I

i8086 support — Overview

Let's C includes a number of routines that support the i8086 microprocessor. They are as follows:

_copy	Copy memory from one address to another
csreg	Read the CS segment register
dsreg	Read the DS segment register
esreg	Read the ES segment register
exargs	Parse the command line
execall	Pass a command to **command.com**
getanb	Get unbuffered input from **aux** device
getcnb	Get unbuffered input from **con** device
in	Read a word from a port
inb	Read a byte from a port
intcall	Call an MS-DOS interrupt
out	Output a word to a port
outb	Output a byte to a port
ptoreg	Convert C ponters to register pairs
PTR	Expand pointers to offset/segment
putanb	Send unbuffered output to **aux** device
putcnb	Send unbuffered output to **con** device
regtop	Set a pointer to value of register pair
ssreg	Read the SS segment register
_zero	Zero out a segment of memory

See Also

i8087, Library

Notes

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

i8087 — Technical information

Floating-point co-processor

The Intel i8087 is the mathematics coprocessor for the i8086/88 family of microprocessors. It greatly accelerates the computation of floating-point numbers.

Let's C includes two sets of libraries for use with the i8087: the *sensing* and the *non-sensing* libraries.

If your compiled program is always going to run on a computer system that includes an i8087, you should compile and link programs with the **-VNDP** option. This program will use the non-sensing libraries. These libraries contain instructions that perform floating point operations directly on the i8087 coprocessor; programs compiled with them will *not* operate correctly on a system which does not include an i8087.

You should *not* compile and link programs with the **-VNDP** option if your system does not include an i8087 coprocessor or if you want the compiled program to run on target systems that might or might not contain an i8087.

If you do *not* use the **-VNDP** option, **Let's C** by default will use its *sensing* libraries. These libraries check if an i8087 is present on the system on which the compiled program is being run. If one is present, the libraries use it to perform floating-point operations. If one is not present, the libraries emulate i8087 floating-point operations in software. The compiled program will be somewhat larger

than the same program compiled with the **-VNDP** option, because it will include the code to perform software floating point, and will run slightly slower. The program must be linked with the **-VROM** option if it is to run in ROM.

A program that uses floating-point numbers will not necessarily yield the same results when executed on systems with and without an i8087 coprocessor. In particular, the i8087 represents floating-point numbers internally with an 80-bit representation (64 fraction bits, 15 exponent bits, one sign bit), whereas **Let's C** software floating point uses the 64-bit IEEE representation internally (52 fraction bits, 11 exponent bits, one sign bit). Thus, the low-order digits of floating point computations may differ on systems with and without an i8087.

Compatability With Previous Versions

Versions of **Let's C** prior to 4.0 used DECVAX format rather than IEEE format for software floating point operations. Any program using software floating point that was compiled by a previous version must be recompiled with the current version to use IEEE software floating point. Binary files that include DECVAX format floating point data are not compatible with the current IEEE floating point version.

Checking Presence of the i8087

In i8087 sensing mode, the C runtime startup routine discovers whether an i8087 is present on the machine. This datum is written into the global **char _has8087**. Zero indicates that an i8087 is absent, and a value other than zero indicates that it is present.

If you wish, you can read and change this variable. If you wish to test how a program would work without an i8087, it is easier to clear this byte than to pull the i8087 chip out of your computer. If, however, you set this byte to a non-zero value and an i8087 is not present, your computer will hang when it tries to use the non-existent i8087.

See Also

float, double, technical information

Notes

The assembler **as** will assemble programs that use i8087 opcodes. For a full table of these opcodes, see the entry for **as**.

identifiers — Overview

An *identifier* names one of the following lexical elements:

- Functions
- Labels
- Macros
- Members of a structure, a **union**, or an enumeration
- Objects
- Tags
- **typedef**s

An identifier with internal linkage may have up to at least 31 characters, which may be in either upper or lower case. An identifier with external linkage, however, may have up to at least six characters, and it is not required to recognize both upper and lower case. These limits are defined by the implementation, and may be increased by it.

An identifier is a string of digits and non-digits, beginning with a non-digit. For a translator to know that two identifiers refer to the same entity, the identifiers must be identical. If two identifiers are

meant to refer to the same entity yet differ in any character, the behavior is undefined.

Keywords in C are reserved. Therefore, no identifier may match a keyword.

The Standard allows the programmer to use leading underscores '_' to name internal identifiers, but reserves for the implementation all external identifiers with leading underscores. To reduce "name space pollution," the implementor should not reserve anything that is not explicitly defined in the Standard and that does not begin with a leading underscore.

Identifiers have both *scope* and *linkage*. The scope of an identifier refers to the portion of a program to which it is "visible." An identifier can have program scope, file scope, function scope, or block scope; for more information, see the entry for **scope**. The linkage of an identifier describes whether it is joined only with its name-sakes within the same file, or can be joined to other files. Linkage can be external, internal, or none. For more information, see the entry for **linkage**.

Cross-references

Standard, §3.1.2
The C Programming Language, ed. 2, p. 192

See Also

digit, external name, function prototype, internal name, lexical elements, linkage, name space, nondigit, scope, storage duration, string literal, types

if — C keyword

Conditionally execute an expression
if(*conditional***)** *statement***;**

if is a C keyword that conditionally executes an expression. If *conditional* is nonzero, then *statement* is executed. However, if *conditional* is zero, then *statement* is not executed.

conditional must use a scalar type. It may be a function call (in which case **if** evaluates what function returns), an integer, the result of an arithmetic operation, or the value returned by a relational expression.

An **if** statement can be followed by an **else** statement, which also introduces a statement. If *conditional* is nonzero, then the statement introduced by **if** is executed and the one introduced by **else** is ignored; whereas if *conditional* is equal to zero, then the statement introduced by **if** is ignored and the one introduced by **else** is executed.

Example

For an example of this statement, see **exit**.

Cross-references

Standard, §4.6.4.1
The C Programming Language, ed. 2, pp. 55*ff*

See Also

else, statements, switch

Notes

If the statement controlled by an **if** statement is accessed via a label, the statement controlled by an **else** statement associated with the **if** statement is not executed.

implicit conversions — Definition

The term *implicit conversion* means that the type of an object is changed by the translator without the direct intervention of the programmer. For a list of the rules for implicit conversion, see **conversion**.

Cross-reference

Standard, §3.2

See Also

conversions, explicit conversion

inb() — Extended function (libc)

Read from a port
int inb(int *port***);**

inb provides a C interface to the i8086 machine instruction **in**. It reads a byte (eight bits) from *port*, and returns it as an integer (16 bits).

Example

This example writes a file to the serial port. It uses **inb** to read the current status of the port.

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

/* DOS magic numbers */
#define PRINTER_STATUS 0x3BD
#define PRINTER_OUT 0x3BC
#define PRINTER_BUSY 0x80

main(int argc, char *argv[])
{
      FILE *fp;
      int  data;

      if(argc != 2)
            printf("Usage: print filename\n");
      else if ((fp = fopen(argv[1], "r")) == NULL) {
            printf("Cannot open %s\n", argv[1]);
            exit(EXIT_FAILURE);
      }

      else while((data = getw(fp)) != EOF) {
            while(inb(PRINTER_STATUS) & PRINTER_BUSY)
                  ;
            outb(PRINTER_OUT, data);
      }
      return EXIT_SUCCESS;
}
```

See Also

extended miscellaneous, in, out, outb

INCDIR — Environmental variable

Directory that holds include files

INCDIR names the default directory where **Let's C** seeks its header files. For example, the command

LEXICON

```
set INCDIR=a:\include
```

tells **cc** to look for header files in directory **include** on drive A. This directory is searched, as is the directory that holds the C source files and the directories named with **-I** options to the **cc** command, if any.

It is recommended that you set **INCDIR** in **autoexec.bat** to ensure that it is always set correctly.

See Also

cc, environmental variable

index() — Extended function (libc)

Find a character in a string
char *index(char **string***, **char** *character***);**

index is identical to the ANSI function **strchr**. It scans the given *string* for the first occurrence of *character*. If it finds *character*, it returns a pointer to it. If it does not find *character*, **index** returns NULL.

Having **index** search for a null character will always produce a pointer to the end of a string. For example,

```
char *string;
assert(index(string, 0)==string+strlen(string));
```

will never fail.

Example

For an example of this function, see the entry for **strncpy**.

See Also

extended miscellaneous, memchr, pnmatch, rindex, strchr, strpbrk

Notes

index is not described in the ANSI Standard. It is recommended that you use **strchr** instead of **index** so your programs will more closely approach strict conformity with the Standard.

initialization — Definition

The term *initialization* refers to setting a variable to its first, or initial, value.

Rules of Initialization

Initializers follow the same rules for type and conversion as do assignment statements.

If a static object with a scalar type is not explicitly initialized, it is initialized to zero by default. Likewise, if a static pointer is not explicitly initialized, it is initialized to NULL by default. If an object with automatic storage duration is not explicitly initialized, its contents are indeterminate.

Initializers on static objects must be constant expressions; greater flexibility is allowed for initializers of automatic variables. These latter initializers can be arbitrary expressions, not just constant expressions. For example,

```
double dsin = sin(30);
```

is a valid initializer, where **dsin** is declared inside a function.

To initialize an object, use the assignment operator '='. The following sections describe how to initialize different classes of objects.

Scalars

To initialize a scalar object, assign it the value of a expression. The expression may be enclosed within braces; doing so does not affect the value of the assignment. For example, the expressions

```
int example = 7+12;
```

and

```
int example = { 7+12 };
```

are equivalent.

Unions and Structures

The initialization of a **union** by definition fills only its *first* member.

To initialize a **union**, use an expression that is enclosed within braces:

```
union example_u {
      int member1;
      long member2;
      float member3;
} = { 5 };
```

This initializes **member1** to five. That is to say, the **union** is filled with an **int**-sized object whose value is five.

To initialize a structure, use a list of constants or expressions that are enclosed within braces. For example:

```
struct example_s {
      int member1;
      long member2;
      union example_u member3;
};

struct example_s test1 = { 5, 3, 15 };
```

This initializes **member1** to five, initializes **member2** to three, and initializes the *first* member of **member3** to 15.

Strings and Wide Characters

To initialize a string pointer or an array of wide characters, use a string literal.

The following initializes a string:

```
char string[] = "This is a string";
```

The length of the character array is 17 characters: one for every character in the given string literal plus one for the null character that marks the end of the string.

If you wish, you can fix the length of a character array. In this case, the null character is appended to the end of the string only if there is room in the array. For example, the following

```
char string[16] = "This is a string";
```

writes the text into the array **string**, but does not include the concluding null character because there is not enough room for it.

The same rules apply to initializing an array of wide characters. For example, the following:

```
wchar_t widestring[] = L"This is a string";
```

LEXICON

fills **widestring** with the wide characters corresponding to the characters in the given string literal. The appropriate form of the null character is then appended to the end of the array, and the size of the array is **(17*sizeof(wchar_t))**. The prefix **L** indicates that the string literal consists of wide characters.

A pointer to **char** can also be initialized when the pointer is declared. For example:

```
char *strptr = "This is a string";
```

initializes **strptr** to point to the first character in **This is a string**. This declaration automatically allocates exactly enough storage to hold the given string literal, plus the terminating null character.

Arrays

To initialize an array, use a list of expressions that is enclosed within braces. For example, the expression

```
int array[] = { 1, 2, 3 };
```

initializes **array**. Because **array** does not have a declared number of elements, the initialization fixes its number of elements at three. The elements of the array are initialized in the order in which the elements of the initialization list appear. For example, **array[0]** is initialized to one, **array[1]** to two, and **array[2]** to three.

If an array has a fixed length and the initialization list does not contain enough initializers to initialize every element, then the remaining elements are initialized in the default manner: static variables are initialized to zero, and other variables to whatever happens to be in memory. For example, the following:

```
int array[3] = { 1, 2 };
```

initializes **array[0]** to one, **array[1]** to two, and **array[2]** to zero.

The initialization of a multi-dimensional array is something of a science in itself. The Standard defines that the ranks in an array are filled from right to left. For example, consider the array:

```
int example[2][3][4];
```

This array contains two groups of three elements, each of which consists of four elements. Initialization of this array will proceed from **example[0][0][0]** through **example[0][0][3]**; then from **example[0][1][0]** through **example[0][1][3]**; and so on, until the array is filled.

It is easy to check initialization when there is one initializer for each "slot" in the array; e.g.,

```
int example[2][3] = {
       1,  2,  3,  4,  5,  6
};
```

or:

```
int example[2][3] = {
      {  1,  2,  3 }, {  4,  5,  6 }
};
```

The situation becomes more difficult when an array is only partially initialized; e.g.,

```
int example[2][3] = {
       { 1 }, { 2, 3 }
};
```

which is equivalent to:

```
int example[2][3] = {
      { 1, 0, 0 }, { 2, 3, 0 }
};
```

As can be seen, braces mark the end of initialization for a "cluster" of elements within an array. For example, the following:

```
int example[2][3][4] = {
      5, { 1, 2 }, { 5, 2, 4, 3 }, { 9, 9, 5 },
      { 2, 3, 7 } };
```

is equivalent to entering:

```
int example[2][3][4] = {
      { 5, 0, 0, 0 },
      { 1, 2, 0, 0 },
      { 5, 2, 4, 3 },

      { 9, 9, 5, 0 },
      { 2, 3, 7, 0 },
      { 0, 0, 0, 0 }
};
```

The braces end the initialization of one cluster of elements; the next cluster is then initialized. Any elements within a cluster that have not yet been initialized when the brace is read are initialized in the default manner.

The final entry in a list of initializers may end with a comma. For example:

```
int array[3] = { 1, 2, 3, };
```

will initialize **array** correctly. This is a departure from many current implementations of C.

ANSI C requires that the initializers of a multi-dimensional array be parsed in a top-down manner. Some implementations had parsed such initializers in a bottom-up manner. Code that expects bottom-up parsing may behave differently under ANSI C, and probably without warning. This is a quiet change that may require that some code be rewritten.

Cross-references

Standard, §3.5.7
The C Programming Language, ed. 2, pp. 218*ff*

See Also

array, declarations

int — C keyword

The type **int** holds an integer. It is usually the same size as a word (or register) on the target machine.

int is a signed integral type. This type can be no smaller than an **short** and no greater than a **long**.

A **int** can encode any number between **INT_MIN** and **INT_MAX**. These are macros that are defined in the header **limits.h**;

The types **signed** and **signed int** are synonyms for **int**.

Cross-references

Standard, §2.2.4.2, §3.1.2.5, §3.2.1.1, §3.5.2
The C Programming Language, ed. 2, p. 211

LEXICON

See Also

types

Notes

Because **int**s may be the size of **short**s on some machines and the size of **long**s on others, programs that are meant to be portable can avoid bugs by explicitly declaring all **int**s to be either **short** or **long**.

intcall() — i8086 support (libc)

Call MS-DOS interrupt
#include <dos.h>
int intcall(struct reg **srcreg***, struct reg** **destreg***, int** *intnum***);**

intcall lets you call MS-DOS interrupts. The arguments *srcreg* and *destreg* point to elements in the structure **reg**, which is defined in the header file **dos.h**, as follows:

```
struct reg {
      unsigned r_ax;
      unsigned r_bx;
      unsigned r_cx;
      unsigned r_dx;
      unsigned r_si;
      unsigned r_di;
      unsigned r_ds;
      unsigned r_es;
      unsigned r_flags;
};
```

intcall sets the processor registers to the values given in *srcreg*, without setting the processor flags. Then it calls the interrupt specified by *intnum* to perform the desired system function. Most often, the manifest constant **DOSINT** (0x21) is used, although **intcall** can handle almost all MS-DOS interrupts. Finally, it sets the structure pointed to by *destreg* to the values of those registers, and returns.

Example

The following program uses function 8 of interrupt 21, which receives raw input from the keyboard and does not echo it on the screen. The program receives up to 80 characters typed at the keyboard, and echoes them to the screen either when the carriage return is pressed or when the limit of 80 characters is exceeded.

The sample program **fdir.c**, which is included with your copy of **Let's C**, also demonstrates **intcall**.

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

char getch(void)
{
      struct reg r;

      r.r_ax = CONRAW;
      intcall(&r, &r, DOSINT);

      /* mask off top of ax pair */
      return (r.r_ax & 0xff);
}
```

```
main(void)
{
      char string[80];
      int i;

      for(i = 0; i <80; i++)
            if((string[i] = getch()) == '\r')
                  break;

      printf("%s\n", string);
      return(EXIT_SUCCESS);
}
```

See Also

dos.h, i8086 support, ptoreg, PTR, regtop, signals/interrupts

Notes

Registers that are not included in the structure **reg** cannot be passed to a system routine explicitly.

incall cannot use interrupts 25 and 26, absolute disk read and write, because they do not restore the stack correctly when they exit.

integer constant — Definition

An *integer constant* is a constant that holds an integer. An integer constant has the following structure:

- It begins with a digit.
- It has no period or exponent.
- It may have a prefix that indicates its base, as follows: **0X** and **0x** both indicate hexadecimal. **0** (zero) indicates octal.
- It may have a suffix that indicates its type. **u** and **U** indicate an unsigned integer; **l** and **L** indicate a long integer.

A hexadecimal number may consist of the digits '0' through '9' and the letters 'a' through 'f' or 'A' through 'F'. An octal number may consist of the digits '0' through '7'.

When an integer constant initializes a variable, the form of the constant should match that of the variable as closely as possible. For example, when an integer constant initializes a **long int**, the constant should have the suffix **l** or **L**. If the constant does not have this suffix, the variable may not be initialized correctly.

The type of an integer constant is fixed by the following rules:

- A decimal integer constant that has no suffix is given the *first* of the following types that can represent its value: **int**, **long int**, or **unsigned long int**.
- A hexadecimal or octal integer constant that has no suffix is given the first of the following types that can represent its value: **int**, **unsigned int**, **long int**, or **unsigned long int**.
- An integer constant with the prefixes **u** or **U** is given the first of the following types that can represent its value: **unsigned int** or **unsigned long int**.
- An integer constant with the prefixes **l** or **L** is given the first of the following types that can represent its value: **long int** or **unsigned long int**.
- An integer constant with both the unsigned and the long suffixes is an **unsigned long int**.

These rules, as they preserve the value of a given constant, are part of what is known as the *value-*

preserving rules.

Cross-references

Standard, §3.1.3.2
The C Programming Language, ed. 2, p. 193

See Also

constants, conversions

internal name — Definition

An *internal name* is an identifier that has internal linkage. The minimum maximum for the length of an internal name is 31 characters, and an implementation must distinguish upper-case and lower-case characters.

Cross-references

Standard, §3.1.2
The C Programming Language, ed. 2, p. 35

See Also

external name, identifiers, linkage

interrupt — Definition

An *interrupt* is an interruption of the sequential flow of a program. It can be generated by the hardware, from within the program itself, or from the operating system.

See Also

Definitions, intcall, interrupt handling, interrupts

isalnum() — Character handling (ctype.h)

Check if a character is a numeral or letter
#include <ctype.h>
int isalnum(int *c***);**

The macro **isalnum** tests whether *c* is a letter or a numeral. A letter is any character for which **isalpha** returns true; likewise, a numeral is any character for which **isdigit** returns true. *c* must be a value that is representable as an **unsigned char** or **EOF**.

isalnum returns nonzero if *c* is a letter or a numeral, and zero if it is not.

Cross-references

Standard, §4.3.1.1
The C Programming Language, ed. 2, pp

See Also

character handling

Notes

The operation of this function is affected by the program's locale, as set by the function **setlocale**. See **localization** for more information.

isalpha() — Character handling (ctype.h)

Check if a character is a letter
#include <ctype.h>
int isalpha(int *c***);**

The macro **isalpha** tests whether *c* is a letter. In the **C** locale, a letter is any of the characters 'a' through 'z' or 'A' through 'Z'. In any other locale, a letter is any character for which the functions **iscntrl**, **isdigit**, **ispunct**, and **isspace** all return false. *c* must be a value that is representable as an **unsigned char** or **EOF**.

isalpha returns nonzero if *c* is an alphabetic character, and zero if it is not.

Cross-references

Standard, §4.3.1.2
The C Programming Language, ed. 2, p. 249

See Also

character handling

Notes

The operation of this function is affected by the program's locale, as set by the function **setlocale**. See **localization** for more information.

isascii() — Extended macro (xctype.h)

Check if a character is an ASCII character
#include <xctype.h>
int isascii(*c***) int** *c***;**

The macro **isascii** tests whether the argument *c* is an ASCII character (0 <= *c* <= 0177). It returns a number other than zero if *c* is an ASCII character, and zero if it is not. Many other **ctype** macros will fail if passed a non-ASCII value other than **EOF**.

See Also

extended character handling

Notes

To conform to the ANSI Standard, this macro has been moved from the header **ctype.h** to the header **xctype.h**. This may require that some code be altered.

This macro is not described in the ANSI Standard. Any program that uses it does not conform strictly to the Standard, and may not be portable to other compilers or environments.

iscntrl() — Character handling (ctype.h)

Check if a character is a control character
#include <ctype.h>
int iscntrl(int *c***);**

The macro **iscntrl** tests whether *c* is a control character under the implementation's character set. The Standard defines a control character as being a character in the implementation's character that cannot be printed. *c* must be a value that is representable as an **unsigned char** or **EOF**.

iscntrl returns nonzero if *c* is a control character, and zero if it is not.

Cross-references

Standard, §4.3.1.3

LEXICON

The C Programming Language, ed. 2, p. 249

See Also

character handling

Notes

The operation of this function is affected by the program's locale, as set by the function **setlocale**. See **localization** for more information.

isdigit() — Character handling (ctype.h)

Check if a character is a numeral
#include <ctype.h>
int isdigit(int *c***);**

The macro **isdigit** tests whether *c* is a numeral (any of the characters '0' through '9'). *c* must be a value that is representable as an **unsigned char** or **EOF**.

isdigit returns nonzero if *c* is a numeral, and zero if it is not.

Cross-references

Standard, §4.3.1.4
The C Programming Language, ed. 2, p. 249

See Also

character handling

isgraph() — Character handling (ctype.h)

Check if a character is printable
#include <ctype.h>
int isgraph(int *c***);**

The macro **isgraph** tests whether *c* is a printable letter within the **Let's C** character set, but excluding the space character. The Standard defines a printable character as any character that occupies one printing position on an output device. *c* must be a value that is representable as an **unsigned char** or **EOF**.

isgraph returns nonzero if *c* is a printable character (except for space), and zero if it is not.

Cross-references

Standard, §4.3.1.5
The C Programming Language, ed. 2, p. 249

See Also

character handling

Notes

The operation of this function is affected by the program's locale, as set by the function **setlocale**. See **localization** for more information.

islower() — Character handling (ctype.h)

Check if a character is a lower-case letter
#include <ctype.h>
int islower(int *c***);**

The macro **islower** tests whether *c* is a lower-case letter. In the **C** locale, a lower-case letter is any of the characters 'a' through 'z'. In any other locale, this is a character for which the functions **iscntrl**,

isdigit, **ispunct**, **isspace**, and **isupper** all return false. *c* must be a value that is representable as an **unsigned char** or **EOF**.

islower returns nonzero if *c* is is a lower-case letter, and zero if it is not.

Cross-references

Standard, §4..1.6
The C Programming Language, ed. 2, p. 249

See Also

character handling, character set

Notes

The operation of this function is affected by the program's locale, as set by the function **setlocale**. See **localization** for more information.

isprint() — Character handling (ctype.h)

Check if a character is printable
#include <ctype.h>
int isprint(int *c***);**

The macro **isprint** tests whether *c* is a printable letter within the implementation's character set, including the space character. The Standard defines a printable character as any character that occupies one printing position on an output device. *c* must be a value that is representable as an **unsigned char** or **EOF**.

isprint returns nonzero if *c* is a printable character, and zero if it is not.

Cross-references

Standard, §4.3.1.7
The C Programming Language, ed. 2, p. 249

See Also

character handling

Notes

The operation of this function is affected by the program's locale, as set by the function **setlocale**. See **localization** for more information.

ispunct() — Character handling (ctype.h)

Check if a character is a punctuation mark
#include <ctype.h>
int ispunct(int *c***);**

The macro **ispunct** tests whether *c* is a punctuation mark in the implementation's character set. The Standard defines a punctuation mark as being any printable character, except the space character, for which the function **isalnum** returns false. *c* must be a value that is representable as an **unsigned char** or **EOF**.

ispunct returns nonzero if *c* is a punctuation mark, and zero if it is not.

Cross-references

Standard, §4.3.1.8
The C Programming Language, ed. 2, p. 249

LEXICON

See Also

character handling

Notes

The operation of this function is affected by the program's locale, as set by the function **setlocale**. See **localization** for more information.

isspace() — Character handling (ctype.h)

Check if character is white space
#include <ctype.h>
int isspace(int *c***);**

The macro **isspace** tests whether *c* represents a white-space character. In the **C** locale, a white-space character is any of the following: space (' '), form feed ('\f'), newline ('\n'), carriage return ('\r'), horizontal tab ('\t'), or vertical tab ('\v'). In any other locale, a white-space character is one for which the functions **isalnum**, **iscntrl**, **isgraph**, and **ispunct** all return false. *c* must be a value that is representable as an **unsigned char** or **EOF**.

isspace returns nonzero if *c* is a space character, and zero if it is not.

Cross-references

Standard, §4.3.1.1
The C Programming Language, ed. 2, p. 249

See Also

character handling

Notes

The operation of this function is affected by the program's locale, as set by the function **setlocale**. For example, Middle-Eastern languages use alternate characters to denote white space. See **localization** for more information.

isupper() — Character handling (ctype.h)

Check if a character is an upper-case letter
#include <ctype.h>
int isupper(int *c***);**

The macro **isupper** tests whether *c* is a upper-case letter. In the **C** locale, a upper-case letter is any of the characters 'A' through 'Z'. In any other locale, this is a character for which the functions **iscntrl**, **isdigit**, **islower**, **ispunct**, and **isspace** all return false. *c* must be a value that is representable as an **unsigned char** or **EOF**.

isupper returns nonzero if *c* is an upper-case letter, and zero if it is not.

Cross-references

Standard, §4.3.1.6
The C Programming Language, ed. 2, p. 249

See Also

character handling, character sets

Notes

The operation of this function is affected by the program's locale, as set by the function **setlocale**. See **localization** for more information.

isxdigit() — Character handling (libc)

Check if a character is a hexadecimal numeral
#include <ctype.h>
int isxdigit(int *c***);**

isxdigit tests whether *c* is a hexadecimal numeral (any of the characters '0' through '9', any of the letters 'a' through 'd', or any of the letters 'A' through 'D'). *c* must be a value that is representable as an **unsigned char** or **EOF**.

isxdigit returns nonzero if *c* is a hexadecimal numeral, and zero if it is not.

Cross-references

Standard, §4.3.1.11
The C Programming Language, ed. 2, p. 249

See Also

character handling