LAPIS SEMICONDUCTOR

SQ003077E001

RTLU8
Runtime Library Reference

Fourth Edition
Oct. 2011

NOTICE

No copying or reproduction of this document, in part or in whole, is permitted without the consent of LAPIS Semiconductor Co., Ltd.

The content specified herein is subject to change for improvement without notice.

The content specified herein is for the purpose of introducing LAPIS Semiconductor's products (hereinafter "Products"). If you wish to use any such Product, please be sure to refer to the specifications, which can be obtained from LAPIS Semiconductor upon request.

Examples of application circuits, circuit constants and any other information contained herein illustrate the standard usage and operations of the Products. The peripheral conditions must be taken into account when designing circuits for mass production.

Great care was taken in ensuring the accuracy of the information specified in this document. However, should you incur any damage arising from any inaccuracy or misprint of such information, LAPIS Semiconductor shall bear no responsibility for such damage.

The technical information specified herein is intended only to show the typical functions of and examples of application circuits for the Products. LAPIS Semiconductor does not grant you, explicitly or implicitly, any license to use or exercise intellectual property or other rights held by LAPIS Semiconductor and other parties. LAPIS Semiconductor shall bear no responsibility whatsoever for any dispute arising from the use of such technical information.

The Products specified in this document are intended to be used with general-use electronic equipment or devices (such as audio visual equipment, office-automation equipment, communication devices, electronic appliances and amusement devices).

The Products specified in this document are not designed to be radiation tolerant.

While LAPIS Semiconductor always makes efforts to enhance the quality and reliability of its Products, a Product may fail or malfunction for a variety of reasons.

Please be sure to implement in your equipment using the Products safety measures to guard against the possibility of physical injury, fire or any other damage caused in the event of the failure of any Product, such as derating, redundancy, fire control and fail-safe designs. LAPIS Semiconductor shall bear no responsibility whatsoever for your use of any Product outside of the prescribed scope or not in accordance with the instruction manual.

The Products are not designed or manufactured to be used with any equipment, device or system which requires an extremely high level of reliability the failure or malfunction of which may result in a direct threat to human life or create a risk of human injury (such as a medical instrument, transportation equipment, aerospace machinery, nuclear-reactor controller, fuel-controller or other safety device). LAPIS Semiconductor shall bear no responsibility in any way for use of any of the Products for the above special purposes. If a Product is intended to be used for any such special purpose, please contact a ROHM sales representative before purchasing.

If you intend to export or ship overseas any Product or technology specified herein that may be controlled under the Foreign Exchange and the Foreign Trade Law, you will be required to obtain a license or permit under the Law.

Windows is a registered trademark of Microsoft Corporation (USA) in USA and other countries, and other product names and company names are trademarks or registered trademarks.

Contents

Contents

Introduction

Organization of this Manual

This manual documents the RTLU8 Runtime Library. It assumes that the reader is an experienced C programmer and that the user has a good understanding of the architecture of the nX-U8 as the CPU core.

This manual consists of the following chapters.

Chapter 1. Overview

Chapter 1 presents an overview of the RTLU8 Runtime Library and describes the RTLU8 file structure, actual use of the library, the difference between macros and functions, and the basic functions of the library routines.

Chapter 2. Library Functions by Data Model

Chapter 2 documents, in detail, the library functions by data model provided by the library. These routines are RTLU8 original.

Chapter 3. Standard Embedded Routine Reference

Chapter 3 documents, in detail, the standard embedded routines provided by the library. These routines are presented in alphabetical order.

Chapter 4. Standard I/O Routine Reference

Chapter 4 documents, in detail, the library routines that handle standard I/O. These routines are presented in alphabetical order.

Chapter 5. Low-Level Functions

Chapter 5 documents, in detail, the low-level routines (read and write).

Appendix

Appendix presents the standard library routines and the prototypes of the versions of those routines that correspond to the different memory models.

Related Documents

Refer to the following documents as necessary.

CCU8 User's Manual

This manual documents the use of the CCU8 C compiler and the language specifications.

MACU8 Assembler Package User's Manual

This manual documents the use of the software included in the MACU8 assembler package and the specifications of the assembly language.

Notational Conventions

This manual uses a variety of symbols and notations to simplify the descriptions. The table below presents these conventions.

Symbol	Meaning
SAMPLE	This font, a fixed-width font, is used for messages displayed on the screen, sample command input, examples of created source files, and other text.
itarics	Items shown in italics indicate not text that must be input as shown, but rather items that must be replaced with the required information.
[]	Items shown in square brackets are optional items that are included as needed. These items may be omitted.
…	An ellipsis indicates that the preceding item may be repeated.
{*choice1*|*choice2*}	Indicates that one of the items enclosed in curly braces ({}) and delimited by vertical bars (|) is to be selected and inserted. As long as an item is not enclosed in square brackets, at least one of that item must be included.
value1 - *value2*	Indicates a value between *value1* and *value2*.
PROGRAM . . . PROGRAM	Vertically aligned dots indicate that a section of a program has been omitted.

The table below describes the technical terms used throughout this manual.

Term	Meaning
Macro	A macro is a name defined by a #define preprocessor directive. In this document, "macro" is used to refer to both macros that take parameters and simple macros.
Routine	The term routine is used to refer collectively to both functions and macros that take parameters.
Library routine	Refers to the "routines" included in RTLU8.
Type	A name defined using typedef.
Constant macro	A macro that takes no parameters and always gives the same value. These are also referred to as simply "constants".
Null character	The character with the ASCII code 0x00, that is, the character '\0'.
Null string	A character string of length zero, that is, a character string whose first character is the null character.
Null terminator	The null character that ends a character string.
Null pointer	A pointer to address 0. Expressed by the constant macro NULL.

1. Overview

1.1 What is the RTLU8 Runtime Library?

RTLU8 is a C runtime library for microcontrollers that use the nX-U8 as the CPU core. RTLU8 provides a large number of routines that can be used when programming in C. Much effort and time can be saved by using these library routines.

1.2 Structure of the RTLU8 Runtime Library

This section describes the files that make up the RTLU8 Runtime Library.

The RTLU8 Runtime Library consists of 11 header files and several library files.

1.2.1 Header files

The RTLU8 Runtime Library provides 11 header files classified by function. The header files include function prototypes, macro definitions, and type definitions.

The header files are required when compiling programs. The CCU8 compiler includes header files specified with the #include directive in programs during compilation.

The table below lists the header files and their content.

Header file	Content
ctype.h	Character classification and conversion
error.h	Error identifiers
float.h	Numerical limits for floating-point numbers
limits.h	Numerical limits for integers
math.h	Mathematical functions
setjmp.h	Global jump (longjmp)
stdarg.h	Variable numbers of arguments
stddef.h	Standard types and macros
stdio.h	I/O-related processing
stdlib.h	General-purpose utilities
string.h	Character string manipulation routines

1.2.2 Library files

The library routines are included in the library files. The format of the library files is a binary format that is the same as that of the object files output by the RASU8 and RLU8.

The library files are required at link time. The RLU8 searches in the library file for the library routines used in the program, links the program, and creates an absolute object file (.ABS).

The following library files are provided for use wit the nX-U8.

Library file	Contents
LU8100SW.LIB	RTLU8 runtime library for small memory model
LU8100LW.LIB	RTLU8 runtime library for large memory model

At link time, always specify the same memory model used when the program was compiled with the CCU8 compiler.

1.3 Compatibility with the ANSI/ISO 9899 C Standard

RTLU8 is, basically, a subset of the library specifications stipulated by the ANSI/ISO 9899 C standard.

The following standard headers are not included in RTLU8.

Header file	Use
assert.h	Execution time condition checking.
locale.h	Regional settings and changes
signal.h	Signal handling functions
time.h	Date and time processing functions

The functions, macros, constant macros, type names, and their interfaces and functionality all conform to the ANSI/ISO 9899 C standard.

RTLU8 also provides several functions that are not stipulated in the ANSI/ISO 9899 C standard. These are provided to support the data models (the NEAR and FAR models) that are a unique feature of the CCU8 architecture. See Chapter 2, Library Functions Classified by Data Model, for details.

1.4 Using the Library Routines

This section describe the procedures for setting up the environment to use RTLU8, programming with these library routines, compilation, and linking.

1.4.1 Setting the INCLU8 environment variable

Set the INCLU8 environment variable to inform CCU8 of the path where the header files reside. CCU8 searches for header files specified with the #include preprocessor directive in source files on the path specified in the INCLU8 environment variable.

Use the DOS SET command to set the INCLU8 environment variable. The SET command has the following syntax

```
SET INCLU8=path
```

Example: If the header files are stored in the C:\U8\INCLUDE directory, use the following SET command.

```
SET INCLU8=C:\U8\INCLUDE
```

Notes

The header file path can also be specified with the CCU8 /I*path* option. For example, to compile FOO.C using the path specified in the example above using the /I option, use the following command line.

```
CCU8 /TM610001 /IC:\U8\INCLUDE FOO.C
```

1.4.2 Programming

When using these library routines, the corresponding header file for each routine used must be included. Use an #include preprocessor directive to include each required header file. CCU8 inserts the header files specified with the #include preprocessor directive in the source file. See the library reference starting at chapter 3 to determine which header file each library routine requires.

Example 1:

This example applies when using the memcpy function. The header corresponding to the memcpy function is the string.h header. Therefore, the following line must be present in the source file.

```
#include <string.h>
```

Example 2:

There are no constraints on the order of inclusion when multiple header files are included. Either this order:

```
#include <string.h>
#include <math.h>
```

or this order:

```
#include <math.h>
#include <string.h>
```

may be used when both the string.h and math.h header files are required.

There are two notations for specifying the file name in the #include preprocessor directive: using angle brackets (<>) as shown above, or using double quotes ("") to delimit the file name. Always use angle brackets (<>) to include RTLU8 header files. See the CCU8 User's Manual for details on the #include preprocessor directive.

1.4.3 Procedures from Compilation through Linking

This section presents the procedures for compilation through linking of source files that use these library routines.

1.4.3.1 Compilation and assembly

There is no need to be aware that library routines are being used when compiling and assembling.

Example:

Use the following commands to compile and assemble the source file foo.c.

```
CCU8 /TM610001 FOO.C
RASU8 FOO.ASM
```

The /SD option must be specified in both the CCU8 and RASU8 command lines to create an object file that includes C source level debugging information.

1.4.3.2 Linking with the library

After the compilation and assembly operations, the RLU8 linker is used to create an absolute object file. At this time, in addition to the object file created by the compilation and assembly operations, a startup routine and the runtime library must be specified.

Example 1:

Use the following command line to link the file FOO.OBJ.

```
RLU8 FOO C:\U8\STARTUP\S610001SW,,, C:\U8\LIB\LU8100SW.LIB /CC
```

In this example, the startup routine S610001SW.OBJ is stored in C:\U8\STARTUP, and the runtime library LU8100SW.LIB is stored in C:\U8\LIB.

If the library file is located on the path specified by the LIBU8 environment variable, the path for the library file may be omitted.

Example 2:

The RLU8 command line will have the following form if LU8100SW.LIB is stored in C:\U8\LIB and the LIBU8 environment variable is set to C:\U8\LIB.

```
RLU8 FOO C:\U8\STARTUP\S610001SW,,, LU8100SW.LIB /CC
```

1.5 Role of the Header Files

The header files function as the interface between the library and the user program. Inclusion of the header file corresponding to the library routine tells the compiler the type of (more precisely, it gives the function prototype for) the library routine and the constants and types used by that routine.

1.5.1 Including macros, constants, and types

Inclusion of the header file is required to define the macros, constants, and types included in the library.

The header files include macros, constants, and types used by the library routines. Programs can also use these. The content of these definitions as used by the library routines must be exactly the same as the content of these definitions as used by user programs.

In most cases, the programmer only has to know the meaning of the macros, constants, and types included in header files, and does not need to know the details of the actual definitions.

Example:

This example shows the use of a variable-argument list. Since the macros va_start, va_arg, and va_end, as well as the type va_list, are defined in stdarg.h, this header file is included. The programmer does not need to know the details of these definitions.

```
#include <stdarg.h>

int func(int num , ...)
{
     int i;
     int total;
     va_list arg;

     va_start(arg , num);
     total = 0;
     for(i = 0 ; i < num ; ++i)
     {
          total += va_arg(arg , int);
     }
     va_end(arg);
     return total;
}
```

1.5.2 Including function prototype definitions

The calling formats, that is the types of each of the arguments and the return value, of all the functions in the library are declared in the header files. These declarations are usually called function prototypes. The compiler verifies that the format, that is, the number of arguments, the types of those arguments, and the type of the return value, of the calls to library functions used in the user program match the function prototype in the header file. If a call does not match, the compiler issues a warning or error.

This type checking by the compiler is extremely important for program safety, since, unlike errors in algorithms, problems due to function calling format mismatches are very hard to find.

Example:

Example of use of the strlen function.

```
#include <string.h>

int i;

int func(void)
{
     int len;
     .
     .
     .
     len = strlen(i);         /* Warning */
     .
     .
     .
}
```

The prototype of the strlen function in the string.h header file is as follows.

```
     size_t strlen(char *);
```

Since the variable i, which has type int, is specified as the argument in the call to the strlen function, the compiler issues a warning regarding this call.

The compiler is able to perform this check because string.h is included at the start of the file. If string.h were not included, the compiler would not perform this check.

1.6 Functions and Macros

1.6.1 The differences between functions and macros

The term "library routine" as used in this document actually refers to both functions and macros that takes parameters. Each of the library routines included in the RTLU8 is either a function, a macro, or both. Section 1.8, "Contents of the Header Files", and the library reference starting at chapter 3 indicate explicitly how each library routine is implemented.

Normally, there is no need to be concerned whether a routine is implemented as a function or a macro. However, the programmer needs to be concerned with the differences between functions and macros in the following cases.

(1) While function calls are translated into calls to subroutines by the compiler, macro calls are expanded into inline C code by the preprocessor. This means that a macro call is faster than a function call by the amount of the overhead involved in a function call. However, since each time a macro is expanded, the same code gets inserted in the program, the size of the resultant program will be larger than if a function had been used.

(2) At compile time, a function name has meaning as an address, but since macros are expanded at preprocessing time, they have already disappeared from the code by compile time. This means that macros cannot be used through a function pointer.

(3) Although the compiler applies type checking to function calls, it does not apply a type check to macro calls. This means that the programmer must take responsibility for the types of the arguments and the return values for macro calls.

1.6.2 Method for calling functions overridden as macros

Several of the library routines included in RTLU8 are provided as both functions and macros. The function toupper in ctype.h is one example. Routines of this type are indicated as "macro/function" in both section 1.8, "Contents of the Header Files", and in the library reference starting at chapter 3.

Routines of this type are first defined as function prototypes in the header file, and then, after that, are defined as macros. This means that they normally are used as macros. This section presents two methods for using this type of routine as a function.

1.6.2.1 Remove the macro definition with #undef

The first method is to use the #undef preprocessor directive to remove the definition of the routine name as a macro. Insert the #undef preprocessor directive between the #include preprocessor directive that includes the header file and the first use of the routine in your program. The safest place for this directive is immediately following the #include preprocessor directive for the corresponding header file.

Example:

This example removes the toupper macro with the #undef preprocessor directive.

```
#include <ctype.h>
#undef toupper      /* Remove the macro definition. */

void func(void)
{
     int c;
     .
     .
     .
     c = toupper(c) ;    /* Here, toupper is called as a function. */
     .
     .
     .
}
```

1.6.2.2 Enclose the routine name in parentheses

The second method is to enclose the routine name in parentheses when calling that library routine. The preprocessor verifies that there is an open (left) parenthesis immediately following the name of a macro that takes parameters. Therefore, enclosing the routine name in parentheses prevents the preprocessor from expanding the macro.

Example:

This example calls the toupper function by enclosing toupper in parentheses.

```
#include <ctype.h>

void func(void)
{
     int  c;
     .
     .
     .
     c = (toupper)(c) ;  /* Here, toupper is called as a function. */
     .
     .
     .
}
```

1.7 Contents of the Header Files

This section describes the functions, macros, global variables, constant macros, and types included in RTLU8, organized by header file.

The "Type" column entries indicate one of the following.

Function

Macro

Macro/function

Constant macro

Type

Global variable

The term "Macro/function" refers to library routines provided as both a macro and a function. See the library reference starting at chapter 3 for details on the functions, macros, and macro/function routines.

1.7.1 Character classification and conversion: <ctype.h>

The header ctype.h declares routines that classify and convert 1-byte character types.

Name	Type	Description
isalnum	Macro/function	Tests whether a character is an alphanumeric character.
isalpha	Macro/function	Tests whether a character is an alphabetic character.
iscntrl	Macro/function	Tests whether a character is a control character (0x00 to 0x1F, and 0x7F).
isdigit	Macro/function	Tests whether a character is a digit.
isgraph	Macro/function	Tests whether a character is a printing character (0x21 to 0x7E) other than space (' ').
islower	Macro/function	Tests whether a character is a lowercase character.
isprint	Macro/function	Tests whether a character is a printing character (0x20 to 0x7E) including space (' ').
ispunct	Macro/function	Tests whether a character is a punctuation character (0x21 to 0x2F, 0x3A to 0x40, 0x5B to 0x60, and 0x7B to 0x7E).
isspace	Macro/function	Tests whether a character is a space character (0x09 to 0x0D and ' ').
isupper	Macro/function	Tests whether a character is an uppercase character.
isxdigit	Macro/function	Tests whether a character is a hexadecimal digit.

Name	Type	Description
tolower	Macro/function	Converts an uppercase character to the corresponding lowercase character.
toupper	Macro/function	Converts an lowercase character to the corresponding uppercase character.

1.7.2 Error identification <errno.h>

The header errno.h includes information related to errors generated in library routines. The global variable errno and constant macros for the values to which it may be set are defined in errno.h.

Name	Type	Description
errno	Global variable	A global variable of type volatile int that holds the error status. The initial value is 0, and when an error occurs in a library routine, it is set to one of the following non-zero values according to the error status.
EDOM	Constant macro	Constant that indicates a domain error. A domain error occurs when an attempt is made to calculate a mathematical function of a value that is outside its domain, for example, if a value larger than 1 or smaller than -1 is passed to the asin function.
ERANGE	Constant macro	Constant that indicates that an overflow occurred. Overflows occur when the result of a mathematical function exceeds the range of values that can be expressed by an numbers of type double.

1.7.3 Limit values for floating-point numbers <float.h>

The header float.h declares constant macros that express the limit values for floating-point numbers of type float, double, and long double. Since long double is the same as double in CCU8, the limit values for type long double are the same as those for type double.

Name	Type	Description
DBL_DIG	Constant macro	The number of decimal digits that can be represented by type double.
DBL_EPSILON	Constant macro	For the type double, the smallest positive floating-point number such that 1.0 and 1.0 + DBL_EPSILON have recognizably different values.
DBL_MANT_DIG	Constant macro	The number of bits in the mantissa of numbers of type double.
DBL_MAX	Constant macro	The maximum value that can be expressed by type double.
DBL_MAX_EXP	Constant macro	One more than the exponent of the maximum value that can be expressed by type double in binary.

Name	Type	Description
DBL_MAX_10_EXP	Constant macro	The exponent of the maximum value that can be expressed by type double in decimal.
DBL_MIN	Constant macro	The smallest value that can be expressed by type double.
DBL_MIN_EXP	Constant macro	One more than the exponent of the minimum value that can be expressed by type double in binary.
DBL_MIN_10_EXP	Constant macro	The exponent of the minimum value that can be expressed by type double in decimal.
FLT_DIG	Constant macro	The number of decimal digits that can be represented by type float.
FLT_EPSILON	Constant macro	For the type float, the smallest positive floating-point number such that 1.0 and 1.0 + FLT_EPSILON have recognizably different values.
FLT_MANT_DIG	Constant macro	The number of bits in the mantissa of numbers of type float.
FLT_MAX	Constant macro	The maximum value that can be expressed by type float.
FLT_MAX_EXP	Constant macro	One more than the exponent of the maximum value that can be expressed by type float in binary.
FLT_MAX_10_EXP	Constant macro	The exponent of the maximum value that can be expressed by type float in decimal.
FLT_MIN	Constant macro	The smallest value that can be expressed by type float.
FLT_MIN_EXP	Constant macro	One more than the exponent of the minimum value that can be expressed by type float in binary.
FLT_MIN_10_EXP	Constant macro	The exponent of the minimum value that can be expressed by type float in decimal.
FLT_RADIX	Constant macro	The floor of the radix used for floating-point numbers.
FLT_ROUNDS	Constant macro	Indicates that rounding is performed.
LDBL_DIG	Constant macro	The same as DBL_DIG
LDBL_EPSILON	Constant macro	The same as DBL_EPSILON
LDBL_MANT_DIG	Constant macro	The same as DBL_MANT_DIG
LDBL_MAX	Constant macro	The same as DBL_MAX
LDBL_MAX_EXP	Constant macro	The same as DBL_MAX_EXP
LDBL_MAX_10_EXP	Constant macro	The same as DBL_MAX_10_EXP
LDBL_MIN	Constant macro	The same as DBL_MIN_10_EXP
LDBL_MIN_EXP	Constant macro	The same as DBL_MIN_EXP
LDBL_MIN_10_EXP	Constant macro	The same as DBL_MIN_10_EXP

1.7.4 Limit values for integers <limits.h>

The header limits.h declares constant macros that express the limit values for the integer types.

Name	Type	Description
CHAR_BIT	Constant macro	8 The number of bits in the type char.
CHAR_MAX	Constant macro	127 The maximum value of the type char.
CHAR_MIN	Constant macro	-128 The minimum value of the type char.
INT_MAX	Constant macro	32767 The maximum value of the type int.
INT_MIN	Constant macro	-32768 The minimum value of the type int.
LONG_MAX	Constant macro	2147483647 The maximum value of the type long int.
LONG_MIN	Constant macro	-2147483648 The minimum value of the type long int.
SCHAR_MAX	Constant macro	127 The maximum value of the type signed char.
SCHAR_MIN	Constant macro	-128 The minimum value of the type signed char.
SHRT_MAX	Constant macro	32767 The maximum value of the type short short int.
SHRT_MIN	Constant macro	-32768 The minimum value of the type short short int.
UCHAR_MAX	Constant macro	255 The maximum value of the type unsigned char.
UINT_MAX	Constant macro	65535 The maximum value of the type unsigned int.
ULONG_MAX	Constant macro	4294967295 The maximum value of the type unsigned long int.
USHRT_MAX	Constant macro	65535 The maximum value of the type unsigned short int.

1.7.5 Mathematical functions <math.h>

The header math.h declares the mathematical functions. The calculations are all implemented using the type double. Some of the mathematical functions store an error value in the global variable errno if an error occurs. See the documentation on the individual functions in chapter 3, Standard Embedded Routine Reference.

Name	Type	Description
HUGE_VAL	Constant macro	The largest value that can be represented by the type double. This constant expresses the value infinity.
exp	Function	Calculates the exponential.
frexp	Function	Splits a floating-point number into exponent and mantissa.
frexp_n		
frexp_f		
ldexp	Function	Calculates the product of the first argument and 2 raised to the power of the second argument.
log	Macro/function	Calculates the natural logarithm.
log10	Macro/function	Calculates the common logarithm.
modf	Function	Splits a floating-point number into its integer and fractional parts.
modf_n		
modf_f		
cosh	Function	Calculates the hyperbolic cosine.
sinh	Function	Calculates the hyperbolic sine.
tanh	Function	Calculates the hyperbolic tangent.
ceil	Function	Calculates the ceiling of a floating point number (next higher integer).
fabs	Function	Calculates the absolute value of floating-point number.
floor	Function	Calculates the floor of a floating point number (next lower integer).
fmod	Function	Determines the floating-point remainder.
pow	Function	Calculates a value raised to a power.
sqrt	Function	Calculates the square root.
acos	Macro/function	Calculates the arccosine.
asin	Macro/function	Calculates the arcsine.
atan	Function	Calculates the arctangent.
atan2	Function	Calculates the arctangent of the quotient of its arguments. This function can calculate the arctangent of values that would be too large for atan to handle.

Name	Type	Description
cos	Macro/function	Calculates the cosine.
sin	Macro/function	Calculates the sine.
tan	Function	Calculates the tangent.

1.7.6 Global jump <setjmp.h>

The setjmp.h header declares functions used to implement nonlocal jumps and includes related macro and type definitions. These functions allow jumps to locations outside the current function.

Name	Type	Description
jmp_buf jmp_buf_n jmp_buf_f	Type	Global jumps are implemented by using setjmp to save the current environment and using longjmp to restore that environment. The type jmp_buf is the type used to represent a saved environment.
setjmp setjmp_n setjmp_f	Macro	Saves the environment in its argument, which is of type jmp_buf.
longjmp longjmp_n longjmp_f	Function	Restores an environment saved by setjmp. As a result, program execution transfers to the point where setjmp was called.

1.7.7 Variable arguments <stdarg.h>

The header stdarg.h includes definitions and declarations that implement variable argument lists for functions. Use of these definitions and declarations allows the creations of functions that have variable length argument lists, without concern for the way the compiler handles arguments at the assembler language level.

Name	Type	Description
va_list	Type	Data type used to hold information related to the variable argument list. This data type is used by the va_start, va_arg, and va_end routines.
va_start	Macro	Prepares for referencing a variable argument list. This routine must be called before using va_arg.
va_arg	Macro	Returns the next argument in the variable argument list. Use of the va_arg macro allows the function to reference the second and later arguments in order.
va_end	Macro	Terminates reference (use) of a variable argument list.

1.7.8 Common types and macros <stddef.h>

The header stddef.h defines certain commonly used data types and macros.

Name	Type	Description
NULL	Constant macro	Expresses the null pointer constant.
offsetof	Macro	Gives the number of bytes from the start of a structure of a member of that structure.
ptrdiff_t	Type	Signed integer type that represents values that correspond to the difference between two pointers.
size_t	Type	Unsigned integer type that represents the results of the sizeof operator.

1.7.9 I/O processing <stdio.h>

The header stdio.h declares routines that perform I/O processing and defines macros and types that are used by those routines.

Name	Type	Description
EOF	Constant macro	-1 Indicates the end of a file. This value is also used as the return value when an error occurs.
FILE	Type	Type used to represent streams.
stderr	Macro	Pointer to the standard error stream.
stdin	Macro	Pointer to the standard input stream.
stdout	Macro	Pointer to the standard output stream.
fflush	Function	Flushes a stream.
fgetc	Function	Acquires a character from a stream.
fgets	Function	Acquires a character string from a stream.
fgets_n		
fgets_f		
fprintf	Function	Performs formatted output to a stream.
fprintf_n		
fprintf_f		
fputc	Function	Outputs one character to a stream.
fputs	Function	Outputs a character string to a stream.
fputs_n		

Name	Type	Description
fputs_f		
fread	Function	Reads data from a stream.
fread_n		
fread_f		
fscanf	Function	Scans and formats input from a stream.
fscanf_n		
fscanf_f		
fwrite	Function	Writes data to a stream.
fwrite_n		
fwrite_f		
getc	Macro/function	Acquires a single character from a stream.
getchar	Macro/function	Acquires a single character from standard input.
gets	Function	Acquires a character string from standard input.
gets_n		
gets_f		
printf	Function	Writes formatted output to standard output.
printf_n		
printf_f		
putc	Macro/function	Outputs a single character to a stream.
putchar	Macro/function	Outputs a character to standard output.
puts	Function	Outputs a character string to standard output.
puts_n		
puts_f		
scanf	Function	Scans and formats input from the standard input stream.
scanf_n		
scanf_f		
sprintf	Function	Writes formatted data to a character string.
sprintf_nn		
sprintf_nf		
sprintf_fn		

Name	Type	Description
sprintf_ff		
sscanf	Function	Reads formatted data from a character string.
sscanf_nn		
sscanf_nf		
sscanf_fn		
sscanf_ff		
ungetc	Function	Pushes one character back onto an input stream.
vprintf	Function	Writes formatted output.
vprintf_n		
vprintf_f		
vsprintf	Function	Writes formatted data to a character string.
vsprintf_nn		
vsprintf_nf		
vsprintf_fn		
vsprintf_ff		

1.7.10 General utilities <stdlib.h>

The header stdlib.h declares several routines that are of general utility and defines macros and types that are used by those routines.

Name	Type	Description
div_t	Type	Type of the result returned by the div function. This is a structure with two members, quot and rem, of type int.
ldiv_t	Type	Type of the result returned by the ldiv function. This is a structure with two members, quot and rem, of type long.
RAND_MAX	Constant macro	32767 The maximum value of the pseudorandom number returned by the function rand.
abs	Function	Calculates the absolute value of an integer of type int.
atof	Macro/function	Converts a character string to a floating-point number of type double.
atof_n		
atof_f		

Name	Type	Description
atoi	Macro/function	Converts a character string to an integer of type int.
atoi_n		
atoi_f		
atol	Macro/function	Converts a character string to an integer of type long.
atol_n		
atol_f		
bsearch	Function	Performs a binary search for the specified item in a sorted array.
bsearch_nn		
bsearch_nf		
bsearch_fn		
bsearch_ff		
calloc	Function	Allocates the required amount of memory.
calloc_n		
calloc_f		
div	Function	Calculates the quotient and remainder of two values of type int. Stores the calculated quotient and remainder in a structure of type div_t and returns that structure.
free	Function	Releases memory.
free_n		
free_f		
labs	Function	Calculates the absolute value of a value of type long.
ldiv	Function	Calculates the quotient and remainder of two values of type long. Stores the calculated quotient and remainder in a structure of type ldiv_t and returns that structure.
malloc	Function	Allocates memory.
malloc_n		
malloc_f		
qsort	Function	Sorts the elements in an array using the quicksort algorithm.
qsort_n		
qsort_f		
rand	Function	Generates a pseudorandom number.

Name	Type	Description
realloc	Function	Reallocates memory.
realloc_n	Function	Reallocates memory.
realloc_f		
srand	Macro/function	Initializes the sequence of pseudorandom numbers generated by rand.
strtod	Macro/function	Converts a character string to a floating-point number of type double.
strtod_n		
strtod_f		
strtol	Function	Converts a character string to a value of type long.
strtol_n		
strtol_f		
strtoul	Macro/function	Converts a character string to a value of type unsigned long.
strtoul_n		
strtoul_f		

1.7.11 String handling <string.h>

The header string.h declares functions that manipulate strings and memory areas.

Name	Type	Description
memchr	Function	Searches for the first occurrence of a given byte of data in a memory area.
memchr_n		
memchr_f		
memcmp	Function	Compares two memory areas.
memcmp_nn		
memcmp_nf		
memcmp_fn		
memcmp_ff		
memcpy	Function	Copies the data in one memory area to another memory area.
memcpy_nn		
memcpy_nf		
memcpy_fn		
memcpy_ff		

Name	Type	Description
memmove	Function	Copies the data in one memory area to another memory area. Unlike memcpy, memmove operates correctly even if the memory areas overlap.
memmove_nn		
memmove_nf		
memmove_fn		
memmove_ff		
memset	Function	Fills a fixed memory area with the specified data byte.
memset_n		
memset_f		
strcat	Function	Concatenates character strings.
strcat_nn		
strcat_nf		
strcat_fn		
strcat_ff		
strchr	Function	Searches for the first occurrence of a character in a character string.
strchr_n		
strchr_f		
strcmp	Function	Compares two character strings.
strcmp_nn		
strcmp_nf		
strcmp_fn		
strcmp_ff		
strcpy	Function	Copies character strings.
strcpy_nn		
strcpy_nf		
strcpy_fn		
strcpy_ff		
strcspn	Function	Returns the length of the section at the head of the first character string that does not include the other character string.
strcspn_nn		
strcspn_nf		
strcspn_fn		

Name	Type	Description
strcspn_ff		
strlen	Function	Returns the length of a character string.
strlen_n		
strlen_f		
strncat	Function	Concatenates the first n bytes from the start of a character string to the end of the other character string.
strncat_nn		
strncat_nf		
strncat_fn		
strncat_ff		
strncmp	Function	Compares the first n bytes of two character strings.
strncmp_nn		
strncmp_nf		
strncmp_fn		
strncmp_ff		
strncpy	Function	Copies the first n bytes of a character string into the other memory area.
strncpy_nn		
strncpy_nf		
strncpy_fn		
strncpy_ff		
strpbrk	Function	Finds the first place any one of the characters included in the first character string appears in the other character string.
strpbrk_nn		
strpbrk_nf		
strpbrk_fn		
strpbrk_ff		
strrchr	Function	Searches for the last place a given character appears in a character string.
strrchr_n		
strrchr_f		
strspn	Function	Returns the length of the section that occurs first in the first character string that only consists of characters included in the second character string.
strspn_nn		
strspn_nf		

Name	Type	Description
strspn_fn		
strspn_ff		
strstr	Function	Searches for the second character string in the first character string.
strstr_nn		
strstr_nf		
strstr_fn		
strstr_ff		
strtok	Function	Divides a character string into tokens.
strtok_nn		
strtok_nf		
strtok_fn		
strtok_ff		

1.8 Using the Runtime Library Reference

The sections of this document starting from chapter 3 document the routines provided by the RTLU8 runtime library in alphabetical order.

The documentation for the routines has the following format.

Routine name

The name of the routine is shown at the upper left of the page.

Type

The type of the routine (function, macro, or macro/function) is shown at the top right of the page.

Function

Presents a brief explanation of the function of the routine.

Format

Lists the header file that includes the routine declaration or definition and shows the prototype for the routine. Also documents the meanings of the arguments to the routine.

Description

Documents the function of the routine in detail.

Return value

Describes the value returned by the routine.

See also

Lists related routines.

Sample program

Presents programs that actually use the routine. The purpose of this section is limited to showing the function of the routine as it would be used in an actual program. These programs are not necessarily practical programs.

2. Library Functions by Data Model

2.1 Library Routines Corresponding to Data Access Models

The CCU8 compiler supports two data models, the FAR model and the NEAR model, the latter of which is specified as the default access model. Therefore, the RTLU8 runtime library adds unique library functions corresponding to each of these data access models. Modifiers (__near and __far) that determine the data access are specified for these library functions. The __near modifier is used to access NEAR data, and the __far modifier is used to access FAR data. The library function corresponding to these data access types is provided by directly including these modifiers in the source code.

See the CCU8 Language Reference for details on use of the __near and __far modifiers.

2.2 Using the Library Functions

This section describes how to use library functions that correspond to the data access type.

2.2.1 Use of normal library functions

This section presents the use of normal library functions.

Example:

Here we consider functions that take pointer arguments, such as strcpy(char **string1*, const char **string2*), when the __near and __far modifiers are not specified for the arguments.

```
char ndata[128];
const char fdata[] = ”sample”;

void func()
{
     strcpy(ndata, fdata);
}
```

In this case, since the __near and __far modifiers are not specified for the arguments to strcpy, the data model used is limited to the data model determined by the CCU8 command line option used, namely, /near or /far. Since data access types are not specified (for example, in cases where you need to copy from FAR data to NEAR data), it will not be possible to achieve such operations with the notation used above.

2.2.2 Use of library functions corresponding to data models

To support these two data models, RTLU8 provides libraries that correspond to the ANSI/ISO 9899 C standard library routines that take pointer arguments. The following example shows the use of library routines that correspond to the data model.

Example:

This example shows the correct use of the strcpy function, a use that requires caution, and an incorrect cast. This example presumes that the /near option was specified on the CCU8 command line.

```
#include <string.h>

const char __near nstr[] = "near string";
const char __far fstr[] = "far string";
char __near nbuf[20];
char __far fbuf[20];
char op_buf[20];

void func(void)
{
     .
     .
     .
     /* Correct usage examples */
     strcpy(nbuf, nstr);
     strcpy_nn(nbuf, nstr);
     strcpy_nf(nbuf, fstr);
     strcpy_fn(fbuf, nstr);
     strcpy_ff(fbuf, fstr);

     strcpy_nf(nbuf, (char __far*)nstr);
     strcpy_fn((char __far *)nbuf, nstr);
     strcpy_ff((char __far *)nbuf, (char __far *)nstr);
     .
     .
     .
     /* A usage that requires caution */
     strcpy_nf(op_buf, fstr);      /* Pointer types for which no
                                      modifier is specified are
                                      determined to be NEAR model
                                      pointers due to the use of
                                      the /near option. */
     .
     .
     .
     /* Incorrect cast */
     strcpy_nn(nbuf, (char __near *)fstr);
     .
     .
     .
}
```

The cast at the end of the example is particularly dangerous. Since this usage is not a syntax error under CCU8, the CCU8 compiler will not flag a source code error for this statement. However, since a FAR pointer is, in actuality, being cast to a NEAR pointer, the FAR pointer physical segment will be ignored, and the program may function incorrectly. In this case the statement:

```
strcpy_nf(nbuf, fstr);
```

must be used. Note that to cast a NEAR pointer to a FAR pointer, all that needs to be done is to add #0 as the FAR pointer physical segment.

See the appendix at the end of this document for the library routines corresponding to the data models.

2.3 Rules for Routine Names for Corresponding Data Models

The rules for forming the RTLU8-provided routine names that correspond to the data models from the ANSI/ISO 9899 C standard routine names are as follows.

Routines that have a suffix starting with an underscore (_) after an ANSI/ISO 9899 C standard routine name take pointers to the corresponding data model (NEAR or FAR) as arguments. The table below lists these suffixes and their meaning.

		Corresponding data access	
Suffix	**Number of pointer arguments**	**First pointer argument**	**Second pointer argument**
_n	1	NEAR	
_f	1	FAR	
_nn	2	NEAR	NEAR
_nf	2	NEAR	FAR
_fn	2	FAR	NEAR
_ff	2	FAR	FAR

The following are exceptions to the above.

(1) The pointer to the FILE structure defined for standard I/O must be a NEAR pointer.

(2) Functions that take a variable number of arguments (such as printf and scanf) take their other arguments in the same data model as the format character string. Therefore, these are not included in the number of pointer arguments.

(3) The strtod, strtol, and strtoul functions take two pointer arguments. However, since these are rarely used functions, the _nf and _fn versions are not provided. For these functions, the library provides an _n version, which takes all its pointer arguments as NEAR pointers, and an _f version, which takes all its pointer arguments as FAR pointers are provided.

Here we present examples of actual library functions that correspond to the data models.

For example, the following versions of the atol function are provided.

Function	Pointer type
atol_n(s)	The pointer s is a NEAR pointer.
atol_f(s)	The pointer s is a FAR pointer.

The following versions of the strcpy function are provided.

Function	Pointer type
strcpy_nn(s1 , s2)	Here, s1 is a NEAR pointer and s2 is a NEAR pointer.
strcpy_nf(s1 , s2)	Here, s1 is a NEAR pointer and s2 is a FAR pointer.
strcpy_fn(s1 , s2)	Here, s1 is a FAR pointer and s2 is a NEAR pointer.
strcpy_ff(s1 , s2)	Here, s1 is a FAR pointer and s2 is a FAR pointer.

3. Standard Embedded Routine Reference

abs Function

Function

Calculates the absolute value of a numerical value of type int.

Format

#include <stdlib.h>

int abs(int *n*);

n Integer value

Description

The abs function calculates the absolute value of the integer *n*.

Return value

Returns a value in the range 0 to 32767. However, if *n* is -32768, abs returns -32768.

See also

fabs labs

Sample program

```
#include <stdlib.h>

void main(void)
{
int n,res;

n = -1234;
res = abs(n);
}
```

acos — Macro/function

Function

Calculates the arccosine (the inverse cosine function).

Format

#include <math.h>

double acos(double *x*);

x Real number whose arccosine is to be calculated

Description

The acos function calculates the arccosine of its argument *x*. The value of the argument *x* must lie in the range -1 to 1. If acos is called with a value outside that range, a domain error occurs and the global variable errno is set to EDOM.

Return value

Returns the arccosine of *x* in the range 0 to π radians.

See also

asin atan atan2 cos sin tan

Sample program

```
#include <math.h>

void main(void)
{
double x;
double res;

x = 0.5;

res = acos(x);
}
```

asin

Macro/function

Function

Calculates the arcsine (the inverse sine function).

Format

#include <math.h>

double asin(double *x*);

x Real number whose arcsine is to be calculated

Description

The asin function calculates the arcsine of its argument *x*. The value of the argument *x* must lie in the range -1 to 1. If asin is called with a value outside that range, a domain error occurs and the global variable errno is set to EDOM.

Return value

Returns the arcsine of *x* in the range $-\pi/2$ to $\pi/2$ radians.

See also

acos atan atan2 cos sin tan

Sample program

```
#include <math.h>

void main(void)
{
double x;
double res;

x = 0.5;

res = asin(x);
}
```

atan Function

Function

Calculates the arctangent (the inverse tangent function).

Format

#include <math.h>

double atan(double *x*);

x Real number whose arctangent is to be calculated

Description

The atan function calculates the arctangent of its argument *x*.

Return value

Returns the arctangent of *x* in the range $-\pi/2$ to $\pi/2$ radians.

See also

acos asin atan2 cos sin tan

Sample program

```
#include <math.h>

void main(void)
{
double x;
double res;

x = 0.5;

res = atan(x);
}
```

atan2 Function

Function

Calculates the arctangent of *y*/*x*.

Format

#include <math.h>

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

x, *y* Arbitrary real number values

Description

The atan2 function calculates the arctangent of *y*/*x*. The atan2 function returns correct values even when *x* is 0 or close to 0. The atan2 function returns 0 if both *x* and y are 0.

Return value

Returns the arctangent of *y*/*x* in the range -π to π radians.

See also

acos asin atan cos sin tan

Sample program

```
#include <math.h>

void main(void)
{
double x;
double y;
double res;

x = 2.0;
y = 3.0;

res = atan2(y, x);
}
```

atof

Macro/function

Function

Converts a character string to a floating-point number of type double.

Format

```
#include  <stdlib.h>
double  atof( const char *s );
double  atof_n( const char __near *s );
double  atof_f( const char __far *s )
```

s Character string to be converted

Description

The atof function converts the character string pointed to by the argument *s* to a double-precision floating-point value and returns that value. The atof function is identical to the following function calls.

```
strtod(s, (char *) NULL);
strtod_n(s, (char __near *) NULL);
strtod_f(s, (char __far *) NULL);
```

The character string *s* must have the following format.

[*white space*] [*sign*] [*digit*] [.] [*digit*] [{e|E} [*sign*] *digit*]

The components of the character string have the following interpretations.

Symbol	Meaning
[*white space*]	Tab or space characters (May be omitted.)
[*sign*]	Sign (May be omitted.)
[*digit*] [.] [*digit*]	Character string that expresses the decimal fraction (May be omitted.)
[{e|E} [*sign*] *digit*]	Character string that express the exponent (May be omitted.)

The atof function stops scanning at the point it reads a character it cannot interpret. If the converted value cannot be represented by type double, atof returns HUGE_VAL and sets the global variable errno to ERANGE.

Return value

Returns the converted value of the character string as type double.

See also

atoi atol strtod strtol strtoul

Sample program

```
#include <stdlib.h>

void main(void)
{
double res;

res = atof("1.234e+6");
}
```

atoi Macro/function

Function

Converts a character string to an integer of type int.

Format

#include <stdlib.h>

int atoi(const char *s);

int atoi_n(const char __near *s);

int atoi_f(const char __far *s);

s Character string to be converted

Description

The atoi function converts the character string pointed to by the argument *s* to an integer of type int and returns that value. The atoi function is identical to the following function calls.

```
(int)strtol(s, (char **) NULL, 10);
(int)strtol_n(s, (char __near * __near *) NULL, 10);
(int)strtol_f(s, (char __far * __far *) NULL, 10);
```

The character string *s* must have the following format.

[*white space*] [*sign*] [*digit*]

The components of the character string have the following interpretations.

Symbol	Meaning
[*white space*]	Tab or space characters (May be omitted.)
[*sign*]	Sign (May be omitted.)
[*digit*]	Character string that expresses an integer. (May be omitted.)

The atoi function stops scanning at the point it reads a character it cannot interpret. The result of the atoi function is undefined if an overflow occurs.

Return value

Returns the value of the converted character string as type int.

See also

atof atol strtod strtol strtoul

Sample program

```
#include <stdlib.h>

void main(void)
{
int res;

res = atoi("32767");
}
```

atol

Macro/function

Function

Converts a character string to an integer of type long.

Format

```
#include <stdlib.h>
long atol( const char *s );
long atol_n( const char __near *s );
long atol_f( const char __far *s );
```

s Character string to be converted

Description

The atol function converts the character string pointed to by the argument *s* to an integer of type long and returns that value. The atol function is identical to the following function calls.

```
(long)strtol(s, (char **)NULL, 10);
(long)strtol_n(s, (char __near * __near *)NULL, 10);
(long)strtol_f(s, (char __far * __far *)NULL, 10);
```

The character string *s* must have the following format.

[*white space*] [*sign*] [*digit*]

The components of the character string have the following interpretations.

Symbol	Meaning
[*white space*]	Tab or space characters (May be omitted.)
[*sign*]	Sign (May be omitted.)
[*digit*]	Character string that expresses an integer. (May be omitted.)

The atol function stops scanning at the point it reads a character it cannot interpret. If the converted value is not within the range of values that can be represented by type long, the atol function returns either LONG_MAX or LONG_MIN and sets the global variable errno to ERANGE.

Return value

Returns the value of the converted character string as type long.

See also

atof atoi strtod strtol strtoul

Sample program

```
#include <stdlib.h>

void main(void)
{
long res;

res = atol("-2147483647");
}
```

bsearch **Function**

Function

Performs a binary search for a specified item in a sorted array.

Format

```
#include <stdlib.h>
void *bsearch( const void *key, const void *base, size_t nelem,
                    size_t size, int (*cmp )( const void *, const void *) );
void __near *bsearch_nn( const void __near *key, const void __near *base,
            size_t nelem, size_t size, int (*cmp_nn )( const void __near *, const void __near *) );
void __far *bsearch_nf( const void __near *key, const void __far *base, size_t nelem,
                size_t size, int (*cmp_nf )( const void __near *, const void __far *) );
void __near *bsearch_fn( const void __far *key, const void __near *base, size_t nelem,
                size_t size, int (*cmp_fn )( const void __far *, const void __near *) );
void __far *bsearch_ff( const void __far *key, const void __far *base, size_t nelem,
                size_t size, int (*cmp_ff )( const void __far *, const void __far *) );
```

key	Search key
base	Array to be searched
nelem	Number of elements in the array
size	Size of each element of the array in bytes
cmp, cmp_nn, cmp_nf, cmp_fn, cmp_ff	Pointer to a comparison function

Description

The bsearch function searches for an element that matches *key* in the array *base* that has *nelem* elements, and returns the address of that element. If an element that matches the specified item is not found, bsearch returns NULL. The elements of the array must be sorted in advance.

The function specified by *cmp* is a comparison function written by the user, and must take two arguments of type pointer to void (void *). If the arguments to this function are *elem1* as the first argument and *elem2* as the second argument, then this comparison function must return a value as follows according to the arguments it receives.

Condition	Return value
∗elem1 < ∗elem2	A negative value
∗elem1 == ∗elem2	0
∗elem1 > ∗elem2	A positive value

Return value

Returns a pointer to the element in the array that matched *key*. Returns NULL if there is no matching element.

See also

qsort

Sample program

```
#include <stdlib.h>

char *array[5];
char a[10] = "apple";
char b[10] = "cherry";
char c[10] = "orange";
char d[10] = "peach";
char e[10] = "pear";
char **curr_ptr;

int compare(char *, char **);

void main(void)
{
array[0] = a;
array[1] = b;
array[2] = c;
array[3] = d;
array[4] = e;

curr_ptr =
(char **)bsearch("peach", array, 5, sizeof(char *), compare );
}

int compare( char *ele1, char **ele2)
{
return(strcmp(ele1, *ele2));
}
```

calloc Function

Function

Allocates the required amount of memory.

Format

#include <stdlib.h>

void *calloc(size_t *nelem*, size_t *size*);

void __near *calloc_n(size_t *nelem*, size_t *size*);

void __far *calloc_f(size_t *nelem*, size_t *size*);

nelem Total number of elements

size Size of each element

Description

The calloc function allocates memory with *nelem* × *size* bytes in the dynamic segment. It initializes all of the contents of the allocated memory to 0.

Return value

Returns a pointer to the newly allocated memory. Returns NULL if the requested memory could not be allocated or if either *nelem* or *size* was 0.

See also

free malloc realloc

Sample program

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

void main(void)
{
char *s;

s = (char *)calloc(10, sizeof(char));
strcpy(s, ”sample”);
}
```

ceil Function

Function

Calculates the ceiling of a value.

Format

#include <math.h>

double ceil(double *x*);

x Floating-point value

Description

The ceil function finds the smallest integer greater than or equal to *x*.

Return value

Returns the integer found as a value of type double.

See also

floor fmod

Sample program

```
#include <math.h>

void main(void)
{
double num;
double up;

num = 12.3;

up = ceil(num);
}
```

COS Macro/function

Function

Calculates the cosine.

Format

#include <math.h>

double cos(double *x*);

x Angle in radians

Description

The cos function calculates the cosine of the input value *x*.

Return value

Returns a value in the range -1 to 1.

See also

acos asin atan atan2 sin tan

Sample program

```
#include <math.h>

void main(void)
{
double x;
double res;

x = 0.5;

res = cos(x);
}
```

cosh Function

Function

Calculates the hyperbolic cosine.

Format

#include <math.h>

double cosh(double *x*);

x Angle in radians

Description

The cosh function calculates the hyperbolic cosine $(e^x + e^{-x})/2$ of the argument *x*.

Return value

Returns the hyperbolic cosine of the argument *x*. If the result of the calculation is too large, cosh returns HUGE_VAL and sets the global variable errno to ERANGE.

See also

acos asin atan atan2 cos sin sinh tan tanh

Sample program

```
#include <math.h>

void main(void)
{
double x;
double res;

x = 0.5;

res = cosh(x);
}
```

div Function

Function

Calculates the quotient and remainder of two integers of type int.

Format

#include <stdlib.h>

div_t div(int *numer*, int *denom*);

numer Numerator

denom Denominator

Description

The div function divides the argument *numer* by the argument *denom* and returns the result in a structure of type div_t. The structure div_t has two members of type int, quot and rem, and the div function insert the quotient in quot and the remainder in rem before returning the structure.

Return value

Returns a structure that has quot (quotient) and rem (remainder) members.

See also

ldiv

Sample program

```
#include <stdlib.h>

void main(void)
{
div_t res;
int num, den;
int quot, rem;

num = 32767;
den = 1000;

res = div(num, den);
quot = res.quot;
rem = res.rem;
}
```

exp Function

Function

Calculates the exponential function e^x.

Format

#include <math.h>

double exp(double *x*);

x Floating-point value

Description

The exp function calculates the exponential function e^x.

Return value

Returns e^x. If an overflow occurs, exp returns HUGE_VAL, and if an underflow occurs, exp returns 0.0. In both these cases, it sets the global variable errno to ERANGE.

See also

frexp ldexp log log10 pow sqrt

Sample program

```
#include <math.h>

void main(void)
{
double x;
double res;

x = 5.5;

res = exp(x);
}
```

fabs Function

Function

Calculates the absolute value of a floating-point number.

Format

#include <math.h>

double fabs(double *x*);

x Floating-point value

Description

The fabs function calculates the absolute value of the floating-point number given as the argument *x*.

Return value

Returns the absolute value of its argument *x*.

See also

abs labs

Sample program

```
#include <math.h>

void main(void)
{
double num;
double val;

num = 12.3;

val = fabs(num);
}
```

floor

Function

Function

Calculates the floor of a value.

Format

#include <math.h>

double floor(double *x*);

x Floating-point value

Description

The floor function finds the largest integer that does not exceed *x*.

Return value

Returns the largest integer that does not exceed *x* as a floating-point number.

See also

ceil fmod

Sample program

```
#include <math.h>

void main(void)
{
double num;
double down;

num = 12.3;

down = floor(num);
}
```

fmod Function

Function

Calculates the floating-point remainder.

Format

#include <math.h>

double fmod(double *x*, double *y*);

x, *y* Floating-point value

Description

The fmod function calculates the remainder of *x* divided by *y* such that *x* = a*y* + f, where a is an integer and f is a value that has the same sign as *x* and meets the condition |f| < |*y*|.

Return value

Returns the remainder as a floating-point number. If *y* is 0, it sets the global variable errno to EDOM.

See also

ceil fabs floor modf

Sample program

```
#include <math.h>

void main(void)
{
double x;
double y;
double res;

x = 7.0;
y = 2.0;

res = fmod(x, y);
}
```

free Function

Function

Releases memory.

Format

#include <stdlib.h>

void free(void ∗*ptr*);

void free_n(void __near ∗*ptr*);

void free_f(void __far ∗*ptr*);

ptr Pointer to the memory to be released

Description

The free function releases memory allocated by calloc, malloc, or realloc. The argument *ptr* must be a pointer that was returned by calloc, malloc, or realloc. Operation is not guaranteed if any other pointer is passed to free. If a null pointer is passed to free as *ptr*, free returns without performing any operations.

Return value

None.

See also

calloc malloc realloc

Sample program

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

void main(void)
{
char *s;
s = (char *)malloc(10);
strcpy(s, "sample");
     .
     .
     .
free(s);
}
```

frexp Function

Function

Divides a floating-point number into its mantissa and exponent parts.

Format

#include <math.h>

double frexp(double *x*, int ∗*pexp*);

double frexp_n(double *x*, int __near ∗*pexp*);

double frexp_f(double *x*, int __far ∗*pexp*);

x	Floating-point value
pexp	Pointer to an area to hold the exponent

Description

The frexp function divides its argument *x* into a mantissa m (such that the absolute value of m is greater than or equal to 0.5 and strictly less than 1.0) and an exponent n, such that $x = m \times 2^n$. It stores the exponent n, which is an integer value, at the location pointed to by *pexp*.

Return value

Returns the value of the mantissa, m.

See also

ldexp modf

Sample program

```
#include <math.h>

void main(void)
{
double x;
double mant;
int pexp;

x = 18.4;

mant = frexp(x, &pexp);
}
```

isalnum - isxdigit

Macro/function

Function

These functions determine the type of characters.

Format

```
#include <ctype.h>
int isalnum( int c );
int isalpha( int c );
int iscntrl( int c );
int isdigit( int c );
int isgraph( int c );
int islower( int c );
int isprint( int c );
int ispunct( int c );
int isspace( int c );
int isupper( int c );
int isxdigit( int c );
```

c Single-byte character (an integer in the range 0x00 to 0xff)

Description

These routines determine the type of a character *c*, and return the result of that determination. These routines presuppose the ASCII character set.

The result of the determination is undefined if a value for *c* outside the range 0x00 to 0xff is used.

The table lists the classification determined by each of these routines.

Routine	Classification performed
isalnum	Tests whether a character is a digit ('0' to '9') or a letter ('a' to 'z' or 'A' to 'Z').
isalpha	Tests whether a character is a letter ('a' to 'z' or 'A' to 'Z').
iscntrl	Tests whether a character is a control character (0x00 to 0x1f, and 0x7f).
isdigit	Tests whether a character is a digit ('0' to '9').
isgraph	Tests whether a character is a printing character (0x21 to 0x7e) other than space (' ').
islower	Tests whether a character is a lowercase character ('a' to 'z').
isprint	Tests whether a character is a printing character (0x20 to 0x7e) including space (' ').
ispunct	Tests whether a character is a punctuation character (0x21 to 0x2f, 0x3a to 0x40, 0x5b to 0x60, and 0x7b to 0x7e).
isspace	Tests whether a character is a space character (0x09 to 0x0d and ' ').
isupper	Tests whether a character is an uppercase character ('A' to 'Z').
isxdigit	Tests whether a character is a hexadecimal digit ('0' to '9', 'a' to 'f', or 'A' to 'F').

Return value

These routines return a value other than zero if the condition is met, and zero if the condition is not met.

The return value is undefined if *c* has a value outside the range 0x00 to 0xff.

See also

toupper tolower

Sample program

```
#include <ctype.h>

void main(void)
{
int c;
int     retval1 , retval2 , retval3 , retval4 , retval5;

/* Test the type of the characters 'a' through 'z'. */

for (c = 'a'; c <= 'z'; ++c)
     {
     retval1 = isalnum(c);  /* TRUE, since all are alphabetic. */
     retval2 = islower(c);  /* TRUE, since all are lower case. */
     retval3 = isupper(c);  /* FALSE, since none are upper case. */
     retval4 = isdigit(c);  /* FALSE, since none are digits. */
     retval5 = isxdigit(c); /* TRUE for 'a' to 'f',  */
                            /* FALSE for all others. */
     }
}
```

labs Function

Function

Calculates the absolute value of an integer of type long.

Format

#include <stdlib.h>

long labs(long *n*);

n Integer value

Description

The labs function calculates the absolute value of the integer *n* of type long.

Return value

Returns a value in the range 0 to 2147483647. However, if *n* is -2147483648, labs returns -2147483648.

See also

abs fabs

Sample program

```
#include <stdlib.h>

void main(void)
{
long n, res;

n = -123456;
res = labs(n);
}
```

ldexp

Function

Function

Computes a real number from the mantissa and exponent.

Format

#include <math.h>

double ldexp(double *x*, int *xexp*);

x	Floating-point value
xexp	Integer exponent

Description

The ldexp function computes $x \times 2^{xexp}$.

Return value

Returns the result of computing $x \times 2^{xexp}$. If the result of the calculation is too large, ldexp sets the global variable errno to ERANGE.

See also

exp frexp modf

Sample program

```
#include <math.h>

void main(void)
{
double x;
double val;

x = 4.5;

val = ldexp(x, 5);
}
```

ldiv Function

Function

Calculates the quotient and remainder for two integer values of type long.

Format

#include <stdlib.h>

ldiv_t ldiv(long int *numer*, long int *denom*);

numer Numerator

denom Denominator

Description

The ldiv function divides the argument *numer* by the argument *denom* and returns the result in a structure of type ldiv_t. The structure ldiv_t has two members of type long, quot and rem, and the ldiv function insert the quotient in quot and the remainder in rem before returning the structure.

Return value

Returns a structure that has quot (quotient) and rem (remainder) members.

See also

div

Sample program

```
#include <stdlib.h>

void main(void)
{
ldiv_t res;
long num, den;
long quot, rem;

num = 165536;
den = 1000;

res = ldiv(num, den);
quot = res.quot;
rem = res.rem;
}
```

log

Macro/function

Function

Calculates the natural logarithm of the argument x.

Format

#include <math.h>

double log(double *x*);

x Value whose natural logarithm will be calculated

Description

The log function calculates the natural logarithm of the argument *x*.

Return value

Returns the calculated value, ln(*x*). If the argument is negative, it sets the global variable errno to EDOM. If the argument *x* is zero, log returns -HUGE_VAL, and if the result value is too large, log returns HUGE_VAL. In both of these cases, it sets the global variable errno to ERANGE.

See also

exp log10

Sample program

```
#include <math.h>

void main(void)
{
double x;
double res;

x = 10;

res = log(x);
}
```

log10 Macro/function

Function

Calculates the common logarithm of the argument x.

Format

#include <math.h>

double log10(double *x*);

x Value whose base-ten logarithm will be calculated

Description

The log10 function calculates the base-ten logarithm of the argument *x*.

Return value

Returns the calculated value. If the argument is negative, it sets the global variable errno to EDOM. If the argument *x* is zero, log10 returns -HUGE_VAL, and if the result value is too large, log10 returns HUGE_VAL. In both of these cases, it sets errno to ERANGE.

See also

exp log

Sample program

```
#include <math.h>

void main(void)
{
double x;
double res;

x = 10;

res = log10(x);
}
```

longjmp Function

Function

Performs a global (nonlocal) jump.

Format

#include <setjmp.h>

void longjmp(jmp_buf *environment*, int *value*);

void longjmp_n(jmp_buf_n *environment*, int *value*);

void longjmp_f(jmp_buf_f *environment*, int *value*);

environment An area of storage used to hold the execution environment

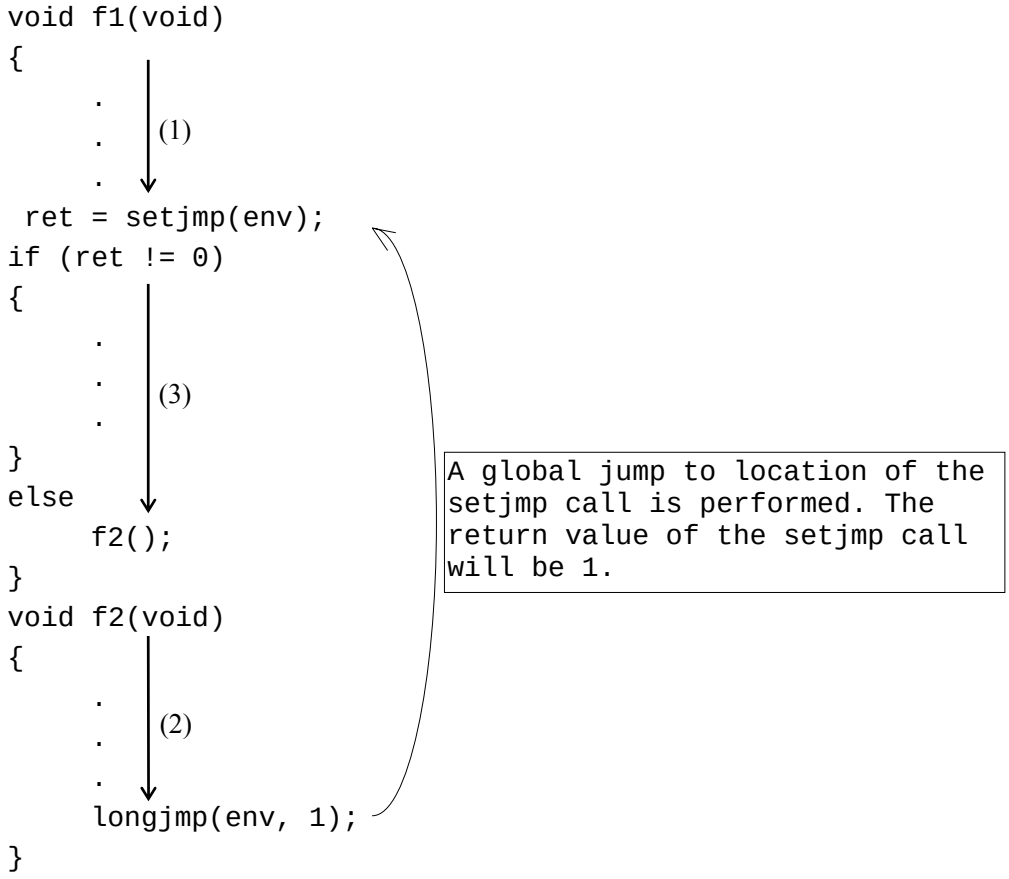
value The value to be used as the return value from setjmp

Description

The longjmp function performs a global (that is, nonlocal) jump to the place from which the corresponding setjmp was called.

Global jumps can be performed by using the setjmp and longjmp functions. The longjmp function restores the execution environment saved in advance in *environment* by setjmp. As a result, the program appears, after longjmp was called, to have executed as though it had returned from setjmp. The argument *value* becomes the return value from setjmp when the program returns to the setjmp execution environment.

This section presents a simple example of setjmp and longjmp operation. The program executes in the order (1), (2), and then (3).



The value of *value* must be a value other than 0. If zero is passed, setjmp will return 1.

Observe the following points when using longjmp. Programs that fail to observe these points may behave in unexpected ways.

(1) Always save the execution environment with setjmp before calling longjmp.

(2) The longjmp function must not be called after the function that calls setjmp has returned.

Return value

None.

See also

setjmp

Sample program

```
#include <errno.h>
#include <setjmp.h>

void function1( void );
void function2( void );

jmp_buf environment;

void main(void)
{
     int retval;
     retval = setjmp(environment);
     if ( retval != 0 )
     {
          /* error process  */
     }
function1( );
     .
     .
     .
function2( );
     .
     .
     .
}

void function1( void )
{
     if (errno)
          longjmp(environment , 1);
     .
     .
     .
}

void function2( void )
{
     if (errno)
          longjmp( environment , 2 );
     .
     .
     .
}
```

malloc Function

Function

Allocates memory.

Format

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

size The size of the memory area to be allocated

Description

The malloc function allocates a memory area of *size* bytes in the dynamic segment. Each time malloc is called, it consumes *size* + 2 bytes if *size* is even and *size* + 3 bytes if *size* is odd due to memory management and boundary requirements. The allocated memory is not initialized.

The area with the largest size of the areas remaining after all the logical segments are allocated to the address space by the RLU8 linker is allocated as the dynamic segment.

Return value

Returns a pointer to the allocated memory. Returns NULL if it was unable to allocate memory of the requested size or if *size* was 0.

See also

calloc free realloc

Sample program

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

void main(void)
{
     char *s;

     if ((s = (char *)malloc(10)) != NULL)
     {
          strcpy(s, "sample");
     }
}
```

memchr Function

Function

Searches for a 1-byte data item in a fixed memory region.

Format

#include <string.h>

void ∗memchr(const void ∗*region*, int c, size_t *count*);

void __near ∗memchr_n(const void __near ∗*region*, int c, size_t *count*);

void __far ∗memchr_f(const void __far ∗*region*, int c, size_t *count*);

region	Pointer to a memory area
c	Data to search for
count	Number of bytes to search over

Description

The memchr function tests whether *c* is present in the first *count* bytes from the start of *region*. Although *c* is of type int, it must have a value in the range 0x00 to 0xff.

Return value

Returns a pointer to the first place *c* appears, if *c* is present in the first *count* bytes of *region*. Returns NULL if *c* is not found. Returns NULL if *count* is 0.

See also

memcmp memcpy memset strchr

Sample program

```
#include <string.h>

char data[16] =
{
     0x00,0x10,0x20,0x30,0x40,0x50,0x60,0x70,
     0x80,0x90,0xa0,0xb0,0xc0,0xd0,0xe0,0xf0
};

void main(void)
{
     char *ptr;
     .
     .
     .
     /* Returns the address of data[8]. */
     ptr = memchr( data , 0x80 , 16 );
     .
     .
     .
     /* Returns NULL since there is no 0xff. */
     ptr = memchr( data , 0xff , 16 );
     .
     .
     .
     /* Returns NULL since there is no 0x80  */
     /* in the first 4 bytes.                */
     ptr = memchr( data , 0x80 , 4 );
     .
     .
     .
}
```

memcmp Function

Function

Compares two memory areas.

Format

#include <string.h>

int memcmp(const void *region1, const void *region2, size_t *count*);

int memcmp_nn(const void __near *region1, const void __near *region2, size_t *count*);

int memcmp_nf(const void __near *region1, const void __far *region2, size_t *count*);

int memcmp_fn(const void __far *region1, const void __near *region2, size_t *count*);

int memcmp_ff(const void __far *region1, const void __far *region2, size_t *count*);

region1 Memory area 1

region2 Memory area 2

count Number of bytes to compare

Description

The memcmp function compares *region1* to *region2* from the start one byte at a time for *count* bytes. Unlike the strcmp function, memcmp does compare data following any null characters ('\0') that appear in the data.

Return value

Returns the following as the result of the comparison.

Return value	Comparison result
0	*region1* and *region2* are identical.
Positive	*region1* is larger than *region2*.
Negative	*region1* is smaller than *region2*.

See also

memchr memcpy memset strcmp

Sample program

```
     #include <string.h>

     char buf1[16] = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
     char buf2[16] = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
     char buf3[16] = {0,1,2,3,4,5,6,7,8,9,10, 1, 2, 3, 4, 5};

     void main( void )
     {
          int     ret;

          /*  Returns 0 since the contents   */
          /*  of the buffers are identical.  */
          ret = memcmp(buf1, buf2, 16);
          .
          .
          .
          /*  Returns a positive value since the first is larger. */
          ret = memcmp(buf1, buf3, 16);
          .
          .
          .
          /*  Returns a negative value since the second is larger. */
          ret = memcmp( buf3 , buf2 , 16 );
          .
          .
          .
}
```

memcpy Function

Function

Copies the data from one memory area to another memory area.

Format

#include <string.h>

void ∗memcpy(void ∗*dest*, const void ∗*src*, size_t *count*);

void __near ∗memcpy_nn(void __near ∗*dest*, const void __near ∗*src*, size_t *count*);

void __near ∗memcpy_nf(void __near ∗*dest*, const void __far ∗*src*, size_t *count*);

void __far ∗memcpy_fn(void __far ∗*dest*, const void __near ∗*src*, size_t *count*);

void __far ∗memcpy_ff(void __far ∗*dest*, const void __far ∗*src*, size_t *count*);

dest	Target of the copy
src	Source for the copy
count	Number of bytes to copy

Description

The memcpy function copies the first *count* bytes in *src* into *dest*. Unlike strcpy and strncpy, data including and beyond and null characters ('\0') that appear in the data will be copied.

Normal operation is not guaranteed if the source and destination areas overlap. Use the memmove function to copy between areas of memory that overlap.

Return value

Returns *dest*.

See also

memchr memcmp memmove memset strcpy strncpy

Sample program

```
#include <string.h>

char data1[16] =
     {
          0x00, 0x10, 0x20, 0x30, 0x40, 0x50, 0x60, 0x70,
          0x80, 0x90, 0xa0, 0xb0, 0xc0, 0xd0, 0xe0, 0xf0
     };
char data2[16];

void main( void )
{
     char *retptr;
     .
     .
     .
     retptr = memcpy(data2, data1,16);
     .
     .
     .
}
```

memmove Function

Function

Copies the data from one memory area to another memory area.

Format

```
#include <string.h>
void *memmove( void *dest, const void *src, size_t count );
void __near *memmove_nn( void __near *dest, const void __near *src, size_t count );
void __near *memmove_nf( void __near *dest, const void __far *src, size_t count );
void __far *memmove_fn( void __far *dest, const void __near *src, size_t count );
void __far *memmove_ff( void __far *dest, const void __far *src, size_t count );
```

dest Target of the copy

src Source for the copy

count Number of bytes to copy

Description

The memmove function copies the first *count* bytes in *src* into *dest*. The memmove function operates correctly even if the source and destination areas for the copy operation overlap. Unlike strcpy and strncpy, data including and beyond and null characters ('\0') that appear in the data will be copied.

Return value

Returns *dest*.

See also

memcpy strcpy strncpy

Sample program

```
#include <string.h>

char data[] =
{
0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f,
0x20, 0x21, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27,
0x28, 0x29, 0x2a, 0x2b, 0x2c, 0x2d, 0x2e, 0x2f
};

void main(void)
{
     .
     .
     .
     /* Copy the 32 bytes of data starting           */
     /* at data + 16 into the area starting at data. */
     /* Although the memory areas overlap, */
     /* the copy is performed correctly.   */
     memmove(data, data+16 , 32);
     .
     .

}
```

memset Function

Function

Initializes a fixed memory area to a specified data byte.

Format

#include <string.h>

void *memset(void *region, int c, size_t count);

void __near *memset_n(void __near *region, int c, size_t count);

void __far *memset_f(void __far *region, int c, size_t count);

region	Memory area
c	Data byte to store
count	Byte count

Description

The memset function initializes each byte in the first *count* bytes from the start of *region* to *c*. Although *c* is of type int, it must have a value in the range 0x00 to 0xff.

Return value

Returns *region*.

See also

memchr memcpy memcmp memmove

Sample program

```
#include <string.h>

char data[64];

void main(void)
{
     char *retptr;

     /* Initialize the 32 bytes from the start */
     /* of the buffer data to 0xff.            */
     retptr = memset( data , 0xff , 32 );
}
```

modf Function

Function

Separates a floating-point number into integer and fractional parts.

Format

#include <math.h>

double modf(double *x*, double ∗*pint*);

double modf_n(double *x*, double __near ∗*pint*);

double modf_f(double *x*, double __far ∗*pint*);

x	Floating-point value
pint	Pointer to an area to hold the integer part

Description

The modf function separates its argument *x*, a floating-point number, into its integer and fractional parts, stores the integer part of *x* in the area pointed to by *pint*, and returns the fractional part as the return value.

Return value

Returns the fractional part of the argument *x* as a signed value.

See also

fmod frexp ldexp

Sample program

```
#include <math.h>

void main(void)
{
     double x;
     double pint;
     double frac;

     x = 10.2;

     frac = modf(x, &pint);
}
```

offsetof **Macro**

Function

Determines the offset of a field in a structure.

Format

#include <stddef.h>

size_t offsetof(*structname*, *fieldname*);

structname	Name of a structure
fieldname	Name of a member in that structure

Description

The offsetof macro determines the number of bytes the member *fieldname* of the structure *structname* is offset from the start of that structure.

Return value

Returns the number of bytes by which the member *fieldname* of the structure *structname* is offset from the start of that structure.

Sample program

```
#include <stddef.h>

typedef struct {
int member1;
long member2;
char member3;
} structname;

void main(void)
{
size_t ret1;
size_t ret2;
size_t ret3;

ret1 = offsetof(structname, member1);
ret2 = offsetof(structname, member2);
ret3 = offsetof(structname, member3);
}
```

pow Function

Function

Calculates *x* raised to the power of *y*.

Format

#include <math.h>

double pow(double *x*, double *y*);

x Base

y Power to which *x* is to be raised

Description

The pow function calculates *x* raised to the *y* power.

Return value

Returns the value of calculating *x* to the *y* power. Depending on the values of the arguments, pow may cause an overflow or may be unable to calculate the result. If an overflow occurs, pow returns HUGE_VAL and sets the global variable errno to ERANGE. If *x* is negative and *y* is not an integer, pow sets errno to EDOM. If both arguments *x* and *y* are 0, pow returns 1.

See also

exp sqrt

Sample program

```
#include <math.h>

void main(void)
{
     double x;
     double y;
     double val;

     x = 2.0;
     y = 3.0;

     val = pow(x, y);
}
```

qsort Function

Function

Sorts an array using the quicksort algorithm.

Format

```
#include <stdlib.h>
void qsort( void *base, size_t n, size_t size, int (*cmp)( const void *, const void * ) );
void qsort_n( void __near *base, size_t n, size_t size,
              int (*cmp_nn)( const void __near *, const void __near * ) );
void qsort_f( void __far *base, size_t n, size_t size,
              int (*cmp_ff)( const void __far *, const void __far * ) );
```

base Start of the array to be sorted

n Number of elements in the array

size Size of each element in the array

cmp, *cmp_nn*, *cmp_ff* Pointer to a comparison function

Description

The qsort function sorts the elements of the array using the quicksort algorithm. The qsort function calls the user-defined comparison function passed as *cmp* to sort the elements of the array.

The function specified by *cmp* is a comparison function written by the user, and must take two arguments of type pointer to void (void *). If the arguments to this function are *elem1* as the first argument and *elem2* as the second argument, then this comparison function must return a value as follows according to the arguments it receives.

Condition	Return value
*elem1 < *elem2	A negative value
*elem1 == *elem2	0
*elem1 > *elem2	A positive value

Return value

None.

See also

bsearch

Sample program

```
#include <stdlib.h>

int compare(int *, int *);
int base[ ] = {12, 23, 15, 128, 43, 25};

void main( void )
{
     qsort(base, 6, sizeof (int), compare);
}

int compare(int *elem1, int *elem2)
{
     return(*elem1 - *elem2);
}
```

rand Function

Function

Generates a pseudorandom number.

Format

#include <stdlib.h>

int rand(void);

Description

The rand function generates a pseudorandom number in the range 0 to RAND_MAX and returns that value.

Return value

Returns a pseudorandom number.

See also

srand

Sample program

```
#include <stdlib.h>

int random[20];

void main( void )
{
     int i;

     for (i = 0; i < 20; ++i)
     random[i] = rand( );
}
```

realloc Function

Function

Reallocates memory.

Format

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

ptr Pointer to the memory to be reallocated

size The size of the memory area to be allocated

Description

The realloc function reallocates memory allocated with calloc or malloc.

The realloc function allocates a memory area with the requested size, and returns a pointer to that area. If memory is newly allocated, the contents of the original area is copied to the newly allocated area. If *ptr* is NULL, realloc operates identically to malloc. If *size* is 0, and *ptr* is not NULL, the memory pointed to by *ptr* is freed.

Return value

Returns a pointer to the reallocated memory. If the memory reallocation could not be performed, realloc returns NULL.

See also

calloc free malloc

Sample program

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

char string1[] = "library";
char string2[] = "reference.";

void main(void)
{
     char *s1, *s2;

     s1 = (char *)malloc(strlen(string1) + 1);
     strcpy(s1, string1);

     /* Perform memory reallocation.
     At this point, the contents of s1 are copied to s2.*/
     s2 = (char *)realloc(s1, strlen(s1) + strlen(string2) + 1);

     /* Concatenate string2 onto s2. */
     strcat(s2, string2);
     /* The contents of s2 will now be "library reference." */
}
```

setjmp Macro

Function

Saves the current program execution environment to allow a global jump.

Format

```
#include <setjmp.h>
int   setjmp( jmp_buf environment );
int   setjmp_n( jmp_buf_n environment );
int   setjmp_f( jmp_buf_f environment );
```

environment Area to hold the execution environment

Description

The setjmp function saves the current program execution environment in *environment*.

Global jumps can be performed by using the setjmp and longjmp functions. The environment saved by setjmp can be restored by longjmp. As a result, the program appears, after longjmp was called, to have executed as though it had returned from setjmp.

The setjmp function returns zero when it is called to save the environment, but setjmp returns a *value* other than 0, in particular the value of the *value* argument to longjmp when the environment is restored by longjmp. Thus the program that calls the setjmp macro can determine whether it has just saved the environment, whether the environment has been restored by longjmp, or even from which longjmp the environment has been restored by referencing this return value.

Return value

Always returns 0 when called to save the execution environment. The return value when setjmp returns as a result of a call to longjmp will be a value other than 0, in particular, the value of the second argument, *value*, passed to longjmp.

See also

longjmp

Sample program

See the longjmp documentation.

sin — Macro/function

Function

Calculates the sine function.

Format

#include <math.h>

double sin(double *x*);

x Angle in radians

Description

The sin function calculates the sine of the argument *x*.

Return value

Returns the sine of the argument *x*.

See also

acos asin atan atan2 cos tan

Sample program

```
#include <math.h>

void main(void)
{
     double x;
     double res;

     x = 0.5;

     res = sin(x);
}
```

sinh Function

Function

Calculates the hyperbolic sine function.

Format

#include <math.h>

double sinh(double *x*);

x Angle in radians

Description

The sinh function calculates the hyperbolic sine $(e^{x}-e^{-x})/2$ of the argument *x*.

Return value

Returns the hyperbolic sine of the argument *x*. If the result of the calculation is too large, sinh returns HUGE_VAL with an appropriate sign and sets the global variable errno to ERANGE.

See also

acos asin atan atan2 cos cosh sin tan tanh

Sample program

```
#include <math.h>

void main(void)
{
     double x;
     double res;

     x = 0.5;

     res = sinh(x);
}
```

sprintf Function

Function

Writes text to a character string using the specified format.

Format

#include <stdio.h>

int sprintf(char ∗*buffer*, const char ∗*format* [, *argument*, ...]);

int sprintf_nn(char __near ∗*buffer*, const char __near ∗*format* [, *argument*, ...]);

int sprintf_nf(char __near ∗*buffer*, const char __far ∗*format* [, *argument*, ...]);

int sprintf_fn(char __far ∗*buffer*, const char __near ∗*format* [, *argument*, ...]);

int sprintf_ff(char __far ∗*buffer*, const char __far ∗*format* [, *argument*, ...]);

buffer	Buffer to hold the resultant character string
format	Character string that specifies the format
argument	Optional arguments corresponding to the conversion specifiers

Description

The sprintf function creates a character string according to the specified *format* character string, and writes that character string to *buffer*. A null character is appended at the end of the created character string.

The *format* argument is a character string that consists of normal characters and an arbitrary number of conversion specifications. The number and types of the optional arguments that follow the *format* argument must match the number of conversion specifications in the format string and the types specified by those conversion specifications. Operation is not guaranteed if the number of optional arguments passed is smaller than the number of conversion specifications or if the type indicated by a conversion specification does not match the type of the corresponding argument. If the number of optional arguments exceeds the number of conversion specifications, the superfluous *arguments* will be ignored.

Conversion specifications have the following format.

% [*flags*] [*width*] [*.prec*] [{h|l|L}] *type*

The *flags* field is a sequence of flag characters. The *width* field specifies the width of the converted field. The *.prec* field specifies the precision. The {h|l|L} field is a size modifier. The *type* field specifies conversion specification character.

The flag, field width, precision, and size modifier fields are optional. The table presents an overview of the functions of these options.

Option	Meaning
flags	Prefixes that specify whether the output will be left or right justified, inclusion of a sign for numeric values, inclusion of a decimal point, octal, or hexadecimal.
width	Specifies the minimum width of the character field output.
.prec	Specifies the maximum number of characters output. For integers, specifies the minimum number of digits output.
{h|l|L}	Specifies the size of the argument.
	h short int
	l long
	L long double

Conversion Specifier Characters (*type*)

The table lists the conversion specifier characters.

Conversion Specifier Character	Type	Output format
d, i	int	Converts an int to a signed decimal character string.
o	unsigned int	Converts an unsigned into to an unsigned octal character string.
u	unsigned int	Converts an unsigned into to an unsigned decimal character string.
x	unsigned int	Converts an unsigned into to an unsigned hexadecimal character string. Digits with the values 10, 11, 12, 13, 14, and 15 are expressed as a, b, c, d, e, and f, respectively.
X	unsigned int	Converts an unsigned into to an unsigned hexadecimal character string. Digits with the values 10, 11, 12, 13, 14, and 15 are expressed as A, B, C, D, E, and F, respectively.
f	double	Converts a double to the signed format [-]*d.dddddd*.
e	double	Converts a double to the signed format [-]*d.dddddde+/-dd*.
E	double	The same as "e" except that E is used for the exponent.
g	double	Converts a double to the format specified by e or f. Normally, the f format is used. If the exponent is smaller than -4 or larger than the precision, the e format is used.
G	double	The same as "g" except that it converts to either E or f format.
c	int	Converts an int to a single character.
s	char *	Outputs the characters from the character string specified by the corresponding argument until either the precision is reached or the end of the character string is reached.
p	void *	Outputs the input argument as a pointer type.
n	int *	Stores the number of characters output up to that point in area pointed to by the corresponding argument, which must be of type pointer to int.
%	-	Outputs a percent character ("%").

The output formats described in the above table assume that none of the flag, width specifier, precision, or size modifier fields are specified. The section starting on the following page describes how the output format is influenced by combinations of the options with the conversion specifiers.

Flag Characters (*flags*)

The following types of flag are provided.

Flag character	Meaning
-	Justify the output character string to the left end of the field. If this flag is not specified, the character string is justified to the right.
+	Specifies that a sign indicator always be attached at the front of numbers. If this flag is not specified, a sign indicator is only output for negative numbers.
Space (0x20)	If the value is positive, put space in front of the number. If the value is negative, insert a minus sign (-) in front of the number.
#	This flag can be applied to conversion specifiers that apply to numerical data types, and allocate an appropriate format corresponding to the conversion specifier. See the following table.
0	If a 0 appears before one of the d, e, E, f, g, G, i, u, x, or X conversion specifiers, specifies that the field be filled with zeros instead of spaces. The 0 flag is ignored for the d, i, o, u, x, and X conversion specifiers if a precision is specified, and is ignored if the - flag is specified.

The # flag has the following effects when specified in combination with the conversion specifiers listed in the table.

Conversion Specifier Character	Influence of the # flag
c, d, i, u, s	No effect.
o	A 0 is attached at the front of the number for values other than 0.
x, X	Either 0x or 0X is attached at the front of the number.
e, E, f	A decimal point is always attached.
g, G	A decimal point is always attached and a 0 is also attached to the right of the decimal point.

Field width (*width*)

The field width specifies the minimum width of the field when writing out the character string.

When the field width is specified, if the converted character string is smaller than the specified field width space characters are added to make up the difference. The space characters are added at the right if the - flag is specified, and at the left otherwise. If the first character in the field width specification is '0', then zeros will be added instead of spaces. If the converted character string is larger than the specified field width, the field is expanded to the length of the converted character string.

It is also possible to indirectly specify the field width with an argument of type int by using an asterisk (*) as the field width specification. For example, this program:

```
char buf[20];
int width = 8;
int number = 1234;

sprintf(buf, "|%*d|", width, number);
```

will output the following character string to buf. Note that the value of the argument width is used as the field width.

```
|    1234|
```

Precision (.*prec*)

The precision field must always start with a period (.). The precision is specified in the same manner as the field width. If there are no digits following the period, the precision is taken to be 0.

The number of characters output when the precision is specified differs for the different conversion specifiers. The table lists the effects of specifying the precision as *n* for the different conversion specifiers.

Conversion Specifier Character	Effect on output
d, i, o, u, x, X	At least *n* character will be output.
e, E, f	The number of digits output after the decimal point will be *n*.
g, G	Valid digits in excess of the first *n* will not be output.
s	No more than *n* characters will be output.

Size modifier character

The size modifier characters modifies the type of the corresponding argument.

Modifier character	Size
h	For the d, i, o, u, x, X, and n conversion specifiers, causes the corresponding argument to be interpreted as a short int or an unsigned short int.
l	For the d, i, o, u, x, X, and n conversion specifiers, causes the corresponding argument to be interpreted as a long int or an unsigned long int. For the e, E, f, g, and G conversion specifiers, the corresponding argument will be interpreted as a double.
L	For the e, E, f, g, and G conversion specifiers, the corresponding argument will be

interpreted as a long double.

Important!

In the variable argument list, arguments that reference addresses depend on the format character string data access.

The types of the variable arguments to the printf family of functions must all be placed in the same memory area as the format character string. That is, if the format character string is NEAR data, the arguments that reference addresses must all be NEAR data, and if the format character string is FAR data, the arguments that reference addresses must all be FAR data. Operation is not guaranteed if an argument that references an address specifies a data model that differs from data model of the format character string. See the CCU8 User's Manual for details on data access. The following presents an example of this requirement.

```
#include <stdio.h>

char __near nbuf[20];
char __far fbuf[20];

char __near nstr[] = "near string";
char __far fstr[] = "far string";

const char __far format[] = "%s %d %p";

int res;

void main( void )
{
     int i = 10;

     strcpy_fn( fbuf, nstr );
     /* Since format is located in FAR memory,    */
     /* the arguments must also be in FAR memory. */
     res = sprintf_nf( nbuf, format, fbuf, i, fstr );
}
```

Return value

Returns the number of bytes output to *buffer*. Note that this does not include the null byte that terminates the output. If an error occurs, sprtinf returns EOF.

See also

sscanf

Sample program

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

char buf1[128];
char buf2[128];
char string[20];
int res1;
int res2;

void main(void)
{
     res1 = sprintf(buf1, "|%d|%4x|%04X|%+12.4f|",
                    10, 0xabc, 0xAB, 1234.567);

     strcpy(string, "ABCDEFG");
     res2 = sprintf(buf2, "|%-15s|%15s|", string, "abcdefg");
}
```

sqrt Function

Function

Calculates the square root of its argument, which is a real number.

Format

#include <math.h>

double sqrt(double *x*);

x Nonnegative floating-point value

Description

The sqrt function calculates the square root of its argument *x*.

Return value

Returns the calculated value of the square root of *x*. If *x* is negative, it sets the global variable errno to EDOM, and if the value is too large, it sets errno to ERANGE.

See also

exp log pow

Sample program

```
#include <math.h>

void main(void)
{
     double x;
     double val;

     x = 9.0;

     val = sqrt(x);
}
```

srand

Macro/function

Function

Initializes the pseudorandom number series.

Format

#include <stdlib.h>

void srand(unsigned int *seed*);

seed Value for initialization

Description

The srand function initializes the pseudorandom number series. The pseudorandom number series generated by rand can be changed by changing the value of *seed*.

Return value

None.

See also

rand

Sample program

```
#include <stdlib.h>

int random[20];

void main( void )
{
     int i;

     srand( 123 );
     for (i = 0; i < 20; ++i)
          random[i] = rand( );
}
```

sscanf Function

Function

Reads a character string and converts it to appropriate types according to a format.

Format

#include <stdio.h>

int sscanf(const char **string*, const char **format* [, *address*, ...]);

int sscanf_nn(const char __near **string*, const char __near **format* [, *address*, ...]);

int sscanf_nf(const char __near **string*, const char __far **format* [, *address*, ...]);

int sscanf_fn(const char __far **string*, const char __near **format* [, *address*, ...]);

int sscanf_ff(const char __far **string*, const char __far **format* [, *address*, ...]);

string	Character string to read from
format	Character string that specifies the format
address	Optional arguments corresponding to the conversion specifiers

Description

The sscanf function reads characters from the character string specified by *string*, converts them to data of the appropriate types according to the format string specified by *format*, and stores the results in the areas specified by the corresponding *address* arguments.

The format character string consists of white space, conversion specifications, and characters other than percent sign (%). When sscanf encounters a white space character in the format string, it reads in all the white space characters up to the next non-space character. A conversion specification starts with a percent sign (%) and specifies how part of the character string should be interpreted. When sscanf encounters a conversion specification, it acquires tokens from the corresponding character string and converts them. Other characters are read in and skipped as long as they match characters in the string being read.

The number of conversion specifications must be the same as the number of optional arguments that follow the *format* argument. Operation is not guaranteed if the number of corresponding arguments is smaller than the number of conversion specifications. If there are more corresponding arguments than there are conversion specifications, the excess arguments are ignored. Furthermore, the type required by the conversion specification and the type of the corresponding argument must match. Correct results cannot be expected if the types do not match.

Conversion specifications have the following format.

% [∗] [*width*] [{h|l|L}] *type*

The asterisk indicates that the next field (token) is to be read over and skipped. Nothing is written to the corresponding argument. The *width* field specifies the maximum number of characters (the input width) in the field to be read in. The characters h, l, and L are size modifier characters for the argument and change the type of the argument. The *type* field is a conversion specifier character.

The asterisk, input width, and type modifier character may be omitted.

Conversion Specifier Character

The table below describes the conversion specifier characters. The table lists the type of the corresponding argument and the interpretation of the character string read for each conversion specifier character.

Conversion Specifier Character	Argument type	Method for interpreting the character string read
d	int ∗	The character string is converted to a decimal integer. The format of the character string must be the same as that of the character strings interpreted by the strtol function when the radix is specified to be 10.
i	int ∗	The character string is converted to a decimal integer. The format of the character string must be the same as that of the character strings interpreted by the strtol function when the radix is specified to be 0.
o	unsigned int ∗	The character string is converted to an octal integer. The format of the character string must be the same as that of the character strings interpreted by the strtol function when the radix is specified to be 8.
u	unsigned int ∗	The character string is converted to an unsigned decimal integer. The format of the character string must be the same as that of the character strings interpreted by the strtoul function when the radix is specified to be 10.
x, X	unsigned int ∗	The character string is converted to an unsigned hexadecimal integer. The format of the character string must be the same as that of the character strings interpreted by the strtol function when the radix is specified to be 16.
f	float ∗	The character string is converted to a floating-point number. The format of the character string must be the same as that of the character strings interpreted by the strtod function when converting decimal notation to floating point.

Conversion Specifier Character	Argument type	Method for interpreting the character string read
e, E	float *	The character string is converted to a floating-point number. The format of the character string must be the same as that of the character strings interpreted by the strtod function when converting exponential notation to floating point.
g, G	float *	The character string is converted to a floating-point number. The format of the character string must be the same as that of the character strings interpreted by the strtod function when converting either decimal or exponential notation to floating point.
c	char *	Exactly the number of characters (including white space characters) specified as the input width are copied to the array specified by the argument. This operation does not insert a null character at the end of the specified array. If no input width is specified, exactly 1 character is copied.
s	char *	A character string not include white space is copied to the array specified by the argument. A null character is stored at the end of the array.
p	void *	A character string is read in as a pointer to void.
n	int *	The number of characters read in up to this point is written to the area specified by the argument.
%	-	A percent sign (%) is read in. Nothing is stored in an argument.
[...]	char *	Characters that match any of the set of characters enclosed in the square brackets ([]) are copied to the character string specified by the argument. White space may be included in the character set. Note that a set specifier of the form []...] allows the right square bracket character (']') to be included in the set.
[^...]	char *	Characters that do not match any of the set of characters enclosed in the square brackets ([]) are copied to the character string specified by the argument. White space may be included in the character set. Note that a set specifier of the form [^[...] allows the right square bracket character (']') to be included in the set.

The types of the arguments described above presume that no type modifier is specified. The table below describes how the type is changed when a type modifier is specified.

Type Modifier

The type modifier changes the type of the corresponding argument as described in the table.

Type Modifier	Interpreted type
h	For the d, i, o, u, x, X, and n conversion specifiers, the corresponding argument is taken to be a pointer to either short int or unsigned short int. The type modifier 'h' is ignored for all other conversion specifiers.
l	For the d, i, o, u, x, X, and n conversion specifiers, the corresponding argument is taken to be a pointer to either long int or unsigned long int. For the e, E, f, g, and G conversion specifiers, the corresponding argument is taken to be a pointer to double. The type modifier 'l' is ignored for all other conversion specifiers.
L	For the e, E, f, g, and G conversion specifiers, the corresponding argument is taken to be a pointer to long double. The type modifier 'L' is ignored for all other conversion specifiers.

Important!

In the variable argument list, arguments that reference addresses depend on the format character string data access.

The types of the variable arguments to the scanf family of functions must all be placed in the same memory area as the format character string. That is, if the format character string is NEAR data, the arguments that reference addresses must all be NEAR data, and if the format character string is FAR data, the arguments that reference addresses must all be FAR data. Operation is not guaranteed if an argument that references an address specifies a data model that differs from data model of the format character string. See the CCU8 User's Manual for details on data access. The following presents an example of this requirement.

```
#include <stdio.h>

char __near nstr[] = "input_data 1.234";
const char __far format[] = "%s %lf%n";

char __far fbuf[30];
double __far fd;
int __far fcnt;

int res;

void main( void )
{
     /* Since format is located in FAR memory,    */
     /* the arguments must also be in FAR memory. */
     res = sscanf_nf( nstr, format, fbuf, &fd, &fcnt );
}
```

Return value

Returns the number of input data items correctly read in. Returns EOF if an error occurred.

See also

sprintf

Sample program

```
#include <stdio.h>

int year, month, date;
char name[15];
float height;
int res;

void main(void)
{
     res=sscanf("1993.11.17,T.YAMADA,170.5","%d.%d.%d , %s , %f",
                    &year, &month, &date, name, &height );
}
```

strcat Function

Function

Appends strings.

Format

#include <string.h>

char *strcat(char **string1*, const char **string2*);

char __near *strcat_nn(char __near **string1*, const char __near **string2*);

char __near *strcat_nf(char __near **string1*, const char __far **string2*);

char __far *strcat_fn(char __far **string1*, const char __near **string2*);

char __far *strcat_ff(char __far **string1*, const char __far **string2*);

string1 String to be appended to

string2 String to append

Description

The strcat function appends *string2* to *string1* starting at the null character ('\0') in *string1* and appends a null character ('\0') at the end of the resultant string.

Return value

Returns *string1*.

See also

strncat strcpy strncpy

Sample program

```
     #include <string.h>

     char string1[128] = "library ";
     char string2[128] = "reference ";

     void main(void)
     {
          char  *retptr;
               .
               .
               .
          /* Creates the string "library reference". */
          retptr = strcat(string1 , string2);

          /* Creates the string "library reference manual". */
          retptr = strcat( retptr , "manual" );
               .
               .
               .
}
```

strchr Function

Function

Locates the position of the first occurrence of a character in a character string.

Format

```
#include <string.h>
char *strchr( const char *string, int c );
char __near *strchr_n( const char __near *string, int c );
char __far *strchr_f( const char __far *string, int c );
```

string Character string

c Character to be located

Description

The strchr function locates the character *c* in *string*. The null character ('\0') may be specified as *c*. Although *c* is of type int, it must have a value in the range 0x00 to 0xff.

Use the strrchr function to find the last occurrence of *c* in a string.

Return value

Returns a pointer to the position where the character first appears. If the character is not found, strchr returns NULL.

See also

memchr strcspn strrchr strspn

Sample program

```
     #include <string.h>

     char string[] = "01234567890123456789012345678 9";

     void main(void)
     {
          char *ptr;

          /* Since the first occurrence of '9' is at the ninth */
          /* position, strchr returns a pointer to string[9].  */
          ptr = strchr(string , '9');
               .
               .
               .
          /* If '\0' is specified, strchr returns */
          /* a pointer to the end of the string.  */
          ptr = strchr(string , '\0');
               .
               .
               .
          /* Since the character 'A' does not appear */
          /* in the string, strchr returns NULL.     */
          ptr = strchr(string , 'A');
}
```

strcmp Function

Function

Compares two character strings.

Format

#include <string.h>

int strcmp(const char **string1*, const char **string2*);

int strcmp_nn(const char __near **string1*, const char __near **string2*);

int strcmp_nf(const char __near **string1*, const char __far **string2*);

int strcmp_fn(const char __far **string1*, const char __near **string2*);

int strcmp_ff(const char __far **string1*, const char __far **string2*);

string1 First character string to compare

string2 Second character string to compare

Description

The strcmp function compares *string1* and *string2* lexicographically.

Return value

Returns the following as the result of the comparison.

Return value	Comparison result
0	*string1* and *string2* are identical.
Positive	*string1* is larger than *string2*.
Negative	*string1* is smaller than *string2*.

See also

memcmp strncmp

Sample program

```
     #include <string.h>

     /* string1 is larger than string2. */
     char string1[] = "ABCDE";
     char string2[] = "AAAAA";

     void main(void)
     {
          int retval;

          /* Since the first string is larger, */
          /* strcmp returns a positive value.  */
          retval = strcmp(string1, string2);
               .
               .
               .
          /* Since the second string is larger, */
          /* strcmp returns a negative value.   */
          retval = strcmp(string2, string1);
               .
               .
               .
          /* Since the character strings are */
          /* identical strcmp returns 0.     */
          retval = strcmp(string1, string1);
}
```

strcpy Function

Function

Copies a character string.

Format

#include <string.h>

char *strcpy(char **string1*, const char **string2*);

char __near *strcpy_nn(char __near **string1*, const char __near **string2*);

char __near *strcpy_nf(char __near **string1*, const char __far **string2*);

char __far *strcpy_fn(char __far **string1*, const char __near **string2*);

char __far *strcpy_ff(char __far **string1*, const char __far **string2*);

string1 Destination string

string2 Source string

Description

The strcpy function copies *string2*, including its terminating null character ('\0'), into *string1*.

Return value

Returns *string1*.

See also

memcpy strcat strncat strncpy

Sample program

```
#include <string.h>

char string[128];

void main(void)
{
     char *retptr;

     retptr = strcpy(string , "string data");
}
```

strcspn Function

Function

Determines the length of the initial section of a string that does not include any characters from a specified set of characters.

Format

#include <string.h>

size_t strcspn(const char **string1*, const char **string2*);

size_t strcspn_nn(const char __near **string1*, const char __near **string2*);

size_t strcspn_nf(const char __near **string1*, const char __far **string2*);

size_t strcspn_fn(const char __far **string1*, const char __near **string2*);

size_t strcspn_ff(const char __far **string1*, const char __far **string2*);

string1 Character string

string2 Character string that holds the character set

Description

The strcspn function searches for the first occurrence in *string1* of any character included in *string2* and returns that position as an offset from the start of *string1*. In other words, it determines the length of the section of the character string from the start of *string1* that consists of characters that do not appear in *string2*. The terminating null character ('\0') in *string1* is not considered in the search.

The strcspn function is quite similar to strpbrk. However strpbrk differs in that it returns a pointer to the position of the first character that appears. This library also provides the strspn function, which provides exactly the opposite functionality.

Return value

Returns the length from the start of *string1* up to the position of the first appearance of a character included in *string2*. The strcspn function returns the length of *string1* if not even one character from *string2* is present in *string1*, or if *string2* is the null string ("").

See also

strchr strrchr strpbrk strspn

Sample program

```
#include <string.h>

char string1[] = "ABCDEFG1234567";
char string2[] = "1234567";

void main(void)
{
     size_t retval;
     .
     .
     .
/*
Since there are 7 characters up to the point where one of the
characters in "1234567" appears in "ABCDEFG1234567",
strcspn returns 7.
*/
     retval = strcspn(string1 , string2);
     .
     .
     .
/*
Since none of the characters in "XYZ" appear in "ABCDEFG1234567",
strcspn returns the length of the character string.
*/
     retval = strcspn(string1 , "XYZ");
}
```

strlen Function

Function

Determines the length of a character string.

Format

#include <string.h>

size_t strlen(const char **string*);

size_t strlen_n(const char __near **string*);

size_t strlen_f(const char __far **string*);

string Character string

Description

The strlen function determines the length *string*, that is, the number of characters (number of bytes) from the start of *string* to the point just before the terminating null character ('\0').

Return value

Returns the length of *string*.

See also

None.

Sample program

```
#include <string.h>

char string[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

void main( void )
{
     size_t length;

/* Returns 26, the length of the character string.  */
length = strlen(string);
}
```

strncat Function

Function

Appends part of a character string starting at its beginning to the end of another character string.

Format

#include <string.h>

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

char __near *strncat_nn(char __near **string1*, const char __near **string2*, size_t *count*);

char __near *strncat_nf(char __near **string1*, const char __far **string2*, size_t *count*);

char __far *strncat_fn(char __far **string1*, const char __near **string2*, size_t *count*);

char __far *strncat_ff(char __far **string1*, const char __far **string2*, size_t *count*);

string1 String to be appended to

string2 String to append

count Number of characters to append

Description

The strncat function appends *count* bytes from the start of *string2* starting at the null character ('\0') in *string1*.

If *count* is longer than *string2*, then strncpy appends the whole of *string2* onto *string1*. In this case, the operation is identical to strcat. If *count* is 0, or if *string2* is the null string (""), the content of *string1* is unchanged.

Return value

Returns *string1*.

See also

strcat strcmp strcpy strncpy

Sample program

```
     #include <string.h>

     char string1[128] = "library ";
     char string2[128] = "reference";
     char string3[128] = "manual";

     void main(void)
     {
          char *retptr;

          /*
          Appends the first 3 character from the start of "reference".
          The resultant character string will be "library ref".
          */
          retptr = strncat(string1, string2 , 3);

          /*
          Here, a count longer than the length of "manual" is specified.
          The resultant character string will be "library refmanual".
          */
          retptr = strncat(retptr, string3, 20);

          /*
          Here, 0 is specified as the character count.
          The content of the character string will not be changed.
          */
          retptr = strncat(retptr, string3, 0);
}
```

strncmp Function

Function

Compares exactly the specified number of characters in two character strings.

Format

#include <string.h>

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

int strncmp_nn(const char __near **string1*, const char __near **string2*, size_t *count*);

int strncmp_nf(const char __near **string1*, const char __far **string2*, size_t *count*);

int strncmp_fn(const char __far **string1*, const char __near **string2*, size_t *count*);

int strncmp_ff(const char __far **string1*, const char __far **string2*, size_t *count*);

string1 First character string to compare

string2 Second character string to compare

count Number of characters to compare

Description

The strncmp function lexicographically compares *count* characters from the start of *string1* and *string2*. If *count* is less than the lengths of the strings being compared, the first *count* bytes from the beginnings of the strings are compared. If *count* is greater than the length the strings being compared, the strings up to the terminating null characters ('\0') are compared. If *count* is greater than the lengths of both *string1* and *string2*, the result of the comparison is identical to the result of a comparison using strcmp.

Return value

Returns the following as the result of the comparison.

Return value	Comparison result
0	*string1 and string2 are identical.*
Positive	*string1 is larger than string2.*
Negative	*string1 is smaller than string2.*

See also

memcmp strcat strcmp strcpy strncat strncpy

Sample program

```
#include <string.h>

/* string1 is greater than string2 starting at the seventh byte. */
char string1[] = "1234567890";
char string2[] = "1234560000";

void main(void)
{
     int retval;

     /* Compares up to the sixth byte. Returns 0. */
     retval = strncmp(string1 , string2 , 6);

     /*
     Compares up to the seventh byte.
     Since the first byte is greater, returns a positive number.
     */
     retval = strncmp(string1 , string2 , 7);
}
```

strncpy Function

Function

Copies the specified number of bytes from a character string.

Format

#include <string.h>

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

char __near *strncpy_nn(char __near **string1*, const char __near **string2*, size_t *count*);

char __near *strncpy_nf(char __near **string1*, const char __far **string2*, size_t *count*);

char __far *strncpy_fn(char __far **string1*, const char __near **string2*, size_t *count*);

char __far *strncpy_ff(char __far **string1*, const char __far **string2*, size_t *count*);

string1 Destination string

string2 Source string

count Number of characters to copy

Description

The strncpy function copies *count* bytes from the beginning of *string2* to *string1*. If *count* is less than or equal to the length of *string2*, a null character ('\0') is not added at the end of the copied string. If *count* is greater than the length of *string2*, all of *string2* is copied into *string1*, and the remaining bytes up to *count* bytes are filled with the null character ('\0').

Return value

Returns *string1*.

See also

memcpy strcat strncat strcpy

Sample program

```
     #include <string.h>

     char string1[] = "string";

     char string2[128];

     void main(void)
     {
          char *retptr;
          .
          .
          .
     /*
     The case where the character string is of length 6
     and the specified number of characters is 3.Only the first 3
     characters are copied. A null character is not added.
     */
          retptr = strncpy(string2, string1, 3);
          .
          .
          .
     /*
     The case where the character string is of length 6 and the
     specified number of characters is 10.
     After "string" is copied, the remaining 4 bytes are set to the null
     character.
     The result of the copy is "string\0\0\0\0".
     */
          retptr = strncpy(string2, string1, 10);
}
```

strpbrk Function

Function

Searches a character string for the first occurrence of a character in a set of characters.

Format

#include <string.h>

char *strpbrk(const char *_string1_, const char *_string2_);

char __near *strpbrk_nn(const char __near *_string1_, const char __near *_string2_);

char __near *strpbrk_nf(const char __near *_string1_, const char __far *_string2_);

char __far *strpbrk_fn(const char __far *_string1_, const char __near *_string2_);

char __far *strpbrk_ff(const char __far *_string1_, const char __far *_string2_);

string1 Character string

string2 Character string that holds the character set

Description

The strpbrk function searches for the first occurrence in *string1* of any character included in *string2* and returns a pointer to that character. The terminating null character ('\0') in *string1* is not considered in the search.

This routine is quite similar to strcspn. However strcspn differs in that it returns the offset of the first occurrence of the character from the start of the string.

Return value

Returns a pointer to the position in *string1* of the first character included in *string2*. These routines return NULL if none of the characters in *string2* occur in *string1*, or if either *string1* or *string2* is the null string ("").

See also

strchr strcspn strrchr strspn

Sample program

```
     #include <string.h>

     char string1[] = "ABCDEFG1234567";
     char string2[] = "1234567";

     void main(void)
     {
          char *ptr;

     /*
     Since there are 7 characters up to the point where one of the
     characters in "1234567" appears in "ABCDEFG1234567", strpbrk
     returns a pointer to the seventh character.
     */
          ptr = strpbrk(string1, string2);

     /*
     Since none of the characters in "XYZ" appear in "ABCDEFG1234567",
     strpbrk returns NULL.
     */
          ptr = strpbrk(string1, "XYZ");

     /*
     The strpbrk function returns NULL if a null character string is
     passed as an argument.
     */
          ptr = strpbrk(string1, "");
}
```

strrchr Function

Function

Locates the position of the last occurrence of a character in a character string.

Format

#include <string.h>

char *strrchr(const char **string*, int *c*);

char __near *strrchr_n(const char __near **string*, int *c*);

char __far *strrchr_f(const char __far **string*, int *c*);

string Character string

c Character to be located

Description

The strrchr function locates the last occurrence of *c* in *string*. The null character ('\0') may be specified as *c*. Although *c* is of type int, it must have a value in the range 0x00 to 0xff.

Use the strchr function to find the first occurrence of *c* in a *string*.

Return value

Returns a pointer to the position where the character last appears. If the character is not found, strrchr returns NULL.

See also

memchr strcspn strchr strspn

Sample program

```
     #include <string.h>

     char string[] = "01234567890123456789012345678 9";

     void main(void)
     {
          char *ptr;

     /* Since the 20th position is the last point where '0' appears, */
     /* strrchr returns a pointer to string[20]. */
          ptr = strrchr(string, '0');
          .
          .
          .
     /* If '\0' is specified, strrchr returns a pointer */
     /* to the end of the string. */
          ptr = strrchr(string, '\0');
          .
          .
          .
     /* Since the character 'A' does not appear in the string, */
     /* strchr returns NULL.  */
          ptr = strrchr(string, 'A');
}
```

strspn Function

Function

Finds the length of the first substring that only includes character from a specified character set.

Format

#include <string.h>

size_t strspn(const char **string1*, const char **string2*);

size_t strspn_nn(const char __near **string1*, const char __near **string2*);

size_t strspn_nf(const char __near **string1*, const char __far **string2*);

size_t strspn_fn(const char __far **string1*, const char __near **string2*);

size_t strspn_ff(const char __far **string1*, const char __far **string2*);

string1 Character string

string2 Character string that holds the character set

Description

The strspn function searches for the first occurrence in *string1* of a character not in *string2*, and returns an offset from the start of *string1*. In other words, it determines the length of the section of the character string from the start of *string1* that consists of characters that appear in *string2*. The terminating null character ('\0') in *string1* is not considered in the search.

This library also provides the strcspn function, which provides exactly the opposite functionality.

Return value

Returns the length from the start of *string1* up to the position of the first appearance of a character not included in *string2*. The strspn function returns 0 if the first character of *string1* is not included in *string2*, or if either *string1* or *string2* is the null string ("").

See also

strchr strrchr strpbrk strcspn

Sample program

```
     #include <string.h>

     char string1[] = "ABCDEFGABCDEFG1234567";
     char string2[] = "GFEDCBA";

     void main(void)
     {
          size_t retval;

     /*
     Returns 14, since the first 14 characters of
     "ABCDEFGABCDEFG1234567" consists of characters in "ABCDEFG".
     */
          retval = strspn(string1, string2);

     /*
     Returns 0, since the start of "ABCDEFGABCDEFG1234567"
     does not consist of characters in "XYZ".
     */
          retval = strspn(string1, "XYZ");
}
```

strstr Function

Function

Searches a string for a substring.

Format

#include <string.h>

char *strstr(const char *_string1_, const char *_string2_);

char __near *strstr_nn(const char __near *_string1_, const char __near *_string2_);

char __near *strstr_nf(const char __near *_string1_, const char __far *_string2_);

char __far *strstr_fn(const char __far *_string1_, const char __near *_string2_);

char __far *strstr_ff(const char __far *_string1_, const char __far *_string2_);

string1 Character string to search

string2 Character string to search for

Description

The strstr function searches for *string2* in *string1*.

Return value

Returns a pointer to the place where *string2* first occurs in *string1*. Returns NULL if *string2* is not present in *string1*, or if *string1* is the null string (""). The strstr function returns a pointer to *string1* if *string2* is the null string.

See also

strcspn strspn strchr strrchr strpbrk

Sample program

```
     #include <string.h>

     char string[] =
     /*
     0   ---   1   ---   2   ---   3   ---   4
     01234567890123456789012345678901234567890
     */
     "WORD1     WORD2     WORD3     WORD4    ";

     void main(void)
     {
          char *ptr;

     /*  Search for "WORD1".
       Returns string+0. */
          ptr = strstr(string, "WORD1");

     /* Search for "WORD2".
       Returns string+10.  */
          ptr = strstr(string, "WORD2");

     /* Search for "WORD3".
       Returns string+20. */
          ptr = strstr(string, "WORD3");

     /* Search for "NOTHING".
       Returns NULL, since the substring is not present. */
          ptr = strstr(string , "NOTHING");
}
```

strtod

Macro/function

Function

Converts a character string to a floating-point number of type double.

Format

#include <stdlib.h>

double strtod(const char ∗*s*, char ∗∗*endptr*);

double strtod_n(const char __near ∗*s*, char __near ∗ __near ∗*endptr*);

double strtod_f(const char __far ∗*s*, char __far ∗ __far ∗*endptr*);

s	Character string to be converted
endptr	Pointer to the character at which the scan stopped.

Description

The strtod function converts the character string specified by the argument *s* to a double precision floating-point number. The character string *s* must have the following format.

[*white space*] [*sign*] [*digit*] [.] [*digit*] [{e|E} [*sign*] *digit*]

The components of the character string have the following interpretations.

Symbol	Meaning
[*white space*]	Tab or space characters (May be omitted.)
[*sign*]	Sign (May be omitted.)
[*digit*] [.] [*digit*]	Character string that expresses the decimal fraction (May be omitted.)
[{e|E} [*sign*] *digit*]	Character string that express the exponent (May be omitted.)

The strtod function stops at the point it encounters a character it cannot interpret, and, if *endptr* is not NULL, stores a pointer to that character in *endptr*. If the converted value cannot be represented by type double, strtod returns HUGE_VAL and sets the global variable errno to ERANGE.

Return value

Returns the converted value of the character string as type double.

See also

atof atoi atol strtol strtoul

Sample program

```
#include <stdlib.h>

void main(void)
{
     double res;
     char *endp;

     res = strtod("1.234e+6", &endp);
}
```

strtok Function

Function

Divides a character string into tokens according to the specified delimiters and returns those tokens in order.

Format

```
#include <string.h>
char *strtok( char *string1, const char *string2 );
char __near *strtok_nn( char __near *string1, const char __near *string2 );
char __near *strtok_nf( char __near *string1, const char __far *string2 );
char __far *strtok_fn( char __far *string1, const char __near *string2 );
char __far *strtok_ff( char __far *string1, const char __far *string2 );
```

string1 Character string to be divided into tokens, or NULL

string2 Character string that consists of delimiters

Description

The term "token" as used here refers to substrings of *string1* that consists of characters other than those included in *string2*. The term "delimiter" refers to the characters included in *string2*. For example, if the delimiters are space (' '), colon (':'), and period ('.'), then the character string "RTLU8:Run Time Library." consists of the tokens "RTLU8", "Run", "Time", and "Library".

The strtok function divides *string1* into some number of tokens using the characters included in *string2* as delimiters. Pointers to the separated tokens can be acquired in order by calling these routines consecutively. If a pointer (other than NULL) to a string is passed as *string1*, if there are any delimiters specified by *string2* at the start of *string1*, these routines skip over those delimiters and return a pointer to the first token that appears. A null character ('\0') is stored at the end of the first token. If no token exists, strtok returns NULL. If NULL is passed for *string1*, these routines search for the next token. If a token is present, a pointer to the token is returned. A null character is stored at the end of the token. If no token exists, strtok returns NULL.

The strtok function is normally used as follows.

(1) Pass the string to be parsed as *string1* and acquire the first token.

(2) Pass NULL for *string1* and acquire the next token.

(3) Repeat (2) until NULL is returned.

The contents of *string2* may be changed each time the function is called. These functions insert a null character at the end of the token when they find a token. Note that this means that the contents of *string1* are changed by these functions.

Return value

Returns a pointer to the token if a token is present. If no token is present, returns NULL.

See also

strcspn strspn strchr strrchr strpbrk strstr

Sample program

```
/* Divides the character string into tokens using space, comma,
   semicolon, and colon as the delimiters. Stores pointers
   to the tokens in token_stock[] in order. */
#include <string.h>

char string[] = "   TOKEN1,TOKEN2;  TOKEN3::TOKEN4  ";
char delimiter[] = " ,;:";

char *token_stock[20];

void main(void)
{
     char *token_ptr;
     int token_counter = 0;

/* The first call. Here, strtok returns a pointer
   to the first token, TOKEN1. */
     token_ptr = strtok( string , delimiter);

     while(token_ptr != NULL)
     {
     token_stock[token_counter] = token_ptr; /* Pointer to a token. */
     ++token_counter;
     if(token_counter >= 20)
          break;
     /* Second and later calls. NULL is specified as the first
     argument.
     These calls return pointers to the tokens TOKEN2, TOKEN3,
     and TOKEN4 in order.
     Finally, strtok returns NULL, completing the operation. */
     token_ptr = strtok( NULL , delimiter);
     }

/* The result is as follows.
     token_stock[0] :: "TOKEN1"
     token_stock[1] :: "TOKEN2"
     token_stock[2] :: "TOKEN3"
     token_stock[3] :: "TOKEN4"
     token_stock[4] :: NULL

     The input string, string[], is changed as follows.
     "   TOKEN1\0TOKEN2\0  TOKEN3\0:TOKEN4\0";
*/
}
```

strtol

Macro/function

Function

Converts a character string to an integer of type long.

Format

#include <stdlib.h>

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

long strtol_n(const char __near *s, char __near * __near *endptr, int *base*);

long strtol_f(const char __far *s, char __far * __far *endptr, int *base*);

s	Character string to be converted
endptr	Pointer to the character at which the scan stopped.
base	Radix for conversion

Description

The strtol function converts the character string pointed to by the argument *s* to an integer of type long and returns that value. The character string *s* must have the following format.

[*white space*] [*sign*] [0] [{x|X}] [*digit*]

The components of the character string have the following interpretations.

Symbol	Meaning
[*white space*]	Tab or space characters (May be omitted.)
[*sign*]	Sign (May be omitted.)
[0]	Zero (May be omitted)
[{x|X}]	x or X (May be omitted)
[*digit*]	A string of digits (May be omitted)

The strtol function converts the character string *s*, which may be in a *base* from 2 to 36, according to the radix specified by *base*. That is, if *base* is 16, strtol interprets the character string as a hexadecimal number and recognizes the characters '0' to '9', 'a' to 'f', and 'A' to 'F' when converting the string to a number. If *base* is 0, strtol converts the first 1 or 2 characters to determined the base. The base is determined as follows.

First character	Second character	Value concerted
0	1 to 7	An octal number
0	x or X	A hexadecimal number
1 to 9		A decimal number

The strtol function returns 0 if *base* is 1, negative, or a value greater than 36. When these functions encounter a character they cannot interpret, they stop scanning and, if *endptr* is not NULL, set *endptr* to a pointer to the position of that character. If the acquired value is outside the range of values that can be represented by long, strtol returns LONG_MAX or LONG_MIN and sets the global variable errno to ERANGE.

Return value

Returns the value of the converted character string.

See also

atof atoi atol strtod strtoul

Sample program

```
#include <stdlib.h>

void main(void)
{
     long res;
     char *endp;

     res = strtol("0xabcdef", &endp, 16);
}
```

strtoul

Macro/function

Function

Converts a string to an integer of type unsigned long.

Format

#include <stdlib.h>

unsigned long strtoul(const char ∗*s*, char ∗∗*endptr*, int *base*);

unsigned long strtoul_n(const char __near ∗*s*, char __near ∗ __near ∗*endptr*, int *base*);

unsigned long strtoul_f(const char __far ∗*s*, char __far ∗ __far ∗*endptr*, int *base*);

s	Character string to be converted
endptr	Pointer to the character at which the scan stopped.
base	Radix for conversion

Description

The strtoul function converts the character string pointed to by the argument *s* to an integer of type unsigned long and returns that value. The character string *s* must have the following format.

[*white space*] [*sign*] [0] [{x|X}] [*digit*]

The components of the character string have the following interpretations.

Symbol	Meaning
[*white space*]	Tab or space characters (May be omitted.)
[*sign*]	Sign (May be omitted.)
[0]	Zero (May be omitted)
[{x|X}]	x or X (May be omitted)
[*digit*]	A string of digits (May be omitted)

The strtoul function converts the character string *s*, which may be in a *base* from 2 to 36, according to the radix specified by *base*. That is, if *base* is 16, strtoul interprets the character string as a hexadecimal number and recognizes the characters '0' to '9', 'a' to 'f', and 'A' to 'F' when converting the string to a number. If *base* is 0, strtoul converts the first 1 or 2 characters to determined the base. The base is determined as follows.

First character	Second character	Value concerted
0	1 to 7	An octal number
0	x or X	A hexadecimal number
1 to 9		A decimal number

The strtoul function returns 0 if *base* is 1, negative, or a value greater than 36. When these functions encounter a character they cannot interpret, they stop scanning and, if *endptr* is not NULL, set *endptr* to a pointer to the position of that character. If the acquired value is outside the range of values that can be represented by unsigned long, strtoul returns ULONG_MAX and sets the global variable errno to ERANGE.

Return value

Returns the value of the converted character string.

See also

atof atoi atol strtod strtol

Sample program

```
#include <stdlib.h>

void main(void)
{
     unsigned long res;
     char *endp;

     res = strtoul("0xabcdef", &endp, 16);
}
```

tan Function

Function

Calculates the tangent.

Format

#include <math.h>

double tan(double *x*);

x Angle in radians

Description

The tan function calculates the tangent of the argument *x*.

Return value

Returns the tangent of the argument *x*.

See also

acos asin atan atan2 cos sin

Sample program

```
#include <math.h>

void main(void)
{
     double x;
     double res;

     x = 0.5;

     res = tan(x);
}
```

tanh Function

Function

Calculates the hyperbolic tangent.

Format

#include <math.h>

double tanh(double *x*);

x Angle in radians

Description

The tanh function calculates the hyperbolic tangent, sing(*x*)/cosh(*x*) of the argument *x*.

Return value

Returns the hyperbolic tangent of the argument *x*.

See also

acos asin atan atan2 cos cosh sin sinh tan

Sample program

```
#include <math.h>

void main(void)
{
     double x;
     double res;

     x = 0.5;

     res = tanh(x);
}
```

tolower

Macro/function

Function

Converts uppercase characters to lowercase characters.

Format

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

c One-byte character (an integer in the range 0x00 to 0xff)

Description

The tolower function converts the character *c*, if it is an uppercase character ('A' to 'Z'), to the corresponding lowercase character ('a' to 'z'). Returns *c* unchanged if *c* is any other character.

The result of the conversion is undefined if *c* has a value outside the range 0x00 to 0xff.

Return value

Returns the corresponding lowercase character ('a' to 'z') if *c* is an uppercase character ('A' to 'Z'). Returns *c* without change if *c* is any other character. The return value is undefined if *c* has a value outside the range 0x00 to 0xff.

See also

islower isupper toupper

Sample program

```
     #include <ctype.h>

     char buffer1[] = "0123456789ABCDEFGabcdefg";
     char buffer2[64];

     void main(void)
     {
          int i;

          for(i = 0; buffer1[i] != '\0'; ++i)
          {
               buffer2[i] = tolower(buffer1[i]);
          }
     /*
     The contents of buffer2[] will be as follows.
     "0123456789abcdefgabcdefg"
     */
}
```

toupper

Macro/function

Function

Converts lowercase characters to uppercase characters.

Format

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

c One-byte character (an integer in the range 0x00 to 0xff)

Description

The toupper function converts the character *c*, if it is a lowercase character ('a' to 'z'), to the corresponding uppercase character ('A' to 'Z'). Returns *c* unchanged if *c* is not a lowercase character.

The result of the conversion is undefined if *c* has a value outside the range 0x00 to 0xff.

Return value

Returns the corresponding uppercase character ('A' to 'Z') if *c* is a lowercase character ('a' to 'z'). Returns *c* without change if *c* is any other character. The return value is undefined if *c* has a value outside the range 0x00 to 0xff.

See also

islower isupper tolower

Sample program

```
     #include <ctype.h>

     char buffer1[] = "0123456789ABCDEFGabcdefg";
     char buffer2[64];

     void main(void)
     {
          int i;

          for(i = 0; buffer1[i] != '\0'; ++i)
          {
               buffer2[i] = toupper(buffer1[i]);
          }
     /*
     The contents of buffer2[] will be as follows.
     "0123456789ABCDEFGABCDEFG"
     */
}
```

va_arg va_end va_start Macro

Function

These macros manipulate variable argument lists.

Format

#include <stdarg.h>

void va_start(va_list *ap*, *lastfix*);

type va_arg(va_list *ap*, *type*);

void va_end(va_list *ap*);

ap	Pointer to list of arguments
lastfix	The name of the last fixed argument passed to the called function
type	A data type name

Description

The va_arg, va_end, and va_start macros make it easy to perform operations on the variable argument list when creating functions with a variable number of arguments.

The va_start macro sets *ap* to point to the start of the list of optional arguments. The va_start macro must be called first.

The va_arg macro retrieves the argument that is current pointed to by *ap* as type *type*, and increments *ap* to point to the next argument. The *type* argument specifies the type returned by va_arg. The *ap* argument must be the same *ap* initialized by va_start.

After all the optional arguments have been read, the va_end macro sets up the environment so that processing performed later will be performed correctly. The va_end macro must be called after processing the optional arguments. Later operation is not guaranteed if it is not called.

Return value

The va_start and va_end macros have no return value. The va_arg macro returns the argument currently pointed to by *ap*.

See also

vfprintf vprintf vsprintf

Sample program

```
     #include <stdarg.h>

     int res;

     void main(void)
     {
          res = total_fn(7, 1, 2, 3, 4, 5, 6, 7);
     }

     int total_fn(int num, ...)
     {
          va_list ap;
          int cnt = 0;
          int total = 0;

          va_start(ap, num);
          while(++cnt <= num)
               total += va_arg(ap, int);
          va_end(ap);
          return(total);
}
```

vsprintf Function

Function

Creates text according to a specified format and writes that text to a character string.

Format

#include <stdio.h>

int vsprintf(char **buffer*, const char **format*, va_list *arglist*);

int vsprintf_nn(char __near **buffer*, const char __near **format*, va_list *arglist*);

int vsprintf_nf(char __near **buffer*, const char __far **format*, va_list *arglist*);

int vsprintf_fn(char __far **buffer*, const char __near **format*, va_list *arglist*);

int vsprintf_ff(char __far **buffer*, const char __far **format*, va_list *arglist*);

buffer	Buffer to hold the character string
format	Format specification
arglist	Pointer to list of arguments

Description

The vsprintf function operates identically to sprintf except that it does not take a variable argument list, but rather takes a pointer (*arglist*) to an argument list. It converts those arguments according to the conversion specifications in the format string specified by *format*, and writes the text to the character string specified by *buffer*. It writes a null character at the end of the character string.

See the documentation for sprintf for details on the conversion specifications.

Return value

Returns the number of bytes output to *buffer*. Note that this count does not include the null character written at the end of the string. The vsprintf function returns EOF if an error occurs.

See also

sprintf va_arg va_end va_start

Sample program

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

int inum;
double dnum;
char buf[50];

void main(void)
{
     inum = 127;
     dnum = 123.45;

     vsp(buf, "%d %f %s", inum, dnum, "Hello !!");
}

int vsp(char *s, char *fmt, ...)
{
     va_list ap;
     int cnt;

     va_start(ap, fmt);
     cnt = vsprintf(s, fmt, ap);
     va_end(ap);
     return(cnt);
}
```

4. Standard I/O Routine Reference

4.1 Standard I/O Routines

The standard I/O routines are routines that perform input or output processing using standard I/O. Some of the library routines documented in this chapter take pointer to a stream (FILE *) as an argument. However, the only streams that may be passed are the three streams stdin, stdout, and stderr. Note that these FILE pointers are all handled as NEAR pointers.

4.1.1 Standard I/O streams

The following file numbers are allocated to the standard I/O streams. These are opened during the program startup routines that run before main() is called.

Name	Macro	File number
Standard input	stdin	0
Standard output	stdout	1
Standard error	stderr	2

4.2 Library Reference

This section documents the routines in the RTLU8 runtime library that handle standard I/O. See section 1.9, "Using the Runtime Library Reference" for details on the format of this reference.

fflush Function

Function

Flushes a stream

Format

```
#include  <stdio.h>
int  fflush( FILE __near * stream );
```

stream Pointer to a stream

Description

The fflush function flushes the buffer by writing the contents of the buffer to the *stream*, if the file bound to the stream specified by stream is open for output.

Return value

Returns 0 if the buffer was flushed. However, if the specified stream is open in read-only mode, fflush returns 0 immediately. Returns EOF if an error occurred.

See also

None.

Sample program

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

static char reply[80];
static char buf[BUFSIZ];

void main(void)
{
     fprintf(stderr, "If you want to finish then
                         enter a string \"quit\"\n\n");
     for( ; ; )
     {
          fprintf(stderr, "Enter a string : ");
          fflush(stderr);
          gets(reply);
          if(!strcmp(reply, "quit"))
               break;
     }
}
```

fgetc Function

Function

Reads a character from a stream

Format

#include <stdio.h>

int fgetc(FILE __near * *stream*);

stream Pointer to a stream

Description

The fgetc function gets the next character from the specified input stream.

Return value

If it succeeds, fgetc returns the character read converted to type int without sign extension. Returns EOF if end-of-file was reached or if an error was detected.

See also

fputc getc getchar ungetc

Sample program

```
#include <stdio.h>

void main(void)
{
     int c;

     printf("Input a character : ");
     c = fgetc(stdin);
     printf("The character was : '%c' (%02x)\n", c, c);
}
```

fgets Function

Function

Reads a character string from a stream

Format

#include <stdio.h>

char *fgets(char *s, int *n*, FILE __near * *stream*);

char __near *fgets_n(char __near *s, int *n*, FILE __near * *stream*);

char __far *fgets_f(char __far *s, int *n*, FILE __near * *stream*);

s	Pointer to an area to hold the character string
n	Number of characters to read
stream	Pointer to a stream

Description

The fgets function reads a character string from *stream* and stores it in *s*. The read operation terminates when either *n*-1 characters have been read or when a newline is read. The fgets function stores a newline character at the end of *s*. A terminating null character ('\0') is added as the last character of the input string *s*.

Return value

If it succeeds, fgets returns *s*. Returns NULL if either the file is at end-of-file or a file error occurred.

See also

fputs gets

Sample program

```
#include <stdio.h>

void main(void)
{
     char buf[80];

     printf("Input a string : ");
     fgets(buf, 80, stdin);
     printf("The string was : %s\n", buf);
}
```

fprintf Function

Function

Performs formatted output to a stream.

Format

```
#include <stdio.h>
int fprintf( FILE __near * stream, const char __near * format [, argument, ... ] );
int fprintf_n( FILE __near * stream, const char __near * format [, argument, ... ] );
int fprintf_f( FILE __near * stream, const char __far * format [, argument, ... ] );
```

stream Pointer to a stream

format Format character string

argument Arguments corresponding to the conversion specifications

Description

The fprintf function accepts a sequence of arguments, associates those arguments with the conversion specifications in the format character string specified by *format*, and outputs the formatted data to *stream*. There must be exactly the same number of conversion specifications as there are arguments provided.

See the documentation on sprintf in chapter 3 for details on the conversion specifications and other aspects of formatted output.

Important!

Optional arguments that reference addresses must all use the same data model as the format string.

See the documentation on sprintf in chapter 3 for details.

Return value

Returns the number of bytes output. Returns EOF if an error occurs.

See also

fscanf printf putc sprintf

Sample program

```
#include <stdio.h>

void main(void)
{
     fprintf(stdout, "integer : %d\ncharacter : %c\n", 123, 'A');
}
```

fputc Function

Function

Outputs a single character to a stream.

Format

#include <stdio.h>

int fputc(int c, FILE __near * *stream*);

c Character to output

stream Pointer to a stream

Description

The fputc function outputs the character *c* to the specified stream.

Return value

If it succeeds, fputc returns the character *c*. Returns EOF if an error occurred.

See also

fgetc putc

Sample program

```
#include <stdio.h>

char s[ ] = "This is a test.\n";

void main(void)
{
     int i;

     for(i = 0; s[i] != '\0'; i++)
     fputc(s[i], stdout);
}
```

fputs Function

Function

Outputs a character string to a stream.

Format

#include <stdio.h>

int fputs(const char * *s*, FILE __near * *stream*);

int fputs_n(const char __near * *s*, FILE __near * *stream*);

int fputs_f(const char __far * *s*, FILE __near * *stream*);

s Character string to output

stream Pointer to a stream

Description

The fputs function outputs the null-terminated character string *s* to the specified output stream. The fputs function does not add a newline to the output, and does not output terminating null character in the string.

Return value

If it succeeds, fputs returns a non-negative value. Returns EOF if an error occurred.

See also

fgets gets puts

Sample program

```
#include <stdio.h>

void main(void)
{
     fputs("This is a test.\n", stdout);
}
```

fread Function

Function

Reads data from a stream

Format

```
#include <stdio.h>
size_t fread( void *buffer, size_t size, size_t n, FILE __near * stream );
size_t fread_n( void __near *buffer, size_t size, size_t n, FILE __near * stream );
size_t fread_f( void __far *buffer, size_t size, size_t n, FILE __near * stream );
```

buffer	Area to hold the data read
size	Size of a single item
n	Maximum number of items to read
stream	Pointer to a stream

Description

The fread function reads up to *n* items of a size in bytes given by *size* from the input stream specified by *stream*, and stores those items in *buffer*. The FILE pointer is incremented by exactly the number of bytes read.

Return value

Returns the number of data items read.

See also

fwrite

Sample program

```
#include <stdio.h>

void main(void)
{
     int i, cnt;
     char buf[80];

     cnt = fread(buf, sizeof( char ), 80, stdin);
     printf("Contents of 'buf' : ");
     for(i = 0; i < cnt; ++i)
     {
          printf("%02x ", buf[i]);
     }
}
```

fscanf Function

Function

Scans and formats input from an input stream.

Format

```
#include <stdio.h>
int fscanf( FILE __near * stream, const char * format [, address, ... ] );
int fscanf_n( FILE __near * stream, const char __near * format [, address, ... ] );
int fscanf_f( FILE __near * stream, const char __far * format [, address, ... ] );
```

stream Pointer to a stream

format Format character string

address Arguments corresponding to the conversion specifications

Description

The fscanf function scans a sequence of input fields from a stream, reading them in one character at a time. Then, it formats each field according to conversion specifications in the format character string given in *format*. Finally, fscanf stores the formatted input at the addresses given by the optional arguments following *format*. The number of format specifications and the number of addresses given, must match the number of input fields.

The fscanf function may terminates its scan of certain fields before the normal field termination character (space). It also may terminate its input operation for a variety of reasons.

See the documentation on sscanf in chapter 3 for details on the conversion specifications and other aspects of formatted output.

Important!

Optional arguments that reference addresses must all use the same data model as the format string.

See the documentation on sscanf in chapter 3 for details.

Return value

Returns the number of fields correctly scanned, converted, and stored. Values that were not stored are not included in the return value.

See also

fprintf scanf sscanf

Sample program

```
#include <stdio.h>

void main(void)
{
     int i;

     printf("Input an integer : ");
     if(fscanf(stdin, "%d", &i))
          printf("The integer : %d\n", i);
     else
          printf("Cannot read an integer\n");
}
```

fwrite Function

Function

Writes data to a stream.

Format

#include <stdio.h>

size_t fwrite(const void ∗*buffer*, size_t *size*, size_t *n*, FILE __near ∗ *stream*);

size_t fwrite_n(const void __near ∗*buffer*, size_t *size*, size_t *n*, FILE __near ∗ *stream*);

size_t fwrite_f(const void __far ∗*buffer*, size_t *size*, size_t *n*, FILE __near ∗ *stream*);

buffer	Pointer to the data to be written.
size	Size of a single item
n	Maximum number of items to write
stream	Pointer to a stream

Description

The fwrite function writes up to *n* items, each of *size* bytes, from an area given by *buffer* to *stream*. The FILE pointer attached to the *stream* is incremented by exactly the number of bytes written.

Return value

Returns the number of data items actually written. If an error occurs, the value may be less than the value specified for *n*.

See also

fread

Sample program

```
#include <stdio.h>

void main(void)
{
     int cnt;
     char buf[80];

     cnt = fread(buf, sizeof(char), 80, stdin);
     fwrite(buf, sizeof(char), cnt, stdout);
}
```

getc Macro/function

Function

Reads a character from a stream.

Format

#include <stdio.h>

int getc(FILE __near * *stream*);

stream Pointer to a stream

Description

The getc function reads one character from the specified input stream and increments the FILE pointer for that stream to point to the next character.

Return value

Returns the character read as type int without sign extension. Returns EOF on end of file or error.

See also

fgetc getchar gets putc putchar ungetc

Sample program

```
#include <stdio.h>

void main(void)
{
     int c;

     printf("Input a character : ");
     c = getc(stdin);
     printf("The character was : '%c' (%02x)\n", c, c);
}
```

getchar Macro/function

Function

Reads a character from standard input (stdin).

Format

#include <stdio.h>

int getchar(void);

Description

The getchar function reads one character from the stdin input stream. The getchar function is identical to getc(stdin).

Return value

Returns the character read as type int without sign extension. Returns EOF on end of file or error.

See also

fgetc getc gets putc putchar scanf ungetc

Sample program

```
#include <stdio.h>

void main(void)
{
     int c;

     printf("Input a character : ");
     c = getchar( );
     printf("The character was : '%c' (%02x)\n", c, c);
}
```

gets Function

Function

Reads a character string from standard input (stdin).

Format

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

s Pointer to an area to hold the character string

Description

The gets function reads a character string terminated by a newline character from the standard input stream, stdin and stores that string in *s*. The newline character is replaced by a null character in *s*.

The gets function can read input character strings that include white space characters (space and tab) without problem. When the gets function encounters a newline, it stops reading and copies all the characters read so far to *s*.

Return value

If it succeeds, gets returns the character *s*. Returns EOF if an error occurred.

See also

fgets fputs getc puts scanf

Sample program

```
#include <stdio.h>

void main(void)
{
     char buf[80];

     printf("Input a string : ");
     gets(buf);
     printf("The string was : %s\n", buf);
}
```

printf Function

Function

Writes formatted output to standard output.

Format

```
#include <stdio.h>
int printf( const char * format [, argument, ...] );
int printf_n( const char __near * format [, argument, ...] );
int printf_f( const char __far * format [, argument, ...] );
```

format Format character string

argument Arguments corresponding to the conversion specifications

Description

The printf function applies the conversion specifications in the format character string specified by *format* to the corresponding arguments and outputs the formatted data to standard output. The number of conversion specifications must be exactly the same as the number of optional arguments.

See the documentation on sprintf in chapter 3 for details on the conversion specifications and other aspects of formatted output.

Important!

Optional arguments that reference addresses must all use the same data model as the format string.

See the documentation on sprintf in chapter 3 for details.

Return value

Returns the number of bytes output. Returns EOF if an error occurs.

See also

fprintf fscanf putc puts scanf sprintf vprintf vsprintf

Sample program

```
#include <stdio.h>

void main(void)
{
     printf("integer : %d\n"
     "floating point : %f\n"
     "character : %c\n", 1234, 3.14, 'A');
}
```

putc

Macro/function

Function

Outputs a single character to a stream.

Format

#include <stdio.h>

int putc(int *c*, FILE __near * *stream*);

c	Character to output
stream	Pointer to a stream

Description

The putc function outputs the character *c* to the stream specified by *stream*.

Return value

If it succeeds, putc returns the character *c*. Returns EOF if an error occurred.

See also

fprintf fputc fputs getc getchar printf putchar

Sample program

```
#include <stdio.h>

char s[ ] = "This is a test.\n";

void main(void)
{
     char *p = s;

     while(*p != '\0')
     putc(*p++, stdout);
}
```

putchar

Macro/function

Function

Outputs a character to standard output (stdout).

Format

#include <stdio.h>

int putchar(int *c*);

c Character to output

Description

The putchar function outputs the character *c* to standard output. The call putchar(c) is identical to the call putc(c, stdout).

Return value

If it succeeds, putchar returns the character *c*. Returns EOF if an error occurred.

See also

getc getchar printf putc puts

Sample program

```
#include <stdio.h>

const char s[ ] = "This is a test.\n";

void main(void)
{
     const char *p = s;

     while(*p != '\0')
     putchar(*p++);
}
```

puts Function

Function

Outputs a character string to the standard output stream (stdout).

Format

```
#include  <stdio.h>
int  puts( const char * s );
int  puts_n( const char __near * s );
int  puts_f( const char __far * s );
```

s Character string to output

Description

The puts function outputs the null terminated character string *s* to the standard output stream (stdout) and outputs a newline at the end of that character string.

Return value

If it succeeds, puts returns a non-negative value. Returns EOF if an error occurred.

See also

fputs gets printf putchar

Sample program

```
#include <stdio.h>

void main(void)
{
     puts("This is a test.");
}
```

scanf Function

Function

Scans and formats input from the standard input stream.

Format

```
#include <stdio.h>
int scanf( const char * format [, address , ... ] );
int scanf_n( const char __near * format [, address , ... ] );
int scanf_f( const char __far * format [, address , ... ] );
```

format Format character string

address Arguments corresponding to the conversion specifications

Description

The scanf function scans a sequence of input fields from the stream, reading them in one character at a time from the standard input stream (stdin). Then, it formats each field according to conversion specifications in the format character string given in *format*. Finally, scanf stores the formatted input at the addresses given by the optional arguments following *format*. The number of conversion specifications and the number of addresses given, must match the number of input fields.

See the documentation on sscanf in chapter 3 for details on the conversion specifications and other aspects of formatted output.

Important!

Optional arguments that reference addresses must all use the same data model as the format string.

See the documentation on sscanf in chapter 3 for details.

Return value

Returns the number of fields correctly scanned, converted, and stored. Values that were not stored are not included in the return value.

If scanf reads to the end of the file, it returns EOF. If no fields were stored, it returns 0.

See also

fscanf getc printf sscanf

Sample program

```
#include <stdio.h>

void main(void)
{
     int i;

     printf("Input an interger : ");
     if(scanf("%d", &i))
          printf("The integer : %d\n", i);
     else
          printf("Cannot read an integer\n");
}
```

ungetc Function

Function

Pushes a single character back onto the input stream.

Format

#include <stdio.h>

int ungetc(int *c*, FILE __near * *stream*);

c	Character to push back
stream	Pointer to a stream

Description

The ungetc function returns (pushes back) the character *c* to the originally specified input stream *stream*. This *stream* must be open for read. This character will be returned the next time *stream* is read by getc or fread. A single character can be pushed back in any circumstances. If ungetc is called for a second time without calling getc, the previously pushed back character will be lost.

Return value

If it succeeds, ungetc returns the pushed back character. Returns EOF if the operation fails.

See also

getc

Sample program

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

void main(void)
{
     int i = 0;
     int c;

     printf("Input an integer : ");
     while((c = getchar()) != '\n' && isdigit(c))
          i = 10 * i + c - '0';
     ungetc(c, stdin);
     printf("i : %d, push back character : %c\n", i, getchar());
}
```

vfprintf Function

Function

Writes formatted output to a stream.

Format

```
#include <stdio.h>

int vfprintf( FILE __near * stream, const char * format, va_list arglist );

int vfprintf_n( FILE __near * stream, const char __near * format, va_list arglist );

int vfprintf_f( FILE __near * stream, const char __far * format, va_list arglist );
```

stream Pointer to a stream

format Format character string

arglist Pointer to a list of arguments

Description

The vfprintf function operates identically to the fprintf function except that it takes a pointer to a list of arguments instead of a variable argument list.

The vfprintf function takes a pointer to a list of arguments, applies the conversion specifications in the format character string specified by *format*, and outputs the formatted data to the stream. There must be exactly the same number of conversion specifications as there are arguments provided.

See the documentation on sprintf in chapter 3 for details on the conversion specifications and other aspects of formatted output.

Important!

Optional arguments that reference addresses must all use the same data model as the format string.

See the documentation on sprintf in chapter 3 for details.

Return value

Returns the number of bytes output. Returns EOF if an error occurs.

See also

fprintf va_arg va_end va_start vprintf vsprintf

Sample program

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

int vfprn(char * fmt, ...)
{
     va_list ap;
     int cnt;

     va_start(ap, fmt);
     cnt = vfprintf(stdout, fmt, ap);
     va_end(ap);
}

void main(void)
{
     vfprn("integer : %d\n"
     "floating point : %f\n"
     "character : %c\n", 1234, 3.14, 'A');
}
```

vprintf Function

Function

Writes formatted output.

Format

```
#include <stdio.h>
int vprintf( const char * format, va_list arglist );
int vprintf_n( const char __near * format, va_list arglist );
int vprintf_f( const char __far * format, va_list arglist );
```

format Format character string

arglist Pointer to a list of arguments

Description

The vprintf function operates identically to the printf function except that it takes a pointer to a list of arguments instead of a variable argument list.

The vprintf function takes a pointer to a list of arguments, applies the conversion specifications in the format character string specified by *format*, and outputs the formatted data to the standard output stream. There must be exactly the same number of conversion specifications as there are arguments provided.

See the documentation on sprintf in chapter 3 for details on the conversion specifications and other aspects of formatted output.

Important!

Optional arguments that reference addresses must all use the same data model as the format string.

See the documentation on sprintf in chapter 3 for details.

Return value

Returns the number of bytes output. Returns EOF if an error occurs.

See also

printf va_arg va_end va_start vfprintf vsprintf

Sample program

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

int vprn(const char * fmt, ...)
{
     va_list ap;
     int cnt;

     va_start(ap, fmt);
     cnt = vprintf(fmt, ap);
     va_end(ap);
}

void main(void)
{
     vprn("integer : %d\n"
     "floating point : %f\n"
     "character : %c\n", 1234, 3.14, 'A');
}
```

5. Low-Level Functions

5.1 What is the Low-level function?

The term low-level function as used here refers to functions that depend on the hardware and are normally called from the library routines, not user code. Since the routines documented in Chapter 4, "Standard I/O Routine Reference" in the "RTLU8 Runtime Library Reference" use the low-level routines, the low-level functions must be specified at link time. The standard I/O routines require the low-level routines listed in the table below.

Standard I/O routine	Required low-level function
fgetc , fgets , fscanf , getc , getchar , gets , scanf	read
fflush , fprintf , fputc , fputs , fwrite , printf , putc , putchar , puts , vfprintf , vprintf	write

Since the low-level routines (read and write) depend on the hardware in the system, you will need to write these routines to work in your environment.

If you write new low-level routines, the resultant code must meet the specification provided in section 5.2, "Low-Level Routine Specifications."

5.2 Low-Level Routine Specifications

read

Low-level function

Function

Reads in data.

Format

int read(int *handle*, unsigned char ∗*buffer*, unsigned int *len*);

handle	Handle that references an open file.
buffer	Area to hold the read data
len	Maximum number of bytes to read

Description

The read function reads in *len* bytes of data from the I/O port corresponding to the file specified by *handle* and stores that data in the area specified by *buffer*.

Return value

Returns the number of bytes actually read.

See also

write

write

Low-level function

Function

Writes data.

Format

int write(int *handle*, unsigned char ∗*buffer*, unsigned int *len*);

handle	Handle that references an open file.
buffer	Data to write
len	Number of bytes to write

Description

The write function writes *len* bytes of data from the area specified by *buffer* to the I/O port corresponding to the file specified by *handle*.

Return value

Returns the number of bytes actually written.

See also

read

Appendix

Library Routines Corresponding to the Data Memory Models

This table presents the ANSI/ISO 9899 C standard library routines and the prototypes of the versions of those routines that correspond to the different memory models.

Routine	Prototypes

atof	double atof(const char *s);
	double atof_n(const char __near *s);
	double atof_f(const char __far *s);

atoi	int atoi(const char *s);
	int atoi_n(const char __near *s);
	int atoi_f(const char __far *s);

atol	long atol(const char *s);
	long atol_n(const char __near *s);
	long atol_f(const char __far *s);

bsearch	void *bsearch(const void *key, const void *base, size_t nelem, size_t size, int (*cmp)(const void *elem1, const void *elem2)); void __near *bsearch_nn(const void __near *key, const void __near *base, size_t nelem, size_t size, int (*cmp_nn)(const void __near *elem1, const void __near *elem2)); void __far *bsearch_nf(const void __near *key, const void __far *base, size_t nelem, size_t size, int (*cmp_nf)(const void __near *elem1, const void __far *elem2)); void __near *bsearch_fn(const void __far *key, const void __near *base, size_t nelem, size_t size, int (*cmp_fn)(const void __far *elem1, const void __near *elem2)); void __far *bsearch_ff(const void __far *key, const void __far *base, size_t nelem, size_t size, int (*cmp_ff)(const void __far * elem1, const void __far * elem2));

calloc	void *calloc(size_t nelem, size_t size);
	void __near *calloc_n(size_t nelem, size_t size);
	void __far *calloc_f(size_t nelem, size_t size);

fgets	char *fgets(char *s, int n, FILE __near *stream);
	char __near *fgets_n(char __near *s, int n, FILE __near *stream);
	char __far *fgets_f(char __far *s, int n, FILE __near *stream);

fprintf	int __noreg fprintf(FILE __near *stream, const char *format [, argument, ...]);
	int __noreg fprintf_n(FILE __near *stream, const char __near *format [, argument, ...]);
	int __noreg fprintf_f(FILE __near *stream, const char __far *format [, argument, ...]);

Routine	Prototypes

fputs	int fputs(char *s, FILE __near *stream);
	int fputs_n(char __near *s, FILE __near *stream);
	int fputs_f(char __far *s, FILE __near *stream);

fread	size_t fread(void *buffer, size_t size, size_t n, FILE __near *stream);
	size_t fread_n(void __near *buffer, size_t size, size_t n, FILE __near *stream);
	size_t fread_f(void __far *buffer, size_t size, size_t n, FILE __near *stream);

free	void free(void *ptr);
	void free_n(void __near *ptr);
	void free_f(void __far *ptr);

frexp	double frexp(double x, int *pexep);
	double frexp_n(double x, int __near *pexep);
	double frexp_f(double x, int __far *pexep);

fscanf	int __noreg fscanf(FILE __near *stream, const char *format [, address, ...]);
	int __noreg fscanf_n(FILE __near *stream, const char __near *format [, address, ...]);
	int __noreg fscanf_f(FILE __near *stream, const char __far *format [, address, ...]);

fwrite	size_t fwrite(const void * buffer, size_t size, size_t n, FILE __near * stream);
	size_t fwrite_n(const void __near * buffer, size_t size, size_t n, FILE __near * stream);
	size_t fwrite_f(const void __far * buffer, size_t size, size_t n, FILE __near * stream);

gets	char *gets(char *s);
	char __near *gets(char __near *s);
	char __far *gets(char __far *s);

longjmp	void longjmp(jmp_buf env, int value);
	void longjmp_n(jmp_buf_n env, int value);
	void longjmp_f(jmp_buf_f env, int value);

malloc	void *malloc(size_t size);
	void __near *malloc_n(size_t size);
	void __far *malloc_f(size_t size);

memchr	void *memchr(const void *region, int c, size_t count);
	void __near *memchr_n(const void __near *region, int c, size_t count);
	void __far *memchr_f(const void __far *region, int c, size_t count);

Routine	Prototypes

memcmp	int memcmp(const void *_region1_, const void *_region2_, size_t _count_);
	int memcmp_nn(const void __near *_region1_, const void __near *_region2_, size_t _count_);
	int memcmp_nf(const void __near *_region1_, const void __far *_region2_, size_t _count_);
	int memcmp_fn(const void __far *_region1_, const void __near *_region2_, size_t _count_);
	int memcmp_ff(const void __far *_region1_, const void __far *_region2_, size_t _count_);

memcpy	void *memcpy(void *_dest_, const void *_src_, size_t _count_);
	void __near *memcpy_nn(void __near *_dest_, const void __near *_src_, size_t _count_);
	void __near *memcpy_nf(void __near *_dest_, const void __far *_src_, size_t _count_);
	void __far *memcpy_fn(void __far *_dest_, const void __near *_src_, size_t _count_);
	void __far *memcpy_ff(void __far *_dest_, const void __far *_src_, size_t _count_);

memmove	void *memmove(void *_dest_, const void *_src_, size_t _count_);
	void __near *memmove_nn(void __near *_dest_, const void __near *_src_, size_t _count_);
	void __near *memmove_nf(void __near *_dest_, const void __far *_src_, size_t _count_);
	void __far *memmove_fn(void __far *_dest_, const void __near *_src_, size_t _count_);
	void __far *memmove_ff(void __far *_dest_, const void __far *_src_, size_t _count_);

memset	void *memset(void *_region_, int _c_, size_t _count_);
	void __near *memset_n(void __near *_region_, int _c_, size_t _count_);
	void __far *memset_f(void __far *_region_, int _c_, size_t _count_);

modf	double modf(double _x_, double *_pint_);
	double modf_n(double _x_, double __near *_pint_);
	double modf_f(double _x_, double __far *_pint_);

printf	int __noreg printf(const char *_format_ [, _argument_, ...]);
	int __noreg printf_n(const char __near *_format_ [, _argument_, ...]);
	int __noreg printf_f(const char __far *_format_ [, _argument_, ...]);

puts	int puts(const char *_s_);
	int puts_n(const char __near *_s_);
	int puts_f(const char __far *_s_);

qsort	void qsort(void *_base_, size_t _n_, size_t _size_, int (*_cmp_)(const void *_elem1_, const void *_elem2_)); void qsort_n(void __near *_base_, size_t _n_, size_t _size_, int (*_cmp_nn_)(const void __near *_elem1_, const void __near *_elem2_)); void qsort_f(void __far *_base_, size_t _n_, size_t _size_, int (*_cmp_ff_)(const void __far *_elem1_, const void __far *_elem2_));

realloc	void *realloc(void *_ptr_, size_t _size_);
	void __near *realloc_n(void __near *_ptr_, size_t _size_);
	void __far *realloc_f(void __far *_ptr_, size_t _size_);

Routine	Prototypes

scanf	int __noreg scanf(const char *_format_ [, _address_, ...]);
	int __noreg scanf_n(const char __near *_format_ [, _address_, ...]);
	int __noreg scanf_f(const char __far *_format_ [, _address_, ...]);

setjmp	int setjmp(jmp_buf _env_);
	int setjmp_n(jmp_buf_n _env_);
	int setjmp_f(jmp_buf_f _env_);

sprintf	int __noreg sprintf(char *_buffer_, const char *_format_ [, _argument_, ...]);
	int __noreg sprintf_nn(char __near *_buffer_, const char __near *_format_ [, _argument_, ...]);
	int __noreg sprintf_nf(char __near *_buffer_, const char __far *_format_ [, _argument_, ...]);
	int __noreg sprintf_fn(char __far *_buffer_, const char __near *_format_ [, _argument_, ...]);
	int __noreg sprintf_ff(char __far *_buffer_, const char __far *_format_ [, _argument_, ...]);

sscanf	int __noreg sscanf(const char *_string_, cont char *_format_ [, _address_, ...]); int __noreg sscanf_nn(const char __near *_string_, const char __near *_format_ [, _address_, ...]); int __noreg sscanf_nf(const char __near *_string_, const char __far *_format_ [, _address_, ...]); int __noreg sscanf_fn(const char __far *_string_, const char __near *_format_ [, _address_, ...]); int __noreg sscanf_ff(const char __far *_string_, const char __far *_format_ [, _address_, ...]);

strcat	char *strcat(char *_string1_, const char *_string2_);
	char __near *strcat_nn(char __near *_string1_, const char __near *_string2_);
	char __near *strcat_nf(char __near *_string1_, const char __far *_string2_);
	char __far *strcat_fn(char __far *_string1_, const char __near *_string2_);
	char __far *strcat_ff(char __far *_string1_, const char __far *_string2_);

strchr	char *strchr(const char *_string_, int _c_);
	char __near *strchr_n(const char __near *_string_, int _c_);
	char __far *strchr_f(const char __far *_string_, int _c_);

strcmp	int strcmp(const char *_string1_, const char *_string2_);
	int strcmp_nn(const char __near *_string1_, const char __near *_string2_);
	int strcmp_nf(const char __near *_string1_, const char __far *_string2_);
	int strcmp_fn(const char __far *_string1_, const char __near *_string2_);
	int strcmp_ff(const char __far *_string1_, const char __far *_string2_);

Routine	Prototypes

strcpy	char *strcpy(char **string1*, const char **string2*);
	char __near *strcpy_nn(char __near **string1*, const char __near **string2*);
	char __near *strcpy_nf(char __near **string1*, const char __far **string2*);
	char __far *strcpy_fn(char __far **string1*, const char __near **string2*);
	char __far *strcpy_ff(char __far **string1*, const char __far **string2*);

strcspn	size_t strcspn(const char **sting1*, const char **string2*);
	size_t strcspn_nn(const char __near **sting1*, const char __near **string2*);
	size_t strcspn_nf(const char __near **sting1*, const char __far **string2*);
	size_t strcspn_fn(const char __far **sting1*, const char __near **string2*);
	size_t strcspn_ff(const char __far **sting1*, const char __far **string2*);

strlen	size_t strlen(const char **string*). size_t strlen_n(const char __near **string*). size_t strlen_f(const char __far **string*).

strncat	char *strncat(char **string1*, const char **string2*, size_t *count*);
	char __near *strncat_nn(char __near **string1*, const char __near **string2*, size_t *count*);
	char __near *strncat_nf(char __near **string1*, const char __far **string2*, size_t *count*);
	char __far *strncat_fn(char __far **string1*, const char __near **string2*, size_t *count*);
	char __far *strncat_ff(char __far **string1*, const char __far **string2*, size_t *count*);

strncmp	int strncmp(const char **string1*, const char **string2*, size_t *count*);
	int strncmp_nn(const char __near **string1*, const char __near **string2*, size_t *count*);
	int strncmp_nf(const char __near **string1*, const char __far **string2*, size_t *count*);
	int strncmp_fn(const char __far **string1*, const char __near **string2*, size_t *count*);
	int strncmp_ff(const char __far **string1*, const char __far **string2*, size_t *count*);

strncpy	char *strncpy(char **string1*, const char **string2*, size_t *count*);
	char __near *strncpy_nn(char __near **string1*, const char __near **string2*, size_t *count*);
	char __near *strncpy_nf(char __near **string1*, const char __far **string2*, size_t *count*);
	char __far *strncpy_fn(char __far **string1*, const char __near **string2*, size_t *count*);
	char __far *strncpy_ff(char __far **string1*, const char __far **string2*, size_t *count*);

strpbrk	char *strpbrk(const char **string1*, const char **string2*);
	char __near *strpbrk_nn(const char __near **string1*, const char __near **string2*);
	char __near *strpbrk_nf(const char __near **string1*, const char __far **string2*);
	char __far *strpbrk_fn(const char __far **string1*, const char __near **string2*);
	char __far *strpbrk_ff(const char __far **string1*, const char __far **string2*);

strrchr	char *strrchr(const char **string*, int *c*);
	char __near *strrchr(const char __near **string*, int *c*);
	char __far *strrchr(const char __far **string*, int *c*);

Routine	Prototypes

strspn	size_t strspn(const char *_string1_, const char *_string2_);
	size_t strspn_nn(const char __near *_string1_, const char __near *_string2_);
	size_t strspn_nf(const char __near *_string1_, const char __far *_string2_);
	size_t strspn_fn(const char __far *_string1_, const char __near *_string2_);
	size_t strspn_ff(const char __far *_string1_, const char __far *_string2_);

strstr	char *strstr(const char *_string1_, const char *_string2_);
	char __near *strstr_nn(const char __near *_string1_, const char __near *_string2_);
	char __near *strstr_nn(const char __near *_string1_, const char __far *_string2_);
	char __far *strstr_nn(const char __far *_string1_, const char __near *_string2_);
	char __far *strstr_ff(const char __far *_string1_, const char __far *_string2_);

strtod	double strtod(const char *_s_, char **_endptr_);
	double strtod_n(const char __near *_s_, char __near * __near *_endptr_);
	double strtod_f(const char __far *_s_, char __far * __far *_endptr_);

strtok	char *strtok(char *_string1_, const char *_string2_);
	char __near *strtok_nn(char __near *_string1_, const char __near *_string2_);
	char __near *strtok_nf(char __near *_string1_, const char __far *_string2_);
	char __far *strtok_fn(char __far *_string1_, const char __near *_string2_);
	char __far *strtok_ff(char __far *_string1_, const char __far *_string2_);

strtol	long strtol(const char *_s_, char **_endptr_, int _base_);
	long strtol_n(const char __near *_s_, char __near * __near *_endptr_, int _base_);
	long strtol_f(const char __far *_s_, char __far * __far *_endptr_, int _base_);

strtoul	unsigned long strtoul(const char *_s_, char **_endptr_, int _base_);
	unsigned long strtoul_n(const char __near *_s_, char __near * __near *_endptr_, int _base_);
	unsigned long strtoul_f(const char __far *_s_, char __far * __far *_endptr_, int _base_);

vfprintf	int __noreg vfprintf(FILE __near *_stream_, const char *_format_, va_list _arglist_);
	int __noreg vfprintf_n(FILE __near *_stream_, const char __near *_format_, va_list _arglist_);
	int __noreg vfprintf_f(FILE __near *_stream_, const char __far *_format_, va_list _arglist_);

vprintf	int __noreg vprintf(const char *_format_, va_list _arglist_);
	int __noreg vprintf_n(const char __near *_format_, va_list _arglist_);
	int __noreg vprintf_f(const char __far *_format_, va_list _arglist_);

vsprintf	int __noreg vfprintf(char *_buffer_, const char *_format_, va_list _arglist_);
	int __noreg vfprintf_nn(char __near *_buffer_, const char __near *_format_, va_list _arglist_);
	int __noreg vfprintf_nf(char __near *_buffer_, const char __far *_format_, va_list _arglist_);
	int __noreg vfprintf_fn(char __far *_buffer_, const char __near *_format_, va_list _arglist_);
	int __noreg vfprintf_ff(char __far *_buffer_, const char __far *_format_, va_list _arglist_);