

assert.h is the header file that defines the macro **assert**.

Cross-references

Standard, §4.2
The C Programming Language, ed. 2, pp

See Also

assert, diagnostics, header

atan() — Mathematics (libm)

Calculate inverse tangent
#include <math.h>
double atan(double *arg***);**

atan calculates the inverse tangent of *arg*, which may be any real number.

atan returns the result, which is in the range of from $-\pi/2$ to $\pi/2$ radians.

Cross-references

Standard, §4.5.2.3
The C Programming Language, ed. 2, p. 251

See Also

acos, asin, atan2, cos, mathematics, sin, tan

atan2() — Mathematics (libm)

Calculate inverse tangent
#include <math.h>
double atan2(double *num***, double** *den***);**

atan2 calculates the inverse tangent of the quotient of its arguments *num* and *den*. These may be any real number except zero.

atan2 returns the result, which is in the range of from -π to π. The sign of the return value is drawn from the signs of both arguments.

Cross-references

Standard, §4.5.2.4
The C Programming Language, ed. 2, p. 251

See Also

acos, asin, atan, cos, mathematics, sin, tan

Notes

atan2 is provided in addition to **atan**, to compute arc tangents for numbers that yield very large results.

atexit() — General utility (libc)

Register a function to be performed at exit
#include <stdlib.h>
int atexit(void (**function***)(void));**

atexit registers a function to be executed when the program exits. *function* points to the function to be executed. The registered function returns nothing. **atexit** provides a way to perform additional clean-up operations before a program terminates.

The functions that **atexit** registers are executed when the program exits normally, i.e., when the function **exit** is called or when **main** returns. The functions registered by **atexit** can perform clean-up is needed, beyond what is ordinarily performed when a program exits.

atexit returns zero if *function* could be registered, and nonzero if it could not.

Example

This example sets one function that displays messages when a program exits, and another that waits for the user to press a key before terminating.

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

void
lastgasp(void)
{
      perror("Type return to continue");
}
```

```
void
get1(void)
{
      getchar();
}

main(void)
{
      /* set up get1() as last exit routine */
      atexit(get1);
      /* set up lastgasp() as exit routine */
      atexit(lastgasp);

      /* exit, which invokes exit routines */
      exit(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.10.4.2
The C Programming Language, ed. 2, p. 253

See Also

exit, general utility

Notes

atexit must be able to register at least 32 functions.

Functions registered by **atexit** are executed when **exit** is called. They are executed in *reverse* order of registration.

atof() — General utility (libc)

Convert string to floating-point number
#include <stdlib.h>
double atof(const char **string***);**

atof converts the string pointed to by *string* into a double-precision floating point number, and returns the number it has built. It is equivalent to the call

```
strtod(string, (char **)NULL);
```

string must point to the text representation of a floating-point number. It can contain a leading sign, any number of decimal digits, and a decimal point. It can be terminated with an exponent, which consists of the letters 'e' or 'E' followed by an optional leading sign and any number of decimal digits. For example,

```
1.23
123e-2
123E-2
```

are strings that can be converted by **atof**.

atof ignores leading blanks and tabs; it stops scanning when it encounters any unrecognized character.

Cross-references

Standard, §4.10.1.1
The C Programming Language, ed. 2, p. 251

See Also

atoi, atol, general utility, strtod, strtol, strtoul

Notes

The character that **atof** recognizes as representing the decimal point depends upon the program's locale, as set by the function **setlocale**. See **localization** for more information.

The functionality of **atof** has largely been subsumed by the function **strtod**, but the Standard includes it because it is used so widely in existing code.

atoi() — General utility (libc)

Convert string to integer
#include <stdlib.h>
int atoi(const char ****string*);**

atoi converts the string pointed to by *string* into an integer. It is equivalent to the call

```
(int)strtol(string, (char **)NULL, 10);
```

The string pointed to by *string* may contain a leading sign and any number of numerals. **atoi** ignores all leading white space. It stops scanning when it encounters any non-numeral other than the leading sign character and returns the **int** it has built.

Cross-references

Standard, §4.10.1.2
The C Programming Language, ed. 2, p. 251

See Also

atof, atol, general utilities, strtod, strtol, strtoul

Notes

The functionality of **atoi** has largely been subsumed by the function **strtol**, but the Standard includes it because it is used so widely in existing code.

atol() — General utility (libc)

Convert string to long integer
#include <stdlib.h>
long atol(const char ****string*);**

atol converts the string pointed to by *string* to a **long**. It is equivalent to the call

```
strtol(string, (char **)NULL, 10);
```

The string pointed to by *string* may contain a leading sign and any number of numerals. **atol** ignores all leading white space. It stops scanning when it encounters any non-numeral other than the leading sign and returns the **long** it has built.

Cross-references

Standard, §4.10.1.3
The C Programming Language, ed. 2, p. 251

See Also

atof, atol, general utilities, strtod, strtol, strtoul

Notes

The functionality of **atol** has largely been subsumed by the function **strtol**, but the Standard includes it because it is used so widely in existing code.

auto — C keyword

Automatic storage duration
auto *type identifier*

The storage-class specifier **auto** declares that *identifier* has automatic storage duration.

Cross-references

Standard, §3.5.1
The C Programming Language, ed. 2, p. 210

See Also

storage-class specifiers, storage duration

aux — Operating system device

Logical device for serial port

MS-DOS gives names to its logical devices. **Let's C** uses these names to access these devices via MS-DOS.

aux is the logical device for the the serial port auxiliary device.

Example

The following example opens the auxiliary port and sends it the string **hello, world**.

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

main(void)
{
      FILE *fp, *fopen();
      if ((fp = fopen("aux", "w")) != NULL) {
            printf("aux enabled\n");
            fprintf(fp, "hello, world.\n");
      }
      else printf("aux: cannot open.\n");
      return EXIT_SUCCESS;
}
```

See Also

com1, con, crts, lpt1, nul, operating system devices