OKI

# RTL665S
# Run-Time Library Reference

Program Development Support Software

FIRST EDITION
ISSUE DATE:Mar. 1997

# NOTICE

1. The information contained herein can change without notice owing to product and/or technical improvements. Before using the product, please make sure that the information being referred to is up-to-date.

2. The outline of action and examples for application circuits described herein have been chosen as an explanation for the standard action and performance of the product. When planning to use the product, please ensure that the external conditions are reflected in the actual circuit, assembly, and program designs.

3. When designing your product, please use our product below the specified maximum ratings and within the specified operating ranges including, but not limited to, operating voltage, power dissipation, and operating temperature.

4. OKI assumes no responsibility or liability whatsoever for any failure or unusual or unexpected operation resulting from misuse, neglect, improper installation, repair, alteration or accident, improper handling, or unusual physical or electrical stress including, but not limited to, exposure to parameters beyond the specified maximum ratings or operation outside the specified operating range.

5. Neither indemnity against nor license of a third party's industrial and intellectual property right, etc. is granted by us in connection with the use of the product and/or the information and drawings contained herein. No responsibility is assumed by us for any infringement of a third party's right which may result from the use thereof.

6. The products listed in this document are intended for use in general electronics equipment for commercial applications (e.g., office automation, communication equipment, measurement equipment, consumer electronics, etc.). These products are not authorized for use in any system or application that requires special or enhanced quality and reliability characteristics nor in any system or application where the failure of such system or application may result in the loss or damage of property, or death or injury to humans. Such applications include, but are not limited to, traffic and automotive equipment, safety devices, aerospace equipment, nuclear power control, medical equipment, and life-support systems.

7. Certain products in this document may need government approval before they can be exported to particular countries. The purchaser assumes the responsibility of determining the legality of export of these products and will take appropriate and necessary steps at their own expense for these.

8. No part of the contents contained herein may be reprinted or reproduced without our prior permission.

9. MS-DOS is a registered trademark of Microsoft Corporation.

# TABLE OF CONTENTS

# Introduction



# Chapter 1. Overview

# Chapter 2. Standard Built-In Routines Reference

**Library Referene** (alphabetic order)

# Chapter3. Standard Input/Output Routines Reference



# Appendix



# Addendum. Low-Level Routines

# *Introduction*

# The RTL665S Run-Time Library

RTL665S is a C run-time library for microcontrollers based on the OLMS-66K series nX-8/500S CPU core. It supplies many routines frequently used in application programming. Using these routines can save much time and effort.

In principle, the library conforms to the ANSI/ISO 9899 C standard. It allows most existing user programs written in C to be reused directly or with only minimal modification.

# The Organization of this Manual

This manual describes the RTL665S run-time library. This manual is written assuming that the reader is an experienced C programmer and is thoroughly familiar with the nX-8/500S CPU.

This manual consists of the following three chapters.

**Chapter 1. Overview**

This chapter provides an overview of the RTL665S run-time library. This chapter explains the RTL665S library file organization, the use of the library, and the difference between macros and functions. It also describes the functions that take pointers to code memory as arguments, and gives an overview of the functions of the library routines.

**Chapter 2. Standard Built-In Routines Reference**

This chapter describes in detail the standard built-in routines of the library. It is organized alphabetically by routine.

**Chapter 3. Standard Input/Output Routines Reference**

This chapter describes in detail the library routines that handle standard input/output. It is organized alphabetically by routine.

# Related Documents

Refer to the following documents as required.

- CC665S User’s Manual

  Describes the use of the CC665S C compiler and provides the language specifications.

- MAC66K Assembler Package User’s Manual

  Describes the use of the software included in the MAC66K Assembler Package and provides the language specifications for the assembly language.

- RTL665S.DOC

  Provides the latest information not included in this manual.

- SPRNS500.DOC

  Describes SPRNS50*x*.LIB, the non-floating point string conversion library.

# Typographical Conventions and Terminology

To help the reader locate, identify, and understand information easily, this manual uses visual cues and standard text formats. The following typographical conventions are used in this guide.

| Symbol | Explanation |
|---|---|
| `SAMPLE` | Messages displayed on the screen, examples of command line input, and examples of listing files to be created use this type style. |
| *Italics* | Items that are written in italics are not to be entered as typed, but rather are to be replaced by the required information in the user input. |
| **[ ]** | Items enclosed in square brackets are optional items that are entered as needed. |
| … | Three dots in a row indicate that the preceding item may be repeated as required. |
| `{choice1\|choice2}` | Items enclosed in curly braces ({ }) and separated by vertical bars indicate that one of the items is to be selected and entered. Items not surrounded by square brackets must be included exactly once in the input. |
| *value1 to value2* | Indicates a value between *value1* and *value2*, inclusive. |
| Ctrl+C | Indicates that the Ctrl key and the C key are to be pressed at the same time. |
| `PROGRAM`<br>.<br>.<br>.<br>`PROGRAM` | Vertically aligned dots indicate that a section of the program example has been omitted. |

The table below lists terms used throughout this manual and their meanings.

| Term | Meaning |
|---|---|
| Macro | A name defined by the `#define` preprocessor directive. In this document, function-like macros (i.e. macros that take parameters) are sometimes referred to simply by the term “macro”. |
| Routine | Both functions and function-like macros are referred to as “routines”. |
| Library routine | A routine that is included in the RTL665S run-time library. |
| Type | A name defined using `typedef`. |
| Constant macro | A macro that takes no parameters and that always expands to the same constant value. Constant macros are also referred to as simply “constants” in this document. |
| Null character | The character that has the ASCII code 0x00. That is, the character '\0'. |
| Null string | A string of length zero, that is a string whose first byte is the null character. |
| Null terminator | The null character that terminates a character string. |
| Null pointer | A pointer to the address zero. Expressed by the NULL constant macro. |

# Chapter 1

# Overview

This chapter provides a simple description of the RTL665S run-time library, including its structure, use, and the library routines it provides.

# 1.1 RTL665S Run-Time Library Organization

This section describes the files that make up the RTL665S run-time library.

The RTL665S run-time library consists of eleven header files and several library files.

## 1.1.1 Header Files

Eleven header files are provided. These files are differentiated by function. These header files include function prototype declarations, macro definitions, and type definitions.

These header files are necessary when compiling user programs. The CC665S C compiler includes the header files specified with the `#include` preprocessor directive in the source program.

The table below lists the header files and their content.

| **Header File** | **Content** |
|---|---|
| ctype.h | Character classification and conversion |
| errno.h | Error identifiers |
| float.h | Floating point limit values |
| limits.h | Integer limit values |
| math.h | Mathematical functions |
| setjmp.h | Global jump functions |
| stdarg.h | Variable arguments |
| stddef.h | Standard types and macros |
| stdio.h | Input/output processing |
| stdlib.h | General purpose utilities |
| string.h | Character string operations |

## 1.1.2 Library Files

Each library file contains all the library routines. The format of the library files is the same binary format as that of object files output by the RAS66K and RL66K programs.

The library files are required at link time. The RL66K linker searches for the library routines used in the program in a library file, and links the program and those routines together to create an absolute object file with the .ABS extension.

The library files provided for the nX-8/500S are as follows.

| **Library File** | **Memory Model** |
|---|---|
| L66KS50*x*.LIB | RTL665S run time library full set version |
| R66KS50*x*.LIB | RTL665S run time library reentrant version |
| SPRNS50*x*.LIB | Non-floating point string conversion library |

The small *x* in the library file names above varies with the memory model. The letters for the memory models available are as follows.

| ***x*** | **Memory Model** |
|---|---|
| S | Small memory model |
| E | Effective medium memory model |
| M | Medium memory model |
| C | Compact memory model |
| K | Effective large memory model |
| L | Large memory model |

When linking, specify the same memory model as that used when compiling with CC665S.

# 1.2 Compatibility with the ANSI/ISO 9899 C Standard

The RTL665S run-time library is basically a subset of the library specifications proposed in the ANSI/ISO 9899 C Standard.

The header files listed below are not included in the RTL665S run-time library.

**Standard Header Files not Supported by RTL665S**

| Header File | Content |
|---|---|
| assert.h | Execution time condition checking |
| locale.h | Locale setting and changing |
| signal.h | Signal processing functions |
| time.h | Data and time processing functions |

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

The RTL665S run-time library includes a few functions not stipulated in the ANSI/ISO 9899 C standard. These original functions are provided so that user programs can handle the independent ROM and RAM spaces that are a feature of architecture of the nX-8/500S CPU core. For further details, see the Appendix at the back of this manual.

# 1.3 Using the Library Routines

This section describes the environment setup required to use the RTL665S run-time library, and the procedures for using the library routines, from programming and compilation though linking.

## 1.3.1 Setting the INCL66K Environment Variable

The INCL66K environment variable setting provides the CC665S C compiler with the path for the directory that holds the header files. The CC665S C compiler searches for the header files specified with the `#include` preprocessor directive in source files starting with the path specified by the INCL66K environment variable.

Use the DOS SET command to set the INCL66K environment variable. The SET command has the following syntax.

`SET INCL66K=`*path*

**■ Example ■**

Use the following command line when the header files are stored in the A:\66K\INCLUDE directory.

```
SET INCL66K=A:\66K\INCLUDE
```

**■ See also ■**

The header file path can also be specified by using the CC665S C compiler's /I*path* option. For example, the path in the example above could also be specified by using the /I option as shown below.

```
CC665S /TM66589 /IA:\66K\INCLUDE FOO.C
```

## 1.3.2 Program Notation

When using a library routine, the corresponding header file must be included in the source file. The `#include` preprocessor directive is used to include required header files. The CC665S C compiler inserts the header files specified with the `#include` preprocessor directive in the source file. Refer to the library references of chapters 2 and beyond to determine which header file is required for a given library routine.

**■ Example 1 ■**

This example shows the use of the `memcpy` function. The corresponding header file for the `memcpy` function is `string.h.` Therefore, the following line must be specified in the source file.

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

The `#include` statements used to include header files can be specified in any order in the source program.

**■ Example 2 ■**

If both `string.h` and `math.h` are required, their inclusion can be specified either as:

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

or as:

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

There are two ways to specify the file name in the `#include` preprocessor directive. The first is to enclose the file name angle brackets (< >) as shown in the examples above, and the second is to enclose the name in double quotation marks (" "). Always use angle brackets to include RTL665S header files. See the "CC665S User's Manual" for a detailed description of the `#include` preprocessor directive.

## 1.3.3 The Procedure from Compilation through Linking

This section describes the procedures used from source file compilation through linking.

### 1.3.3.1 Compilation and Assembly

There is no need to be aware of whether or not library routines are used when compiling and assembling source files.

**■ Example ■**

The following commands compile and assemble the foo.c source file.

```
CC665S /TM66589 FOO.C

RAS66K FOO.ASM /CD
```

The /CD option to the RAS66K assembler is required to maintain distinction between upper and lower case letters in variable and function names in the C source program. To use the CDB665 source level debugger, specify the /SD option to the CC655S C compiler and the /CC option to the RAS66K assembler.

### 1.3.3.2 Library Linking

Following the compilation and assembly operations, the next step is the link operation using the RL66K linker to create an absolute object file. Here, in addition to the object file created by the compilation and assembly, you must also specify a startup routine and a library file.

■ **Example 1** ■

Use the following command to link the object file foo.obj.

```
RL66K FOO A:\66K\STARTUP\S66589S,,,A:\66K\LIB\L66KS50S.LIB /CC
```

In this example the S66589S.OBJ startup routine is in the A:\66K\STARTUP directory. Also, the L66KS50S.LIB library file is in the A:\66K\LIB directory.

The library file path specification can be omitted if the library file is in the path indicated by the LIB66K environment variable.

■ **Example 2** ■

The following RL66K command line would be used if the L66KS50S.LIB file were in the A:\66K\LIB directory and the LIB66K environment variable were set to A:\66K\LIB.

```
RL66K FOO A:\66K\STARTUP\S66589S,,,L66KS50S.LIB /CC
```

■ **Major Point** ■

Always specify the /CC option when linking.

Some library routines include their own initialization routine. The execution of these initialization routines is implemented by calling the subroutine with the name `_$$content_of_init` in the startup routine.

The /CC option informs the RL66K linker that these initialization routines exist. If an object file is linked without the /CC option, initialization routine linking will not be performed correctly.

# 1.4 Role of Header Files

The header files function as an interface between user programs and the library. Including the header file that corresponds to a given library routine provides the compiler with the syntax (prototype) of that library routine, as well as any constants and types used by that routine.

## 1.4.1 Inclusion of Macros, Constants, and Types

Header files must be included to define the macros, constants, and types included in the library.

The definitions of the macros, constants, and types used by library routines are included in the header files. Programmers can also use these macros, constants, and types. The definitions of these items as used by the library routines and as used by user programs must be completely identical.

In most cases, the programmer needs only be aware of the meaning of macros, constants, and types included in the header files, and need not be concerned with the details of their definitions.

■ **Example** ■

```
#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;
}
```

This example shows the use of variable arguments. Since the macros `va_start`, `va_arg`, and `va_end` and the type `va_list` are defined in `stdarg.h`, that header file is included. The programmer does not need to know the actual details of the definitions.

### 1.4.2 Inclusion of Function Prototype Declarations

The header files include specifications for the calling syntax for all functions in the library. That is, they include the specifications for the types of the arguments and for the return type. This declaration is generally referred to as a prototype declaration.

The compiler checks that the syntax of calls to library functions in user programs, i.e., the number of arguments, their type, and the return type, conforms to that of the prototype declaration in the header file. The compiler reports a warning or, in certain cases, an error, if a call does not match the function's prototype.

Compiler type checking is extremely important for program reliability. This is because syntax errors in function calls would otherwise become algorithm errors that would be difficult to discover.

**■ Example ■**

The `strlen` function is used in this example.

```
#include <string.h>

int             i;

int     func( void )
{
        int   len;

        .
        .
        .
        len = strlen( i );          /* Warning  */
        .
        .
        .

}
```

The `strlen` function's prototype in the `string.h` header file is as follows:

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

Since the variable `i` (whose type is `int`) is specified as the argument in the first call to the `strlen` function, the compiler issues a warning for this call.

The compiler is able to perform these checks because the `string.h` header file was included at the start of the program. If the `string.h` file were not included, the compiler could not perform these checks.

# 1.5 Functions and Macros

## 1.5.1 Differences between Functions and Macros

The term “library routine” as used in this document actually refers to both functions and function-like macros. The library routines included in the RTL665S run-time library are included as either functions, macros, or both. The form(s) in which each library routine is provided are documented in section 1.7, “Header File Contents,” chapter 2, “Standard Built-In Routines Reference,” and chapter 3, "Standard Input/Output Routines Reference."

Normally programs have no need to be aware of whether a routine that they use is a macro or a function. Programs only need to be aware of the differences between macros and functions in the following cases.

- Although function calls are expanded as subroutine calls, macros calls are expanded to inline code by the preprocessor. That is, a macro is faster than a function by exactly the overhead associated with a function call. However, since the same code is expanded each time a macro is called, the program size will be larger than if a function had been used.

- While a function name has meaning as an address at compile time, macro names are expanded by the preprocessor, and no longer exist at compile time. This means that a routine implemented as a macro cannot be used through a function pointer.

- Although the compiler checks function calls for type matching, it does not type check macro calls. That means that the programmer is responsible for checking the argument and return value types associated with macro calls.

## 1.5.2 Calling Routines with Macro Definitions as Functions

Some of the library routines included in the RTL665S run-time library are provided as both macros and functions. The `toupper` and related functions from `ctype.h` are examples. Routines of this type are listed as “Macro/Function” in section 1.7, “Header File Contents,” chapter 2, “Standard Built-In Routines Reference,” and chapter 3, “Standard Input/Output Routines Reference.”

Since the function prototype declaration for a routine appears before the macro definition in the header file, normally, the macro definition will be used. However, there are two methods for using the function form of such routines. The remainder of this section describes these methods.

### 1.5.2.1 Removing a Macro Definition Using `#undef`

The first method for forcing the use of the function definition of a routine is to remove the macro definition of the routine from the environment using the `#undef` preprocessor directive. Be sure to place the `#undef` preprocessor directive between the line where the header file is included using the `#include` preprocessor directive and the first line where the routine is used. The safest place is immediately following the `#include` directives.

■ **Example** ■

In this example the `#undef` directive removes the definition of the `toupper` macro from the environment.

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

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

### 1.5.2.2 Enclosing the Routine Name in Parentheses

The second method is to enclose the routine name in parentheses when calling the routine. The preprocessor recognizes a function-like macro when it sees a left parenthesis immediately following the macro name. Therefore, preprocessor macro expansion of function-like macros can be defeated by enclosing the macro name in parentheses.

■ **Example** ■

In this example the function definition of the `toupper` routine is called by enclosing the name "`toupper`" in parentheses.

```
#include     <ctype.h>

void    func( void )
{

    int     c;
    .
    .
    .
    c = (toupper) ( c );  /* The routine is called as a function.
*/
    .
    .
    .
}
```

# 1.6 Reentrant Routines

In addition to the L66KS50*x*.LIB full set library file, which includes routines for all the library routines described in this manual, RTL665S also includes the R66KS50*x*.LIB library file, which collects only the reentrant routines.

The reentrant version should be specified if the same library routine is used for both interrupt and normal processing.

See the file RTL665S.DOC to determine which library routines are reentrant, i.e., to determine if they are included in the reentrant version library file.

Some run time library routines set the global variable `errno` to an error value if they receive an incorrect value as an argument. Strictly speaking, these routines cannot be said to be reentrant. However, since there is no processing within the library routines that depends on the value of `errno`, these routines will correctly perform their intended functions even if the value of errno is overwritten during interrupt handling. Therefore, RTL665S treats functions that reset `errno` as reentrant routines.

Be careful when handling the value of `errno` when routines that set its value are used in both interrupt handling and normal processing. Interrupt routines should save `errno` on entry and restore it prior to exit if `errno` is referenced in normal processing.

**■ Example ■**

This example demonstrates the use of the `atol` routine in an interrupt handler.

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

char  data_buf[16];
long  value;

#pragma interrupt GTM_OVF_function 0X2C

void  GTM_OVF_function( void )
{
  /*
    Saves the current value of errno for normal processing.
  */
  int old_errno = errno;

 .
 .
 .
  value = atol(data_buf);
 .
 .
 .

  /*
    Restores the current value of errno for normal processing.
  */
  errno = old_errno;
}
```

# 1.7 Header File Contents

This section describes the functions, macros, global variables, constant macros, and types provided by the RTL665S run-time library.

The classification column classifies each object into one of the following.

Function
Macro
Macro/Function
Constant macro
Type
Global variable

The term “macro/function” indicates that both macro and function definitions of the routine are provided. Detailed descriptions of the functions, macros, and macro/function routines are provided in chapter 2, “Standard Built-In Routines Reference,” and chapter 3, “Standard Input/Output Routines Reference.”

### 1.7.1 Character Classification and Conversion <`ctype.h`>

The header `ctype.h` declares routines for classifying and converting single byte characters.

| Name | Classification | Description |
|---|---|---|
| isalnum | Macro/Function | Tests if a character is either a letter or a decimal digit. |
| isalpha | Macro/Function | Tests if a character is a letter. |
| iscntrl | Macro/Function | Tests if a character is a control character, i.e., is one of 0x00 to 0x1F or 7F. |
| isdight | Macro/Function | Tests if a character is a decimal digit. |
| isgraph | Macro/Function | Tests if a character is a printable character other than space (' '), i.e., if it is in the range 0x21 to 0x7E. |
| islower | Macro/Function | Tests if a character is a lower case letter. |
| isprint | Macro/Function | Tests if a character is a printable character including the space character (' '), i.e., if it is in the range 0x20 to 0x7E. |
| ispunct | Macro/Function | Tests if a character is a punctuation character, i.e., is one of 0x21 to 0x2F, 0x3A to 0x40, 0x5B to 0x60, and 0x7B to 0x7E. |
| isspace | Macro/Function | Tests if a character is a white space character, i.e., is one of 0x09 to 0x0D or space (' '). |
| isupper | Macro/Function | Tests if a character is an upper case letter. |
| isxdigit | Macro/Function | Tests if a character is a hexadecimal digit. |
| tolower | Macro/Function | Converts upper case letters to lower case letters. |
| toupper | Macro/Function | Converts lower case letters to upper case letters. |

### 1.7.2 Error Identification <`errno.h`>

The header `errno.h` includes information related to errors that occur in library routines. The global variable `errno` and constant macros for values that are assigned to `errno` are defined in `errno.h`.

| Name | Classification | Description |
|---|---|---|
| errno | Global variable | The global variable `errno` is of type `volatile int` and holds error state information. Its initial value is zero, and it is set to one of the following non-zero values according to the error state when an error occurs in a library routine. |
| EDOM | Constant macro | The EDOM constant indicates a domain error. Domain errors occur when an attempt is made to apply a mathematical function to a value outside its domain, for example calling the `asin` function with a value greater than one or less than minus one. |
| ERANGE | Constant macro | The ERANGE constant indicates an overflow error. Overflows occur when the result of a mathematical function exceeds the range of values that can be expressed in a value of type `double`. |

### 1.7.3 Floating Point Limits <float.h>

The header `float.h` defines constant macros that express limit values for floating point numbers of type `float`, `double`, and `long double`. Since the types `double` and `long double` are identical in the CC665S C compiler the limits for the `long double` type are the same as those for the `double` type.

| Name | Classification | Description |
|---|---|---|
| DBL_DIG | Constant macro | The number of digits of decimal precision provided by numbers of type `double`. |
| DBL_EPSILON | Constant macro | The smallest positive floating point number such that 1.0 + DBL_EPSILON can be differentiated from 1.0 by numbers of type `double`. |
| DBL_MANT_DIG | Constant macro | The number of bits in the fraction part of numbers of type `double`. |
| DBL_MAX | Constant macro | The largest value that can be represented by numbers of type `double`. |
| DBL_MAX_EXP | Constant macro | The largest integer such that two (2) raised to that number minus one is representable by numbers of type `double`. |
| DBL_MAX_10_EXP | Constant macro | The largest integer such that ten (10) raised to that number is representable by numbers of type `double`. |
| DBL_MIN | Constant macro | The smallest value that can be represented by numbers of type `double`. |
| DBL_MIN_EXP | Constant macro | The smallest integer n such that two (2) raised to the power n minus one is representable by numbers of type `double`. |
| DBL_MIN_10_EXP | Constant macro | The smallest integer such that ten (10) raised to that number is representable by numbers of type `double`. |
| FLT_DIG | Constant macro | The number of digits of decimal precision provided by numbers of type `float`. |
| FLT_EPSILON | Constant macro | The smallest positive floating point number such that 1.0 + FLT_EPSILON can be differentiated from 1.0 by numbers of type `float`. |
| FLT_MANT_DIG | Constant macro | The number of bits in the fraction part of numbers of type `float`. |
| FLT_MAX | Constant macro | The largest value that can be represented by numbers of type `float`. |
| FLT_MAX_EXP | Constant macro | The largest integer such that two (2) raised to that number minus one is representable by numbers of type `float`. |
| FLT_MAX_10_EXP | Constant macro | The largest integer such that ten (10) raised to that number is representable by numbers of type `float`. |
| FLT_MIN | Constant macro | The smallest value that can be represented by numbers of type `float`. |
| FLT_MIN_EXP | Constant macro | The smallest integer such that two (2) raised to that number minus one is representable by numbers of type `float`. |

| Name | Classification | Description |
|---|---|---|
| FLT_MIN_10_EXP | Constant macro | The smallest integer such that ten (10) raised to that number is representable by numbers of type `float`. |
| FLT_RADIX | Constant macro | The floating point exponent representation radix. |
| FLT_ROUNDS | Constant macro | Indicates that rounding to nearest 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 |
| 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 Integer Limits <limits.h>

The header `limits.h` defines constant macros that express limiting values for the integral types.

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

## 1.7.5 Mathematical Functions <math.h>

The header `math.h` declares several mathematical functions. All calculations are performed on objects of type `double`. Certain of these functions set the value of the global variable `errno` to an error value if an error occurs. See the descriptions of each routine in chapter 2, "Standard Built-In Routines Reference."

| Name | Classification | Description |
|---|---|---|
| HUGE_VAL | Constant macro | The maximum value that can be represented by objects of type `double`. This value is used to express infinity. |
| exp | Function | Computes the exponential function. |
| frexp | Function | Breaks a floating point number into its fraction and exponent parts. |
| ldexp | Function | Computes the product of its argument and a power of 2. |
| log | Macro/Function | Computes the natural logarithm. |
| log10 | Macro/Function | Computes the common logarithm. |
| modf | Function | Breaks a floating point number into its integral and fractional parts. |
| cosh | Function | Computes the hyperbolic cosine. |
| sinh | Function | Computes the hyperbolic sine. |
| tanh | Function | Computes the hyperbolic tangent. |
| ceil | Function | Computes the ceiling of a floating point number. |
| fabs | Function | Takes the absolute value of a floating point number. |
| floor | Function | Computes the floor of (i.e., the largest integer not greater than) a floating point number. |
| fmod | Function | Computes the remainder of two floating point numbers. |
| pow | Function | Computes the value of x raised to the y power for two floating point numbers x and y. |
| sqrt | Function | Computes the square root. |
| acos | Macro/Function | Computes the arc cosine. |
| asin | Macro/Function | Computes the arc sine. |
| atan | Function | Computes the arc tangent. |
| atan2 | Function | Computes the principle value of the arc tangent of its two arguments. The `atan2` function can be used to compute the arc tangent of a value too large to be computed by the `atan` function. |
| cos | Macro/Function | Computes the cosine. |
| sin | Macro/Function | Computes the sine. |
| tan | Function | Computes the tangent. |

### 1.7.6 Global Jump <setjmp.h>

The header `setjmp.h` includes declarations for the function that implements the global jump functionality, and definitions of a macro and a type. It is possible to jump out of the currently executing function using these routines.

| Name | Classification | Description |
|---|---|---|
| jmp_buf | Type | Global jumps are implemented by saving an environment using `setjmp`, and then restoring that environment using `longjmp`. The `jmp_buf` type represents stored environment objects. |
| setjmp | Macro | Stores the environment in argument, which must be an object of type `jmp_buf`. |
| longjmp | Function | Restores an environment saved with the `setjmp` routine. As a result, program execution transfers to the place where `setjmp` was called. |

### 1.7.7 Variable Arguments <stdarg.h>

The header `stdarg.h` includes the definitions and declarations used to implement functions that take a variable number of arguments. Using these routines it is possible to create routines that take a variable number of arguments without concern for assembly language level details.

| Name | Classification | Description |
|---|---|---|
| va_list | Type | This type is used to hold information concerning variable arguments lists. It is used by the `va_start`, `va_arg`, and `va_end` routines. |
| va_start | Macro | Prepares to reference a variable arguments list. This routine must be invoked prior to using `va_arg`. |
| va_arg | Macro | Returns the next argument value in a variable arguments list. The `va_arg` routine allows the second and later arguments to the function to be accessed sequentially. |
| va_end | Macro | Performs the clean-up activities required after referencing a variable arguments list. |

### 1.7.8 General Definitions <stddef.h>

The header `stddef.h` defines certain data types and macros that are used widely.

| Name | Classification | Description |
|---|---|---|
| NULL | Constant macro | Express a null pointer. |
| offsetof | Macro | Returns the location of a structure member as the number of bytes from the start of that structure. |
| ptrdiff_t | Type | The type `ptrdiff_t` is a signed integral type that represents the difference between two pointers. |
| size_t | Type | The type `size_t` is an unsigned integral type that represents the result of the `sizeof` operator. |

### 1.7.9 Input/Output Processing <stdio.h>

The header `stdio.h` declares routines that perform input/output processing, and includes macros and type definitions used by those routines.

| Name | Classification | Description |
|---|---|---|
| EOF | Constant macro | –1<br>Although the original meaning of EOF in the ANSI/ISO9899 C standard is end-of-file, it is also used as the error return value by RTL665. |
| FILE | Type | Type for streams. |
| stderr | Macro | Pointer to standard error stream. |
| stdin | Macro | Pointer to standard input stream. |
| stdout | Macro | Pointer to standard output stream. |
| fgetc | Function | Gets a character from a stream. |
| fgets | Function | Gets a string from a stream. |
| fprintf | Function | Sends formatted output to a stream. |
| fputc | Function | Outputs a character to a stream. |
| fputs | Function | Outputs a string to a stream. |
| fscanf | Function | Scans and formats input from an input stream. |
| getc | Macro/Function | Gets a character from a stream. |
| getchar | Macro/Function | Gets a character from the standard input. |
| gets | Function | Reads a string from the standard input. |
| printf | Function | Writes formatted output to the standard output. |
| putc | Macro/Function | Outputs a character to a stream. |

| Name | Classification | Description |
|---|---|---|
| putchar | Macro/Function | Outputs a character to the standard output |
| puts | Function | Outputs a string to the standard output. |
| scanf | Function | Scans the standard input stream, and inputs with formatting. |
| sprintf | Function | Writes formatted data as a string. |
| sscanf | Function | Reads formatted data from a string. |
| ungetc | Function | Pushes a character back in an input stream. |
| vfprintf | Function | Writes formatted output to a stream. |
| vsprintf | Function | Writes formatted data as a string. |
| vprintf | Function | Writes formatted output. |

## 1.7.10 General Utilities `<stdlib.h>`

The header `stdlib.h` defines several general purpose utility routines and macros and types used by those routines.

| Name | Classification | Description |
|---|---|---|
| div_t | Type | The type `div_t` is the structure type returned by the `div` function. It is a structure with two members of type `int` that hold the quotient and remainder. |
| ldiv_t | Type | The type `ldiv_t` is the structure type returned by the `ldiv` function. It is a structure with two members of type `long` that hold the quotient and remainder. |

| Name | Classification | Description |
|---|---|---|
| RAND_MAX | Constant macro | 32767<br>The maximum value of the pseudo-random numbers returned by the `rand` function. |
| abs | Function | Returns the absolute value of an integer value of type `int`. |
| atof | Macro/Function | Converts a character string to a floating point number of type `double`. |
| atoi | Macro/Function | Converts an integer of type `int` to a character string. |
| atol | Macro/Function | Converts an integer of type `long` to a character string. |
| bsearch | Function | Searches a sorted array for the specified item using a binary search. |
| calloc | Function | Allocates the required amount of memory. |
| div | Function | Computes the quotient and remainder of two integers of type `int`, stores the quotient and remainder in a structure of type `div_t`, and returns that structure. |
| free | Function | Releases allocated memory. |
| itoa | Function | Converts an integer of type `int` to a character string in the specified radix. |
| labs | Function | Returns the absolute value of an integer of type `long`. |
| ltoa | Function | Converts an integer of type `long` to a character string in the specified radix. |
| ldiv | Function | Computes the quotient and remainder of two integers of type `long`, stores the quotient and remainder in a structure of type `ldiv_t`, and returns that structure. |
| malloc | Function | Allocates memory. |
| qsort | Function | Sorts the elements in an array using the Quicksort algorithm. |
| rand | Function | Generates a pseudo-random number. |
| realloc | Function | Reallocates memory. |
| srand | Macro/Function | Initializes the sequence of pseudo-random numbers returned by `rand`. |
| strtod | Macro/Function | Converts a character string to a floating point number of type `double`. |
| strtol | Function | Converts a character string to an integer of type `long`. |
| strtoul | Macro/Function | Converts a character string to an integer of type `unsigned long`. |
| ultoa | Function | Converts an integer of type `unsigned long` to a character string in the specified radix. |

### 1.7.11 String Handling `<string.h>`

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

| Name | Classification | Description |
|---|---|---|
| memchr | Function | Searches a memory area for the place where a certain single byte datum first appears. |
| memcmp | Function | Compares two memory areas. |
| memcpy | Function | Copies the data in a memory area to another memory area. |
| memmove | Function | Copies the data in a memory area to another memory area. Unlike `memcpy`, `memmove` operates correctly if the two areas overlap. |
| memset | Function | Fills a fixed memory area with a specified single byte datum. |
| strcat | Function | Concatenates character strings. |
| strchr | Function | Searches a character string for the place where a certain character first appears. |
| strcmp | Function | Compares character strings. |
| strcpy | Function | Copies character strings. |
| strcspn | Function | Computes the length of the initial section of the first character string that does not include any characters from the second character string. |
| strlen | Function | Computes the length of a character string. |
| strncat | Function | Concatenates the first n bytes of a character string to the end of another character string. |
| strncmp | Function | Compares the first n bytes of two character strings. |
| strncpy | Function | Copies the first n bytes of a character string to another memory area. |
| strpbrk | Function | Searches a character string for the first occurrence of any character in another character string. |
| strrchr | Function | Searches a character string for the last occurrence of a character. |
| strspn | Function | Computes the length of the initial segment of one character string that consists of characters from the other character string. |
| strstr | Function | Searches in one character string for another character string. |
| strtok | Function | Divides a character string into tokens. |

# 1.8 Using the Run-Time Library Reference

Chapters 2 and beyond document all the routines included in the RTL665S run-time library. Each chapter lists its routines in alphabetical order.

The explanations assume the use of CC665S's /WIN option. If this option is not used, arguments that are pointers to ROM (const char *, const void *, etc.) require the use of routine variants supporting such pointers. For further details on these routines, see the appendix "Routines Accessing ROM." For further details on the /WIN option, see the CC665S User's Manual.

Chapter 2: Standard Built-In Routines Reference

**<Routine Name>** Classification

The upper left of each page lists the routines described and the upper right indicates their classification as function, macro, or macro/function.

**Function**

This sections gives a concise description of the routine's function.

**Syntax**

Indicates the header file that declares the routine(s) and gives the prototype(s) for the routine(s) and meaning of the argument(s).

**Description**

Describes the routine's function and usage in detail.

**Return value**

Specifies the return value.

**See also**

Lists related routines.

**Example**

Provides programming examples that actually use the routine. These examples are designed to show the function of the routine in an actual program. These examples are not necessarily actual application programs.

# Chapter 2

# Standard Built-In Routines Reference

This chapter describes the standard built-in routines of the RTL665S library. The routines are ordered alphabetically.

If a call to a routine includes pointers to ROM (const char *, const void *, etc.) among its arguments and the /WIN option is not specified, a special variant of the routine must be used. For further details on the naming conventions for these variants, see the appendix “Routines Accessing ROM.”

# abs

**Function**

## Function

Returns the absolute value of an integer of type `int`.

## Syntax

#include <stdlib.h>

int abs( int *n* );

*n* An integer

## Description

The `abs` function returns the absolute value of its integer argument *n*.

## Return value

The `abs` function returns an integer in the range 0 to 32767. However, if *n* is –32768 it returns –32768.

## See also

fabs labs

## Example

```
#include    <stdlib.h>

void    main( void )
{
    int n,res;

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

# acos

## Macro/Function

### Function

Computes the arc cosine of its argument.

### Syntax

#include <math.h>

double acos( double *x* );

*x* The real number value for which the arc cosine is to be computed

### Description

The `acos` routine computes the arc cosine of its argument *x*. The value of *x* must be in the range –1 to 1. If an argument with a value outside this range is passed to the `acos` routine, a domain error occurs and the global variable `errno` is set to EDOM.

### Return value

The `acos` routine returns the arc cosine of *x*, which is a value in the range 0 to π radians.

### See also

asin atan atan2 cos sin tan

### Example

```
#include    <math.h>

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

    x = 0.5;

    res = acos(x);
}
```

# asin

**Macro/Function**

### Function

Computes the arc sine of its argument.

### Syntax

#include <math.h>

double asin( double *x* );

*x* The real number value for which the arc sine is to be computed.

### Description

The `asin` routine computes the arc sine of its argument *x*. The value of *x* must be in the range –1 to 1. If an argument with a value outside this range is passed to the `asin` routine, a domain error occurs and the global variable `errno` is set to EDOM.

### Return value

The `asin` routine returns the arc sine of *x*, which is a value in the range $-\pi/2$ to $\pi/2$ radians.

### See also

acos atan atan2 cos sin tan

### Example

```
#include    <math.h>

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

    x = 0.5;

    res = asin(x);
}
```

# atan    Function

## Function

Computes the arc tangent of its argument.

## Syntax

#include    <math.h>

double    atan( double *x* );

*x*    The real number value for which the arc tangent is to be computed

## Description

The `atan` function computes the arc tangent of its argument *x*.

## Return value

The `atan` function returns the arc tangent of *x*, which is a value in the range $-\pi/2$ to $\pi/2$ radians.

## See also

acos  asin  atan2  cos  sin  tan

## Example

```
#include    <math.h>

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

    x = 0.5;

    res = atan(x);
}
```

# atan2 Function

## Function

Computes the arc tangent of *y*/*x*.

## Syntax

#include <math.h>

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

*x*, *y* Arbitrary real number values

## Description

The `atan2` function computes the arc tangent of *y*/*x*. This function returns correct values even when *x* is zero or close to zero. Returns zero when both *x* and *y* are zero.

## Return value

The `atan2` function returns the arc tangent of *y*/*x*, which is a value in the range $-\pi$ to $\pi$ radians.

## See also

acos asin atan acos sin tan

## Example

```
#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

This routine converts a character string to a floating point number of type `double`.

### Syntax

#include <stdlib.h>

double atof( char **s* );

*s* Character string to be converted

### Description

The `atof` routine converts the character string pointed to by the argument *s* to a double precision floating point number, and return that value. Note that `atof` is equivalent to the following function call.

strtod( s, ( char * * )NULL );

The string *s* must conform to the following syntax.

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

The symbols used have the following meanings.

| Symbol | Meaning |
|---|---|
| [*white space*] | Some number of tabs and spaces (may be omitted) |
| [*sign*] | Sign (may be omitted) |
| [*digit*] [.] [*digit*] | Character string expressing a decimal fraction (may be omitted) |
| [{e\|E} [*sign*] *digit*] | Character string expressing the exponent (may be omitted) |

The `atof` routine stops scanning when they encounter an unrecognized character. Also, they return HUGE_VAL and set `errno` to ERANGE if the value converted cannot be represented by the type `double`.

### Return value

The `atof` routine returns the value of the converted character string in an object of type `double`.

### See also

atoi atol strtod strtol strtoul

### Example

```
#include    <stdlib.h>

void    main( void )
{
    double   res;

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

# atoi

## Macro/Function

### Function

This routine converts a character string to an integer of type `int`.

### Syntax

#include <stdlib.h>

int atoi( char *s );

*s* Character string to be converted

### Description

The `atoi` routine converts the character string pointed to by the argument *s* to an integer of type `int`, and return that value. Note that `atoi` is equivalent to the following function call.

( int )strtol( s, ( char * * )NULL, 10 );

The string s must conform to the following syntax.

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

The symbols used have the following meanings.

| Symbol | Meaning |
|---|---|
| **[***white space***]** | Some number of tabs and spaces (may be omitted) |
| **[***sign***]** | Sign (may be omitted) |
| **[***digit***]** | Character string expressing an integer (may be omitted) |

The `atoi` routine stops scanning when they encounters an unrecognized character. Also, the return value from `atoi` when an overflow occurs is undefined.

### Return value

The `atoi` routine returns the value of the converted character string in an object of type `int`.

### See also

atof atol strtod strtol strtoul

### Example

```
#include    <stdlib.h>

void    main( void )
{
    int res;

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

# atol

## Macro/Function

### Function

This routine converts a character string to an integer of type `long`.

### Syntax

#include <stdlib.h>

long atol( char **s* );

*s* Character string to be converted

### Description

The `atol` routine converts the character string pointed to by the argument *s* to an integer of type `long`, and return that value. Note that `atol` is equivalent to the following function call.

( long )strtol( s, ( char * * )NULL, 10 );

The string *s* must conform to the following syntax.

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

The symbols used have the following meanings.

| Symbol | Meaning |
|---|---|
| **[***white space***]** | Some number of tabs and spaces (may be omitted) |
| **[***sign***]** | Sign (may be omitted) |
| **[***digit***]** | Character string expressing an integer (may be omitted) |

The `atol` routine stops scanning when they encounter an unrecognized character. If the converted value is too large to be represented by an integer of type `long`, the `atol` routines return either LONG_MAX or LONG_MIN and set `errno` to ERANGE.

### Return value

The `atol` routine returns the value of the converted character string in an object of type `long`.

## See also

atof  atoi  strtod  strtol  strtoul

## Example

```
#include    <stdlib.h>

void    main( void )
{
    long    res;

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

# bsearch **Function**

## Function

This function performs a binary search for a specified item in a sorted array.

## Syntax

#include <stdlib.h>

void *bsearch( void **key*, void **base*, size_t *nelem*, size_t *size*,
int ( **cmp* )( void *, void * ) );

*key* Search key

*base* Array to be searched

*nelem* Number of elements in the array

*size* Byte count indicating the size of each element

*cmp* Pointer to a comparison function

## Description

The `bsearch` function searches for an element that matches *key* in the array *base*, which has *nelem* elements. NULL is returned if no element is found that matches the specified item. Note that the array elements must be sorted in advance.

The function **cmp* is a user-specified comparison function that must take as its arguments two `void` pointers (void *). If these two arguments are *elem1* and *elem2*, the function must return the following integers based on the result of the comparison.

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

## Return value

The `bsearch` function returns a pointer to the element in the array that matches *key*. NULL is returned if there is no matching element.

### See also

qsort

**Note:** The comparison function must have the __noacc modifier. Without this modifier, compiling with CC665S's /REG option causes the function to take its first argument from the accumulator instead of the stack, where bsearch() places it.

For further details, see the sections "/REG Option" and "Functions Modified with __accpass and __noacc" in the CC665S User's Manual.

### Example

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

# calloc

**Function**

### Function

Allocates the required amount of memory

### Syntax

#include <stdlib.h>

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

*nelem* The number of elements

*size* The size of each element

### Description

`calloc` allocates *nelem* × *size* bytes of memory in the dynamic segment. The allocated memory is all initialized to zero.

### Return value

`calloc` returns a pointer to the newly allocated memory. It returns `NULL` if the requested memory could not be allocated or if either *nelem* or *size* was zero.

### See also

free malloc realloc

### Example

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

void    main( void )
{
    char    *s;

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

# ceil

**Function**

### Function

Computes the ceiling of (i.e., rounds up) a floating point number.

### Syntax

#include <math.h>

double ceil( double *x* );

*x* Floating point value

### Description

The `ceil` function finds the smallest integer not less than its argument.

### Return value

The `ceil` function returns the value found as an object of type `double` with an integral value.

### See also

floor fmod

### Example

```
#include    <math.h>

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

    num = 12.3;

    up = ceil(num);
}
```

# cos

**Macro/Function**

## Function

Computes the cosine of its argument.

## Syntax

#include <math.h>

double cos( double *x* );

*x* An angle in radian units

## Description

The `cos` routine computes the cosine of the input value *x*.

## Return value

The `cos` routine returns a value in the range –1 to 1.

## See also

acos asin atan atan2 sin tan

## Example

```
#include    <math.h>

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

    x = 0.5;

    res = cos(x);
}
```

# cosh **Function**

## Function

Computes the hyperbolic cosine of its argument.

## Syntax

#include <math.h>

double cosh( double *x* );

*x* An angle in radian units

## Description

The `cosh` function computes the hyperbolic cosine, i.e., $(e^{x} + e^{-x})/2$, of the input value *x*.

## Return value

The `cosh` function returns the hyperbolic cosine of the argument *x*.

It returns HUGE_VAL and sets the global variable `errno` to ERANGE if the result is too large to represent.

## See also

acos asin atan atan2 cos sin sinh tan tanh

## Example

```
#include    <math.h>

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

    x = 0.5;

    res = cosh(x);
}
```

# div

**Function**

## Function

Computes the quotient and remainder of two values of type `int`.

## Syntax

#include <stdlib.h>

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

*numer* Dividend

*denom* Divisor

## Description

The `div` function divides the argument *numer* by the argument *denom* and returns the result in an object of type `div_t`. The type `div_t` has two elements of type `int`, `quot` and `rem`, and the `div` function stores the quotient in `quot` and the remainder in `rem`.

## Return value

The `div` function returns the a structure that has `quot` (quotient) and `rem` (remainder) as its members.

## See also

ldiv

## Example

```
#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

Computes the exponential function ($e^x$) of its argument.

## Syntax

#include <math.h>

double exp( double *x* );

*x* Floating point value

## Description

The `exp` function computes the exponential function ($e^x$) of its argument *x*.

## Return value

The `exp` function returns the value $e^x$. Returns HUGE_VAL on overflow and 0.0 on underflow. Sets `errno` to ERANGE for both these cases.

## See also

frexp ldexp log log10 pow sqrt

## Example

```
#include    <math.h>

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

    x = 5.5;

    res = exp(x);
}
```

# fabs Function

## Function

Computes the absolute value of a floating point number.

## Syntax

#include <math.h>

double fabs( double *x* );

*x* Floating point value

## Description

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

## Return value

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

## See also

abs labs

## Example

```
#include    <math.h>

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

    num = 12.3;

    val = fabs(num);
}
```

# floor

**Function**

### Function

Truncates a value at the decimal point.

### Syntax

#include <math.h>

double floor( double *x* );

*x* Floating point value

### Description

The `floor` function returns the largest integer not greater than the argument *x*.

### Return value

The `floor` function returns the largest integer not greater than the argument *x* as a floating point number.

### See also

ceil fmod

### Example

```
#include    <math.h>

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

    num = 12.3;

    down = floor(num);
}
```

# fmod **Function**

### Function

Computes the floating point remainder.

### Syntax

#include <math.h>

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

*x*, *y* Floating point value

### Description

The `fmod` function computes the value f, which is the remainder of *x* divided by *y* such that *x* = a*y* + f, where a is an integer, f has the same sign as *x*, and | f | is less than | *y* |.

### Return value

The `fmod` function returns the remainder as a floating point value. It sets the global variable `errno` to EDOM if *y* is zero.

### See also

ceil fabs floor modf

### Example

```
#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.

## Syntax

#include <stdlib.h>

void free( void **ptr* );

*ptr* Pointer to the memory to be released

## Description

`free` releases memory allocated by `calloc`, `malloc`, or `realloc`. *ptr* must be a pointer returned by `calloc`, `malloc`, or `realloc`. The operation is undefined if a pointer to any other area is passed to `free`. `free` returns without taking any action if it is passed a `NULL` pointer.

## Return value

None

## See also

calloc  malloc  realloc

## Example

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

void    main( void )
{
    char    *s;

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

# frexp

**Function**

## Function

Breaks a floating point number into its fraction and exponent parts.

## Syntax

#include <math.h>

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

*x* Floating point value

*pexp* Pointer to the location that will hold the exponent

## Description

The `frexp` function breaks the argument *x* into a fractional part m (such that the absolute value of m is 0.5 or greater and less than 1.0) and an exponent part n, such that the relation $x = m \times 2^n$ holds. Note that it stores the exponent n, which is an integer value, at the location pointed to by *pexp*.

## Return value

The `frexp` function returns the value of the exponent m.

## See also

ldexp modf

## Example

```
#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 routines classify characters.

## Syntax

```
#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 between 0x00 and 0xff inclusive)

## Description

These routines determine the classification of the character *c*, and return the result of that determination. These routines assume the ASCII character set.

The result is undefined for values of *c* outside the range 0x00 to 0xff.

The table below lists these routines and the test each one performs.

| Routine | Test |
| --- | --- |
| isalnum | Tests if a character is a decimal digit ('0' to '9') or an alphabetic character ('a' to 'z' or 'A' to 'Z'). |
| isalpha | Tests if a character is an alphabetic character ('a' to 'z' or 'A' to 'Z'). |
| iscntrl | Tests if a character is a control character, i.e. is one of 0x00 to 0x1f or 0x7f. |
| isdigit | Tests if a character is a decimal digit ('0' to '9'). |
| isgraph | Tests if a character is a printable character other than space (' '), i.e., is in the range 0x21 to 0x7e. |
| islower | Tests if a character is a lower case letter ('a' to 'z'). |
| isprint | Tests if a character is a printable character including the space character (' '), i.e., is in the range 0x20 to 0x7e. |
| ispunct | Tests if a character is a punctuation character, i.e., is one of 0x21 to 0x2f, 0x3a to 0x40, 0x5b to 0x60, and 0x7b to 0x7e. |
| isspace | Tests if a character is a white space character, i.e., is one of 0x09 to 0x0d and space (' '). |
| isupper | Tests if a character is an upper case letter ('A' to 'Z'). |
| isxdigit | Tests if 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 fulfilled, and zero if it is not fulfilled.

The return valve is undefined for valves of *c* outside the 0x00 to 0xff.

## See also

toupper  tolower

### Example

```
#include    <ctype.h>

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

    /*
    The following loop test the classes of the letters 'a' to 'z'.
    */

    for ( c = 'a' ; c <= 'z' ; ++c )
    {
        retval1 = isalnum( c );     /* True, since 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 decimal
                                       digits. */
        retval5 = isxdigit( c );    /* True for 'a' to 'f', false
                                       for the others. */
    }
}
```

# itoa

**Function**

## Function

Converts an integer of type `int` to a character string in the specified base.

## Syntax

#include <stdlib.h>

char *itoa( int *number*, char **s*, int *base* );

*number* Number to be converted

*s* Buffer to store the converted character string

*base* The radix in which to express *number*

## Description

The `itoa` function converts *number* to a null terminated character string, and stores that result in *s*. The argument *base* specifies the radix in which *number* is to be expressed. The value of *base* must be in the range 2 to 36. If *base* is less than 2 or greater than 36, `itoa` sets *s* to the null string.

An area large enough to hold the converted string must be allocated for *s*. The maximum length of the string converted by `itoa`, including the terminating null character, is 17 bytes.

## Return value

The `itoa` function returns a pointer to the character string *s*.

## See also

ltoa ultoa

## Example

```
#include    <stdlib.h>

char    buf[17];

void    main( void )
{
    itoa( 12345, buf, 10 );
}
```

# labs

**Function**

## Function

Returns the absolute value of an integer of type `long`.

## Syntax

#include <stdlib.h>

long labs( long *n* );

*n* Integer

## Description

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

## Return value

The `labs` function returns an integer in the range 0 to 2147483647. However, if *n* is –2147483648 it returns –2147483648.

## See also

abs fabs

## Example

```
#include    <stdlib.h>

void    main( void )
{
    long    n, res;

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

# ldexp  Function

## Function

Computes a real number from a fraction and an exponent.

## Syntax

#include    <math.h>

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

*x*  Floating point value

*xexp*  Integer exponent

## Description

The `ldexp` function computes the value *x* times 2 raised to the power *xexp*.

## Return value

The `ldexp` function returns the computed value *x* times 2 raised to the power *xexp*. It sets the global variable `errno` to ERANGE if the result of the computation is too large to represent.

## See also

exp  frexp  modf

## Example

```
#include    <math.h>

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

    x = 4.5;

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

# ldiv

**Function**

## Function

Computes the quotient and remainder of two integers of type `long`.

## Syntax

#include <stdlib.h>

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

*numer* Dividend

*denom* Divisor

## Description

The `ldiv` function divides the argument *numer* by the argument *denom* and returns the result in an object of type `ldiv_t`. The type `ldiv_t` has two elements of type `long`, `quot` and `rem`, and the `ldiv` function stores the quotient in `quot` and the remainder in `rem`.

## Return value

The `ldiv` function returns a structure that has `quot` (quotient) and `rem` (remainder) as its members.

## See also

div

## Example

```
#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

Computes the natural logarithm of a number *x*.

### Syntax

#include <math.h>

double log( double *x* );

*x* The value that is the object of the logarithm calculation.

### Description

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

### Return value

The `log` function returns the calculated value, ln(*x*). It sets the global variable `errno` to EDOM if the argument *x* is negative. It returns –HUGE_VAL if the argument *x* is zero, and HUGE_VALE if the result is too large to represent. Sets `errno` to ERANGE for both these cases.

### See also

exp log10

### Example

```
#include    <math.h>

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

    x = 10;

    res = log(x);
}
```

# log10

**Macro/Function**

### Function

Computes the common logarithm of its argument.

### Syntax

#include <math.h>

double log10( double *x* );

*x* The value that is the object of the logarithm calculation.

### Description

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

### Return value

The `log10` function returns the calculated value. It sets the global variable `errno` to EDOM if the argument *x* is negative. It returns –HUGE_VAL if the argument *x* is zero, and HUGE_VALE if the result is too large to represent. Sets `errno` to ERANGE for both these cases.

### See also

exp log

### Example

```
#include    <math.h>

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

    x = 10;

    res = log10(x);
}
```

# longjmp

**Function**

### Function

Performs a global jump.

### Syntax

#include        <setjmp.h>

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

*environment*        Area that holds an execution environment

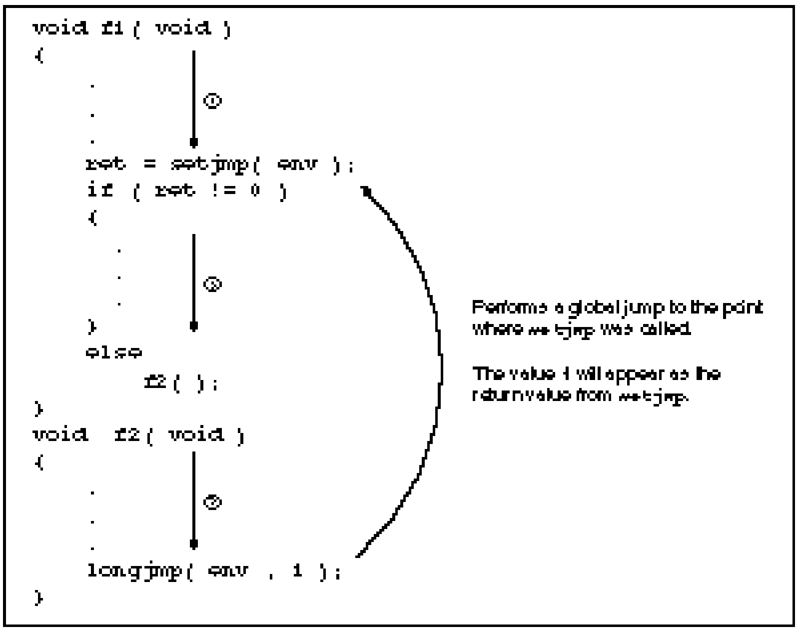
*value*        The value that will be returned by `setjmp`

### Description

The `longjmp` function performs a global jump to the point where `setjmp` was called.

Global jumps can be performed by using the `setjmp` and `longjmp` functions. The `longjmp` function restores an execution environment saved in the argument *environment* in advance by the `setjmp` function. As a result, the program appears to have returned from `setjmp` after `longjmp` is called. The argument *value* becomes the return value from `setjmp` at the point the execution environment is restored.

The figure below shows the operation of `setjmp` and `longjmp` using a simple example. The program execution proceeds in the order ①, ②, and then ③.



The value of *value* must be non-zero. The `setjmp` will return one if zero is specified for *value*.

The following points must be observed when using `longjmp`. The operation of programs that do not observe these points is undefined.

(1) An environment must be saved in advance by `setjmp` before calling `longjmp`.

(2) The `longjmp` function must not be called after the function that called `setjmp` returns.

### Return value

None

### See also

setjmp

### Example

```
#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 );
    .
    .
    .
}
```

# ltoa Function

### Function

Converts an integer of type `long` to a character string in the specified base.

### Syntax

#include <stdlib.h>

char *ltoa( long *number*, char **s*, int *base* );

*number* Number to be converted

*s* Buffer to store the converted character string

*base* The radix in which to express *number*

### Description

The `ltoa` function converts *number* to a null terminated character string, and stores that result in *s*. The argument *base* specifies the radix in which *number* is to be expressed. The value of *base* must be in the range 2 to 36. If *base* is less than 2 or greater than 36, then `ltoa` sets *s* to the null string.

An area large enough to hold the converted string must be allocated for *s*. The maximum length of the string converted by `ltoa`, including the terminating null character, is 33 bytes.

### Return value

The `ltoa` function returns a pointer to the character string *s*.

### See also

itoa ultoa

### Example

```
#include    <stdlib.h>

char    buf[33];

void    main( void )
{
    ltoa( 123456, buf, 10 );
}
```

# malloc

**Function**

## Function

Allocates memory.

## Syntax

#include <stdlib.h>

void *malloc( size_t *size* );

*size* The size of the memory to allocate.

## Description

`malloc` allocates *size* bytes of memory in the dynamic segment. Due to memory boundary management considerations, each time `malloc` is called it may actually consume *size* + n bytes of memory if *size* is even and *size* + (n + 1) bytes of memory if *size* is odd. (The value of n is 2 for the small, effective medium, and medium memory models, and 4 for the compact, effective large and large memory models.) The contents of allocated memory are not initialized.

The dynamic segment is the largest area remaining after RL66K has allocated all logical segments in the address space.

## Return value

`malloc` returns a pointer to the allocated memory. It returns `NULL` if the requested memory could not be allocated or if *size* was zero.

## See also

calloc free realloc

**Note:** When RL66K allocates the dynamic segment, it allocates an area that fills the data memory defined in the DCL file, regardless of whether external RAM is present in the system or not.

Therefore, when the system has only internal RAM or an external RAM with a limited capacity, `malloc` will not return an error (`NULL`) even if all the existent area is used due to multiple calls to `malloc`. This is because the `malloc` function uses the size acquired from the dynamic segment for memory management. Accordingly, normal operation cannot be guaranteed in this case.

To prevent this problem, use the /DM option to specify the valid data memory area at link time. For example, if the actual data memory capacity is only 7FFH bytes, specify /DM(7FFH).

### Example

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

void    main( void )
{
    char    *s;

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

# memchr

**Function**

## Function

This function searches for a specified data byte in a specified memory area.

## Syntax

| | |
|---|---|
| #include | <string.h> |

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

| | |
|---|---|
| *region* | Pointer to a memory area |
| *c* | Datum to be searched for |
| *count* | Number of bytes over which to search |

## Description

The `memchr` function searches for an occurrence of *c* in the first *count* bytes of region. Although *c* is of type `int`, it must have a value in the range 0x00 to 0xff.

## Return value

The `memchr` function returns a pointer to the first occurrence of *c*, if *c* occurs within the first count bytes of *region*. It returns NULL if *c* is not found. It also returns NULL if *count* is 0.

## See also

memcmp memcpy memset strchr

### Example

```
#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;

    .
    .
    .
    /* This call returns the addresses of data[8]. */
    ptr = memchr( data , 0x80 , 16 );
    .
    .
    .
    /* This returns NULL since there is no byte with the
       value 0xff. */
    ptr = memchr( data , 0xff , 16 );
    .
    .
    .
    /* This returns NULL since there is no byte with the
       value 0x80 in the first 4 bytes. */
    ptr = memchr( data , 0x80 , 4 );
    .
    .
    .
}
```

# memcmp

**Function**

## Function

This function compares two memory areas.

## Syntax

#include <string.h>

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

*region1* Memory area 1

*region2* Memory area 2

*count* Number of bytes to compare

## Description

The `memcmp` function compares the first *count* bytes of *region1* and *region2* on a byte by byte basis. Unlike the `strcmp` function, this function continues to compare beyond occurrences of the null character ('¥0').

## Return value

The table below lists the return values according to 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

### Example

```
#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;

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

# memcpy

**Function**

## Function

This function copies data in one memory area to another

## Syntax

#include <string.h>

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

*dest* Copy destination

*src* Copy source

*count* Number of bytes to copy

## Description

The `memcpy` function copies *count* bytes from *src* into *dest*. Unlike `strcpy` and `strncpy`, these function will copy bytes containing the null character ('¥0').

The behavior is undefined if the source and destination areas overlap. Use the `memmove` function to copy overlapping areas.

## Return value

The `memcopy` function returns *dest*.

## See also

memchr memcmp memmove memset strcpy strncpy

### Example

```
#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 in one memory area to another.

## Syntax

| | |
|---|---|
| #include | <string.h> |
| void | *memmove( void **dest* , void **src* , size_t *count* ); |
| *dest* | Copy destination |
| *src* | Copy source |
| *count* | Number of bytes to copy |

## Description

The `memmove` function copies *count* bytes from *src* into *dest*. Unlike `strcpy` and `strncpy`, these function will copy bytes containing the null character ('¥0').

## Return value

The `memmove` function returns *dest*.

## See also

memcpy  strcpy  strncpy

**Example**

```
#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 )
{
    .
    .
    .
    /* Copies 32 bytes of data starting at data + 16 to
       the memory area starting at data.
       Performs the copy correctly, even though the areas
       overlap. */
    memmove( data , data+16 , 32 );
    .
    .
    .
}
```

# memset

**Function**

### Function

Initializes a specified area of memory with a given data byte.

### Syntax

#include        <string.h>

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

*region*     Memory area

*c*             Data byte to be stored in memory.

*count*      Number of bytes

### Description

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

### Return value

The `memset` function returns *region*.

### See also

memchr  memcpy  memcmp  memmove

### Example

```
#include    <string.h>

char    ram_data[64];

void    main( void )
{

    char    *retptr;

    /* Initializes the first 32 bytes of the buffer ram_data
       with 0xff. */
    retptr = memset( ram_data , 0xff , 32 );

}
```

# modf

**Function**

### Function

Breaks a floating point number into its integer and fraction parts.

### Syntax

#include <math.h>

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

*x* Floating point value

*pint* Pointer to location to hold the integer part.

### Description

The `modf` function breaks its floating point argument *x* into integer and fractional parts, stores the integer part of *x* at the location pointed to by *pint*, and returns the fraction part as the value of the function.

### Return value

The `modf` function returns the fraction part of its argument *x* with the sign.

### See also

fmod frexp ldexp

### Example

```
#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.

## Syntax

#include <stddef.h>

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

*structname* structure name

*fieldname* member of structure

## Description

The `offsetof` macro determines the offset of the field *fieldname* in the structure *struct - name* as a number of bytes.

## Return value

The `offsetof` macro returns the offset of the field *fieldname* in the structure *structname* as a number of bytes.

## Example

```
#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

Computes *x* raised to the *y* power.

## Syntax

#include <math.h>

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

*x* Numeric value

*y* The exponent to which *x* is to be raised.

## Description

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

## Return value

The `pow` function returns the computed value of *x* raised to the *y* power. There are cases where, depending on the values of the arguments, either overflow occurs or the calculation cannot be performed. On overflow, the `pow` function 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. `pow` returns 1 if both *x* and *y* are zero.

## See also

exp  sqrt

## Example

```
#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.

## Syntax

#include <stdlib.h>

void qsort( void **base*, size_t *n*, size_t *size*, int ( **cmp* )( void *, void * ) );

| | |
|---|---|
| *base* | The start of the array to be sorted |
| *n* | The number of elements in the array |
| *size* | The size of each element |
| *cmp* | A pointer to a comparison function |

## Description

The `qsort` function sorts an array using the Quicksort algorithm. The `qsort` function sorts the elements in the array by calling the user defined comparison function pointed to by *cmp*.

The function **cmp* is a user-specified comparison function that must take as its arguments two `void` pointers (void *). If these two arguments are *elem1* and *elem2*, the function must return the following integers based on the result of the comparison.

| Condition | Return Value |
|---|---|
| **elem1* < **elem2* | Negative |
| **elem1* = = **elem2* | 0 |
| **elem1* > **elem2* | Positive |

## Return value

None

## See also

bsearch

**Note:** The comparison function must have the __noacc modifier. Without this modifier, compiling with CC665S's /REG option causes the function to take its first argument from the accumulator instead of the stack, where qsort() places it.

For further details, see the sections "/REG Option" and "Functions Modified with __accpass and __noacc" in the CC665S User's Manual.

### Example

```
#include    <stdlib.h>

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

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

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

# rand

**Function**

## Function

Generates pseudo-random numbers.

## Syntax

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

## Description

The `rand` function generates a pseudo-random number in the range 0 to RAND_MAX and returns that value.

## Return value

The `rand` function returns a pseudo-random number.

## See also

srand

## Example

```
#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.

### Syntax

#include <stdlib.h>

void *realloc( void **ptr*, size_t *size* );

*ptr* Pointer to the memory to be reallocated

*size* Allocation size

### Description

`realloc` reallocates memory that was allocated by `calloc` or `malloc`.

`realloc` allocates memory of the requested size and returns a pointer to that memory. If new memory was actually allocated, the content of the original memory is copied to the allocated memory. `realloc` functions identically to `malloc` if *ptr* is `NULL`. If *size* is 0 and *ptr* is not `NULL`, `realloc` frees the memory pointed to by *ptr*.

### Return value

`realloc` returns a pointer to the reallocated memory. `realloc` returns `NULL` if it could not reallocate memory.

### See also

calloc  free  malloc

### Example

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

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

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

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

    /*  Reallocates memory.
        The contents of s1 at this point is copied into s2.
    */
    s2 = ( char * )realloc( s1, strlen( s1 ) + strlen(string2 ) + 1);

    /*  Concatenate s2 and string2. */
    strcat( s2, string2 )

    /*  The contents of s2 is now "library reference."*/
}
```

# setjmp

**Macro**

## Function

Saves the current program execution environment for the global jump function.

## Syntax

#include <setjmp.h>

int setjmp( jmp_buf *environment* );

*environment* Area to hold the execution environment

## Description

The `setjmp` macro saves the current program execution environment in *environment*.

Global jumps can be performed by using the `setjmp` and `longjmp` functions. The `longjmp` function restores an execution environment saved in the argument *environment* in advance by the `setjmp` function. As a result, the program appears to have returned from `setjmp` after `longjmp` is called.

Although the `setjmp` macro returns zero when it is called to save the environment, it returns a value other than zero (the argument to `longjmp`) when the environment is restored by a call to `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

The `setjmp` macro always returns zero when it is called to save the environment. When `setjmp` returns as a result of a call to `longjmp`, it returns the non-zero value that was the value of the second argument (*value*) to `longjmp`.

## See also

longjmp

## Example

See the example under `longjmp`.

# sin

## Macro/Function

### Function

Computes the sine of its argument.

### Syntax

#include <math.h>

double sin( double *x* );

*x* An angle in radian units

### Description

The `sin` routine computes the sine of its argument *x*.

### Return value

The `sin` routine returns the sine of its argument *x*.

### See also

acos asin atan atan2 cos tan

### Example

```
#include    <math.h>

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

    x = 0.5;

    res = sin(x);
}
```

# sinh

Function

### Function

Computes the hyperbolic sine of its argument.

### Syntax

#include <math.h>

double sinh( double *x* );

*x* An angle in radian units

### Description

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

### Return value

The `sinh` function returns the hyperbolic sine of its argument *x*.

If the result is too large to represent, `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

### Example

```
#include    <math.h>

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

    x = 0.5;

    res = sinh(x);
}
```

# sprintf

**Function**

## Function

This function writes text to a character string according to a format specification.

## Syntax

#include <stdio.h>

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

| | |
|---|---|
| *buffer* | Buffer to hold the output character string |
| *format* | Format string |
| *argument* | Argument corresponding to a conversion type specifier |

## Description

The `sprintf` function creates a character string according to the format string pointed to by *format*, and write that string to *buffer*.

The *format* argument consists of normal characters and an arbitrary number of conversion specifiers. The number and types of the arguments following *format* must match the number of conversion specifiers and the types specified by each conversion specifier in *format*. The behavior is undefined if the number of arguments is smaller than the number of conversion specifiers or if the type specified by a conversion specifier does not match the type of the corresponding argument. Extra arguments are ignored if the number of arguments exceeds the number of conversion specifiers.

Conversions specifiers have the following syntax.

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

A sequence of flag characters is specified in the *flags* field. The conversion field width is specified in the *width* field. The precision is specified in the .*prec* field. The terms `h`, `l`, and `L` are type length specifiers. The conversion type specifier is specified in the *type* field.

The flags, field width, precision, and type length are optional. The table that follows provides an overview of these options.

| Option | Meaning |
|---|---|
| *flags* | The flags specify aspects such as left justification or right justification, and the sign, decimal point, or base (octal or hexadecimal) for numeric values. |
| *width* | Specifies the minimum width of the characters output. |
| *.prec* | Specifies the maximum width of the characters output. Specifies the minimum number of digits output for integers. |
| {h\|l\|L} | Determines the size of the corresponding argument.<br>h short int<br>l long<br>L long double |

## Conversion Type Specifier (*type*)

This table lists the conversion type specifiers.

| Conversion Type Specifier | Type | Output Format |
|---|---|---|
| d, i | int | Converts to a signed decimal character string. |
| o | unsigned int | Converts to an unsigned octal character string. |
| u | unsigned int | Converts to an unsigned decimal character string. |
| x | unsigned int | Converts to an unsigned hexadecimal character string. The values 10, 11, 12, 13, 14, and 15 are converted to a, b, c, d, e, and f respectively. |
| X | unsigned int | Converts to an unsigned hexadecimal character string. The values 10, 11, 12, 13, 14, and 15 are converted to A, B, C, D, E, and F respectively. |
| f | double | Converts to a signed format of the form [–] *d.dddddd.* |
| e | double | Converts to a signed format of the form [–] *d.dddddd*e+/–*dd.* |
| E | double | The same as 'e', except that the exponent is indicated by 'E'. |
| g | double | Converts to the format specified by either e or f. Normally expresses values in the f format. However, expresses values in the e format if the exponent is less than –4 or if it is larger than the conversion field precision. |
| G | double | The same as g, except that it converts to the format specified by either E or f. |
| c | int | Converts to a single character. |
| s | char * | Outputs characters from the character string pointed to by the corresponding argument up to the conversion field precision or until the end of that string. The pointer must point to a character string in RAM. |
| S | const char * | The same as s, except that the pointer points to a character string in ROM. |
| p | void * | Outputs the input argument as a pointer. |
| n | int * | Stores the number of characters output thus far into the object pointed to by the corresponding argument. |
| % | — | Outputs a '%' character. |

The output formats described in the table assume that flag characters, a width specifier, a field precision width, and a type length were not specified. The remainder of this section describes the influence of combinations of options and conversion type specifiers on the output format.

## Flag Characters (*flags*)

The following flag characters are supported.

| Flag Character | Description |
|---|---|
| – | Justifies the output character string to the left edge of the field. If not specified, the string is right justified. |
| + | Always attaches a sign at the start of a number. If not specified, a sign character is only output for negative values. |
| Space (0x20) | Inserts a space at the front of positive numbers. A minus sign is output at the start of a negative number. |
| # | Applies to conversion type specifiers for numeric data types. Allocates an appropriate format corresponding to the conversion type specifier. See the following table. |
| 0 | If a 0 precedes one of the d, e, E, f, g, G, i, u, x, or X conversion type specifiers, the field is filled with zeros instead of spaces. The '0' flag is ignored if a precision is specified for d, i, o, u, x, or X conversions or if the '–' flag is specified. |

Conversion type specifiers are modified by the presence of the '#' flag as shown in the following table.

| Conversion Type Specifier | Influence of the '#' Flag |
|---|---|
| c, d, i, u, s, S | No effect |
| o | A zero is inserted at the beginning of the number for non-zero values. |
| x, X | A '0x' prefix is inserted. |
| e, E, f | A decimal point is always inserted. |
| g, G | A decimal point is always inserted, and a zero is inserted following the decimal point. |

## Field Width (*width*)

The field width specifies the minimum width of the field into which the converted character string is written.

When a field width is specified, if the converted string is shorter than the field width, then it is padded with spaces up to the size of the width. The padding spaces are inserted at the right if the '–' flag is specified, and at the left otherwise. Also, if the first character in the field width specification is '0', then the field is padded with zeros instead of spaces. If the converted string is longer than the field width, then the field width is increased to the length of the converted string.

It is possible to specify the field width indirectly with an asterisk ( * ). In this case, the field width will be taken from an argument of type `int`. For example, if the following notation is used,

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

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

then the argument will be used as the field `width` and the following character string will be output to `buf`.

```
|    1234|
```

## Precision (*.prec*)

The precision specifier starts with a period. The precision syntax is the same as that for the field width. The precision is taken to be zero if only a period with no following number is specified.

The number of characters output when the precision is specified differs for each conversion type specifier. The table below lists the operation when a precision of *n* is specified.

| Conversion Type Specifier | Output |
|---|---|
| d, i, o, u, x, X | At least *n* digits are output. |
| e, E, f | Exactly *n* digits are output following the decimal point. |
| g, G | No more than *n* significant digits are output. |
| s, S | No more than *n* characters are output. |

### Type Length Specifier

The type length specifier changes the type of the corresponding argument.

| Type Length Specifier | Size |
|---|---|
| h | For the d, i, o, u, x, and X conversion type specifiers, indicates that the corresponding argument is of type `short int` or `unsigned short int`. |
| l | For the d, i, o, u, x, and X conversion type specifiers, indicates that the corresponding argument is of type `long int` or `unsigned long int`.<br>For the e, E, f, g, and G conversion type specifiers, indicates that the corresponding argument is of type `double`. |
| L | For the e, E, f, g, and G conversion type specifiers, indicates that the corresponding argument is of type `long double`. |

### Return Value

The `sprintf` function returns the number of bytes output to *buffer*. If an error occurs, `sprintf` returns EOF.

### See also

sscanf

### Example

```
#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, "RAM string" );
    res2 = sprintf( buf2, "|%-15s|%15S|", string, "ROM string" );
}
```

# sqrt

**Function**

### Function

Computes the square root of its argument.

### Syntax

#include <math.h>

double sqrt( double *x* );

*x* A non-negative floating point value

### Description

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

### Return value

The `sqrt` function returns the computed value of the square root of its argument *x*. It sets the global variable `errno` to EDOM if *x* is negative, and to ERANGE if the result is too large to represent.

### See also

exp log pow

### Example

```
#include    <math.h>

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

    x = 9.0;

    val = sqrt(x);
}
```

# srand

**Macro/Function**

### Function

Initializes the pseudo-random number sequence.

### Syntax

#include        <stdlib.h>

void        srand( unsigned int *seed* );

*seed*        Initialization value

### Description

The `srand` function initializes the pseudo-random number sequence. The pseudo-random number sequence generated by `rand` can be changed by using a different value for *seed*.

### Return value

None

### See also

rand

### Example

```
#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

This function reads in a character string and convert it to appropriate data types according to a format string.

## Syntax

| | |
|---|---|
| #include | <stdio.h> |
| int | sscanf( char **string*, char **format* **[**, *address*, ...**]** ); |
| *string* | Character string to be read in (input string) |
| *format* | Format string |
| *address* | Arguments corresponding to the conversion specifiers |

## Description

The `sscanf` function reads characters from the string pointed to by *string*, convert them to appropriate types according to the format string pointed to by *format*, and store the results in the locations pointed to by the corresponding *address* arguments.

The format string consists of white space, conversion specifiers, and characters other than the percent character (%). When the `sscanf` function encounters a white space characteristic in the format string, it jumps over all space characters until it encounters a character other than space. A conversion specifier starts with a percent character (%) and specifies how a section of the input string is to be interpreted. When the `sscanf` function encounters a conversion specifier, it acquires a corresponding token from the input string. For all other characters, sscanf reads over matching characters in the input string.

The number of conversion specifiers and the number of arguments following *format* must be identical. The behavior is undefined if the number of arguments is smaller than the number of conversion specifiers. Extra arguments are ignored if the number of arguments exceeds the number of conversion specifiers. Also, the type required by each conversion specifier must match the type of its corresponding argument. The behavior is undefined if they do not match.

Conversion specifiers have the following syntax.

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

An asterisk (*) indicates that the next field (token) is to be jumped over. Nothing is written into the corresponding argument. The *width* item specifies the maximum number of characters (the input width) in the input field. An h, l, or L is a type length specifier, and modifies the type of the argument. The *type* is the conversion type specifier.

The asterisk, input width, and type length items are optional.

## Conversion Type Specifier

The table below lists the conversion type specifiers. This table lists the argument type and the interpretation of the string read for each conversion type specifier.

| Conversion Type Specifier | Argument Type | Input String Interpretation |
|---|---|---|
| d, i | int * | Converts a decimal character string to an integer. The format of the string must be the same as a string interpreted by the `strtol` function when a base of 10 is specified. |
| o | unsigned int * | Converts an octal character string to an integer. The format of the string must be the same as a string interpreted by the `strtol` function when a base of 8 is specified. |
| u | unsigned int * | Converts an unsigned decimal character string to an unsigned integer. The format of the string must be the same as a string interpreted by the `strtoul` function when a base of 10 is specified. |
| x, X | unsigned int * | Converts a hexadecimal character string to an unsigned integer. The format of the string must be the same as a string interpreted by the `strtol` function when a base of 16 is specified. |
| f | float * | Converts a character string to floating point. The format of the string must be the same as a string interpreted by the `strtod` function when converting a decimal expression to floating point. |
| e, E | float * | Converts a character string to floating point. The format of the string must be the same as a string interpreted by the `strtod` function when converting an exponential expression to floating point. |
| g, G | float * | Converts a character string to floating point. The format of the string must be the same as a string interpreted by the `strtod` function when converting either a decimal expression or an exponential expression to floating point. |
| c | char * | Copies the number of characters specified by the field width to the array specified by the argument. Note that white space characters are included. A terminating null character ('¥0') is not written by this operation. If the field width is not specified, a single character is read. |

| Conversion Type Specifier | Argument Type | Input String Interpretation |
|---|---|---|
| s | char * | Copies a character string that includes no space characters to the character string specified by the argument. A terminating null character ('¥0') is written at the end of the string. |
| p | void * | Reads a character string as a pointer to type `void`. |
| n | int * | Stores the number of characters read so far in the area pointed to by the argument. |
| % | — | Reads a percent (%) character. Does not set the corresponding argument. |
| **[…]** | char * | Copies characters that match any of the characters in the character set enclosed in square brackets to the string pointed to by the argument. The space character can also be included in the character set. The syntax "[ ],...]" specifies that the character "]" is included in the character set that is the object of the scan. |
| **[^…]** | char * | Copies characters that do not match any of the characters in the character set enclosed in square brackets to the string pointed to by the argument. The space character can also be included in the character set. The syntax "[^],...]" specifies that the character "]" is included in the character set that is excluded from the object of the scan. |

The argument types in the preceding table assume that a type length was not specified. The changes that occur in the types due to type length specifications are described next.

## Type Length Specifier

The table below shows how the type length changes the type of the corresponding argument.

| Type Length Specifier | Type Interpretation |
|---|---|
| h | For conversion type specifiers d, i, o, u, x, and X, the corresponding argument is interpreted as a pointer to `short int` or `unsigned short int`. The h type length specifier is ignored for other conversion type specifiers. |
| l | For conversion type specifiers d, i, o, u, x, and X, the corresponding argument is interpreted as a pointer to `long int` or `unsigned long int`.<br>For conversion type specifiers e, E, f, g, and G, the corresponding argument is interpreted as a pointer to `double`. The l type length specifier is ignored for other conversion type specifiers. |
| L | For conversion type specifiers e, E, f, g, and G, the corresponding argument is interpreted as a pointer to `long double`. The L type length specifier is ignored for other conversion type specifiers. |

### Return value

The `sscanf` function returns the number of correctly read input data items. It returns EOF if an error occurred.

### See also

sprintf

### Example

```
#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

This function concatenates character strings.

## Syntax

#include <string.h>

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

*string1* The destination character string

*string2* The character string to be concatenated

## Description

The `strcat` function concatenates *string2* starting at the null character ('¥0') that terminates *string1*. It adds a terminating null character ('¥0') at the end of the resultant string.

## Return value

The `strcat` function returns *string1*.

## See also

strncat strcpy strncpy

### Example

```
#include    <string.h>

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

void    main( void )
{
    char  *retptr;

    .
    .
    .
    /* Creates the character string "library reference". */
    retptr = strcat( string1 , string2 );

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

# strchr

**Function**

## Function

This function searches for the first occurrence of a character in a string.

## Syntax

#include <string.h>

char *strchr( char **string* , int *c* );

*string* Character string

*c* Character to be found

## Description

The `strchr` function searches for *c* in string. The null character ('¥0') can be specified for *c*. Although the argument *c* is of type `int`, it must have a value in the range 0x00 to 0xff.

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

## Return value

The `strchr` function returns a pointer to the location where the character first appears. It returns NULL if the character is not found.

## See also

memchr strcspn strrchr strspn

### Example

```
#include    <string.h>

char  string[] = "0123456789012345678901234567890123456789";

void    main( void )
{
    char *ptr;

    /* Since the first occurrence of '9' is at entry 9, */
    /*      this function returns pointers to string[9]  */
    ptr = strchr( string , '9' );
    .
    .
    .
    /* When '¥0' is specified, this function returns
       pointers to the end of the string. */
    ptr = strchr( string , '¥0' );
    .
    .
    .
    /* This call returns NULL since the letter 'A' does not
       occur in the strings. */
    ptr = strchr( string , 'A' );
}
```

# strcmp Function

## Function

This function compares two character strings.

## Syntax

#include <string.h>

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

*string1* String to be compared

*string2* String to be compared

## Description

The `strcmp` function compares the alphabetical order of *string1* and *string2*.

## Return value

The table below lists the return values and their meanings.

| Return Value | Meaning |
|---|---|
| 0 | *string1* and *string2* are identical. |
| Positive | *string1* is larger than (later in alphabetical order than) *string2*. |
| Negative | *string1* is smaller than (earlier in alphabetical order than) *string2*. |

## See also

memcmp strncmp

### Example

```
#include    <string.h>

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

void    main( void )
{
    int retval;

    /* Returns a positive value since the first string is
       larger. */
    retval = strcmp( string1 , string2 );
    .
    .
    .
    /* Returns a negative value since the second string is
       larger. */
    retval = strcmp( string2 , string1 );
    .
    .
    .
    /* Returns zero since the strings are identical. */
    retval = strcmp( string1 , string1 );
}
```

# strcpy

**Function**

## Function

This function copies character strings.

## Syntax

#include <string.h>

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

*string1* Copy destination

*string2* Source string to be copied

## Description

The `strcpy` function copies *string2*, including the terminating null character ('¥0') into *string1*.

## Return value

The `strcpy` function returns *string1*.

## See also

memcpy strcat strncat strncpy

## Example

```
#include    <string.h>

char    string[128];

void    main( void )
{
    char    *retptr;

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

}
```

# strcspn

**Function**

## Function

This function determines the length of the first section of a string that does not contain any characters from a given character set.

## Syntax

#include <string.h>

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

*string1* Character string

*string2* Character set specified as a character string

## Description

The `strcspn` function searches in *string1* for the first occurrence of a character from *string2,* and returns the offset of that point from the start of *string1*. In other words, it determines the length of the starting section of *string1* that consists of characters not contained in *string2*. The terminating null character ('¥0') in *string1* is not included in the search range.

This function is very similar to `strpbrk`. However, it differs in that `strpbrk` returns a pointer to the first character that appears. Note that a function with the opposite functionality, the `strspn` function, is also provided.

## Return value

The `strcspn` function returns the length of the substring from the start of *string1* to the point where the first character in *string2* appears.

This function returns the length of *string1* when none of the characters in *string2* appear in *string1* or when *string2* is the null string ("").

## See also

strchr strrchr strpbrk strspn

### Example

```
#include    <string.h>

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

void    main( void )
{
    size_t  retval;

    .
    .
    .
    /*
        This call returns 7 since there are 7 characters in
        the string "ABCDEFG1234567" that precede the
        appearance of one of the characters in "1234567".
    */
    retval = strcspn( string1 , string2 );
    .
    .
    .
    /*
        This call returns the length of the string "ABCDE
        FG1234567", since none of the characters "XYZ"
        appears in the string.
    */
    retval = strcspn( string1 , "XYZ" );
}
```

# strlen Function

### Function

This function computes the length of a character string.

### Syntax

#include <string.h>

size_t strlen( char **string* );

*string* Character string

### Description

The `strlen` function determines the length of *string*, that is the number of characters (bytes) from the start of the *string* through the character directly preceding the terminating null character ('¥0').

### Return value

The `strlen` function returns the length of *string*.

### See also

None

**Example**

```
#include    <string.h>

char  string[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

void    main( void )
{
    size_t  length;

    /*
        This call returns 26, which is the length of the
        string.
    */
    length = strlen( string );
}
```

# strncat

**Function**

## Function

This function appends the first section of one character string onto the end of another.

## Syntax

#include <string.h>

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

*string1* Destination character string

*string2* Character string to be appended

*count* Number of characters to be appended

## Description

The `strncat` function appends the first *count* bytes of *string2* to *string1* starting at *string1's* terminating null character ('¥0'). It adds a terminating null character ('¥0') to the result string.

All of *string2* is appended to *string1* if *count* is greater than the length of *string2*. This operation is the same as that performed by the `strcat` function. The contents of *string1* will not be changed if *count* is zero or if *string2* is the null string.

## Return value

The `strncat` function returns *string1*.

## See also

strcat strcmp strcpy strncpy

### Example

```
#include    <string.h>

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

void    main( void )
{
    char  *retptr;

    /*
        Concatenates the first three characters of
        "reference".
        The contents of the string then becomes "library
        ref".
    */
    retptr = strncat( string1 , string2 , 3);

    /*
        A count larger than the length of the string "manual"
        is specified.
        The contents of the string then becomes "library ref
        manual".
    */
    retptr = strncat( retptr , string3 , 20);

    /*
        A character count of 0 is specified. The contents of
        the string is not changed.
    */
    retptr = strncat( retptr , string3 , 0);

}
```

# strncmp

**Function**

## Function

This function compares the specified number of characters in two character strings.

## Syntax

#include <string.h>

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

*string1* Character string to be compared

*string2* Character string to be compared

*count* Number of characters to be compared

## Description

The `strncmp` function determines the alphabetical order of first *count* bytes of *string1* and *string2*.

When *count* is smaller than the length of the strings being compared, then the first *count* bytes from the start of the strings form the range of the comparison. When *count* is larger than the length of the strings, then the strings up to the terminating null character ('¥0') form the range of the comparison. The result of `strncmp` when *count* is larger than the length of either *string1* or *string2* is the same as the result of the *strcmp* function.

## Return value

The table below lists the return values and their meanings.

| Return | ValueMeaning |
|---|---|
| 0 | *string1* and *string2* are identical. |
| Positive | *string1* is larger than (later in alphabetical order than) *string2*. |
| Negative | *string1* is smaller than (earlier in alphabetical order than) *string2*. |

## See also

memcmp strcat strcmp strcpy strncat strncpy

### Example

```
#include    <string.h>

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

void    main( void )
{
    int retval;

    /* A comparison up to the sixth byte. The result is zero. */
    retval = strncmp( string1 , string2 , 6 );

    /* A comparison up to the seventh byte. Since the first string
       is larger, the result is a positive value. */
    retval = strncmp( string1 , string2 , 7 );

}
```

# strncpy

**Function**

## Function

This function copies the specified number of bytes.

## Syntax

#include <string.h>

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

*string1* Copy destination

*string2* Source character string

*count* Number of characters to be copied

## Description

The `strncpy` function copies the first *count* bytes of *string2* into *string1*.

If *count* is equal to or less than the length of *string2*, no terminating null character ('¥0') is added to the copied string. If *count* is longer than *string2*, then all of *string2* is copied into *string1*, and furthermore, *string1* is padded with null characters through character number *count*.

## Return value

The `strncpy` function returns *string1*.

## See also

memcpy strcat strncat strcpy

### Example

```
#include    <string.h>

char  string1[] = "string";

char    string2[128];

void    main( void )
{
    char    *retptr;

    .
    .
    .
    /*
        Examples with a string of length 6 and a count of 3.
        Only the first 3 characters are copied. No null
        characters are written to string1.
    */
    retptr = strncpy( string2 , string1 , 3);
    .
    .
    .
    /*
        Examples with a string of length 6 and a count of 10.
        After the string "string" is copied, the remaining 4
        bytes are set to null.
        The result is "string¥0¥0¥0¥0".
    */
    retptr = strncpy( string2 , string1 , 10);

}
```

# strpbrk  Function

## Function

This function locates the first occurrence of any character in a specified character set in a character string.

## Syntax

#include <string.h>

char *strpbrk( char **string1* , char **string2* );

*string1* Character string

*string2* Character string that specifies the character set

## Description

The `strpbrk` function locates the first occurrence of any character in *string2* in *string1*, and return a pointer to that character. The terminating null character ('¥0') in *string1* is not included in the search range.

This function is very similar to the `strcspn` function. However, `strcspn` differs in that it returns the offset of the first appearing character from the start of the string.

## Return value

The `strpbrk` function returns a pointer to position in *string1* where a character from *string2* first appears.

This function returns NULL if none of the characters in *string2* appears in *string1*, or if either *string1* or *string2* is the null string ("").

## See also

strchr strcspn strrchr strspn

**Example**

```
#include    <string.h>

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

void    main( void )
{
    char    *ptr;

    /*
        This call returns a pointer to the seventh byte
        since there are 7 characters in the string "ABCDE
        FG1234567" that precede the appearance of one of the
        characters in "1234567".
    */
    ptr = strpbrk( string1 , string2 );

    /*
        This call returns NULL, since none of the characters
        "XYZ" appears in the string.
    */
    ptr = strpbrk( string1 , "XYZ" );

    /*
        This call returns NULL, since the null string was
        passed in the calls.
    */
    ptr = strpbrk( string1 , "" );
}
```

# strrchr

**Function**

## Function

This function determines the last position in a character string that a certain character appears.

## Syntax

#include <string.h>

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

*string* Character string

*c* Character to search for

## Description

The `strrchr` function determines the last position in *string* that *c* appears. The null character ('¥0') can also be specified for *c*. Although *c* is of type `int`, it must have a value in the range 0x00 to 0xff.

To find the position of the first occurrence of *c*, use the `strchr` function.

## Return value

The `strrchr` function returns a pointer to the position of the last occurrence of the character. It returns NULL if the character was not found.

## See also

memchr strcspn strchr strspn

**Example**

```
#include    <string.h>

char  string[] = "012345678901234567890123456789";

void    main( void )
{
    char *ptr;

    /* Since the last occurrence of '0' is at the twentieth
       position this call returns pointers to string[20]. */
    ptr = strrchr( string , '0' );
    .
    .
    .
    /* When '¥0' is specified, this function returns a
       pointer to the end of the string. */
    ptr = strrchr( string , '¥0' );
    .
    .
    .
    /* This call returns NULL since the character 'A' does
       not appear in the string. */
    ptr = strrchr( string , 'A' );
}
```

# strspn Function

## Function

This function determines the length of the section at the head of a string that consists of characters from a particular set of characters.

## Syntax

#include <string.h>

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

*string1* Character string

*string2* Character string that specifies the character set

## Description

The `strspn` function searches in *string1* for the location of the first character that does not appear in *string2*, and return that point as an offset from the start of *string1*. In other words, it determines the length of the substring starting at the beginning of *string1* that consists only of characters from *string2*. The terminating null character ('¥0') in *string1* is not included in the search range.

The `strcspn` function, which has the exactly opposite functionality, is also provided.

## Return value

The `strspn` function returns the length of the substring from the start of *string1* to the position where the first character not in *string2* appears.

This function returns zero if the first character of *string1* does not occur in *string2*, or if either *string1* or *string2* is the null string.

## See also

strchr strrchr strpbrk strcspn

### Example

```
#include    <string.h>

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

void    main( void )
{
    size_t  retval;

    /*
        This call returns 14, since the first character in
        "ABCDEFGABCDEFG1234567" that is not a character in
        "GFEDCBA" occurs at the fourteenth character.
    */
    retval = strspn( string1 , string2 );

    /*
        This call returns 0, since the character at the
        start of "ABCDEFGABCDEFG1234567" is not a character
        in the string "XYZ".
    */
    retval = strspn( string1 , "XYZ" );
}
```

# strstr **Function**

## Function

This function searches for a substring in a character string.

## Syntax

#include <string.h>

char *strstr( char **string1* , char **string2* );

*string1* String to be searched

*string2* String to search for

## Description

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

## Return value

The `strstr` function returns a pointer to the first occurrence of *string2* in *string1*.

This function returns NULL if *string2* does not appear in *string1*, or if *string1* is the null string ("").

This function returns *string1* if *string2* is the null string.

## See also

strcspn strspn strchr strrchr strpbrk

### Example

```
#include    <string.h>

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

void    main( void )
{
    char    *ptr;


    /*
        This call searches for "WORD1".
        It returns string + 0.
    */
    ptr = strstr( string , "WORD1" );

    /*
        This call searches for "WORD2".
        It returns string + 10.
    */
    ptr = strstr( string , "WORD2" );

    /*
        This call searches for "WORD3".
        It returns string + 20.
    */
    ptr = strstr( string , "WORD3" );

    /*
        This call searches for "NOTHING".
        Since it does not appear in the object string, it
        returns NULL.
    */
    ptr = strstr( string , "NOTHING" );
}
```

# strtod

**Macro/Function**

### Function

This routine converts a character string to a floating point number of type `double`.

### Syntax

#include <stdlib.h>

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

*s* Character string to be converted

*endptr* Pointer that will point to the character where the scan stopped

### Description

The `strtod` routine converts the string pointed to by *s* to a double precision floating point number and returns that value. Note that the string *s* must conform to the following syntax.

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

The symbols used have the following meanings.

| Symbol | Meaning |
|---|---|
| [*white space*] | Some number of tabs and spaces (may be omitted) |
| [*sign*] | Sign (may be omitted) |
| [*digit*] [.] [*digit*] | Character string expressing a decimal fraction (may be omitted) |
| [{e\|E} [*sign*] *digit*] | Character string expressing the exponent (may be omitted) |

At the point where `strtod` reads a character it can't recognize, it stops scanning and if *endptr* is non-null, it sets *endptr* to a pointer that indicates the position of that character. Note that if the converted value is too large to be represented by the type `double`, it returns HUGE_VAL, and sets `errno` to ERANGE.

### Return value

The `strtod` routine returns the value of the converted string in an object of type `double`.

### See also

atof atoi atol strtol strtoul

**Example**

```
#include    <stdlib.h>

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

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

# strtok **Function**

## Function

This function breaks up a string into delimited tokens, and return the tokens in order.

## Syntax

#include <string.h>

char *strtok( char **string1* , char **string2* );

*string1* Character string to be tokenized, or NULL

*string2* Character string consisting of delimiters

## Description

The term "token" as used here refers to substrings of *string1* that consist of characters other than characters from *string2*. Delimiter refers to the characters in *string2*. For example, if the delimiters are space (' '), colon (':'), and period ('.'), and the string is "RTL665: Run Time Library.", then the string would be broken up into the four tokens "RTL665", "Run", "Time", and "Library".

The `strtok` function breaks *string1* up into tokens, taking the characters in *string2* as delimiters. Pointers to the separated tokens can be acquired in order by sequential calls to this function.

If `strtok` is called with a pointer to a string (i.e., not the null pointer) in *string1*, `strtok` will read over any delimiters that may appear at the start of *string1*, and return a pointer to the first token that appears in *string1*. A null character ('¥0') will be placed at the end of this first token. NULL is returned if there are no tokens in *string1*.

If NULL is passed as *string1* to `strtok`, it searches for the next token. If another token exists, it returns a pointer to that token. A null character ('¥0') will be placed at the end of this token. NULL is returned if there are no more tokens.

The `strtok` function is normally used as follows.

(1) The string to be broken down is passed as *string1* and the first token is acquired.

(2) NULL is passed as *string1*, and the next token is acquired.

(3) Step (2) is repeated until NULL is returned.

The contents of *string2* may be changed each time `strtok` is called. This function stores a null character at the end of the token each time a token is discovered. Note that as a result, *string1* is modified.

### Return value

The `strtok` function returns a pointer to a token as long as there are tokens remaining. It returns NULL when there are no more tokens.

### See also

strcspn strspn strchr strrchr strpbrk strstr

### Example

```
/*
  This program breaks a string into tokens using spaces, commas,
  semicolons, and colons as delimiters. Pointers to these tokens are
  stored 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. Returns a pointer to the first token,
        TOKEN1.
    */
    token_ptr = strtok( string , delimiter);

    while (token_ptr != NULL)
    {
        token_stock[token_counter] = token_ptr;
        /* Save the pointer to the token. */
        ++token_counter;
        if (token_counter >= 20)
            break;
        /*
           The second and later calls. NULL is passed as the first
           argument. The calls to strtok return pointers to
           TOKEN2, TOKEN3, and TOKEN4 in that order. The loop ends
           when strtok finally returns NULL.

        */
        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

        string[] is changed to be the following.
            "   TOKEN1¥0TOKEN2¥0  TOKEN3¥0:TOKEN4¥0";

    */
}
```

# strtol

## Macro/Function

### Function

This routine converts character strings to integers of type `long`.

### Syntax

#include <stdlib.h>
long strtol( char **s*, char ***endptr*, int *base* );
*s* Character string to be converted
*endptr* Pointer that will point to the character where the scan stopped
*base* The radix

### Description

The `strtol` routine converts the string pointed to by the argument *s* to an integer of type `long`, and return that value. Note that the string must conform to the following syntax.

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

The symbols used have the following meanings.

| Symbol | Meaning |
|---|---|
| [*white space*] | Some number of tabs and spaces (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` routine converts the string *s* in radix *base* as long as *base* is in the range 2 to 36. That is, if *base* is 16, the string is interpreted in base 16 and converted to a number, with the characters '0' to '9', 'a' to 'f', and 'A' to 'F' recognized as digits. If base is 0, then the radix is determined by the first one or two characters in the digit string. The table below shows how the radix is determined.

| First Character | Second Character | Conversion radix |
|---|---|---|
| 0 | 1 to 7 | Octal |
| 0 | x or X | Hexadecimal |
| 1 to 9 | | Decimal |

The `strtol` routine returns 0 if *base* is negative, 1, or greater than 36.

At the point where `strtol` reads a character it can't recognize, it stops scanning and if *endptr* is non-null, it sets *endptr* to a pointer that indicates the position of that character. Note that if the acquired value cannot be represented by type `long`, `strtol` returns either LONG_MAX or LONG_MIN and sets `errno` to ERANGE.

### Return value

The `strtol` routine returns the converted value.

### See also

atof atoi atol strtod strtoul

### Example

```
#include    <stdlib.h>

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

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

# strtoul

## Macro/Function

### Function

This routine converts character strings to integers of type `unsigned long`.

### Syntax

#include <stdlib.h>
unsigned long strtoul( char *s, char **endptr, int base );

| | |
|---|---|
| *s* | Character string to be converted |
| *endptr* | Pointer that will point to the character where the scan stopped |
| *base* | The radix |

### Description

The `strtoul` routine converts the string pointed to by the argument *s* to an integer of type `unsigned long`, and return that value. Note that the string must conform to the following syntax.

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

The symbols used have the following meanings.

| Symbol | Meaning |
|---|---|
| [*white space*] | Some number of tabs and spaces (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` routine converts the string *s* in radix *base* as long as *base* is in the range 2 to 36. That is, if *base* is 16, the string is interpreted in base 16 and converted to a number, with the characters '0' to '9', 'a' to 'f', and 'A' to 'F' recognized as digits. If base is 0, then the radix is determined by the first one or two characters in the digit string. The table below shows how the radix is determined.

| First Character | Second Character | Conversion radix |
|---|---|---|
| 0 | 1 to 7 | Octal |
| 0 | x or X | Hexadecimal |
| 1 to 9 | | Decimal |

The `strtoul` routine returns 0 if *base* is negative, 1, or greater than 36.

At the point where `strtoul` reads a character it can't recognize, it stops scanning and if *endptr* is non-null, it sets *endptr* to a pointer that indicates the position of that character. Note that if the acquired value cannot be represented by type `unsigned long`, `strtoul` returns ULONG_MAX and sets `errno` to ERANGE.

### Return value

The `strtoul` routine returns the converted value.

### See also

atof atoi atol strtod strtol

### Example

```
#include    <stdlib.h>

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

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

# tan

**Function**

## Function

Computes the tangent of its argument.

## Syntax

#include <math.h>

double tan( double *x* );

*x* An angle in radian units

## Description

The `tan` function computes the tangent of the argument *x*.

## Return value

The `tan` function returns the tangent of the argument *x*.

## See also

acos asin atan atan2 cos sin

## Example

```
#include    <math.h>

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

    x = 0.5;

    res = tan(x);
}
```

# tanh

**Function**

### Function

Computes the hyperbolic tangent of its argument.

### Syntax

#include <math.h>

double tanh( double *x* );

*x* An angle in radian units

### Description

The `tanh` function computes the hyperbolic tangent (sinh(*x*)/cosh(*x*)) of the argument *x*.

### Return value

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

### See also

acos asin atan atan2 cos cosh sin sinh tan

### Example

```
#include    <math.h>

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

    x = 0.5;

    res = tanh(x);
}
```

# tolower

**Macro/Function**

### Function

Converts upper case characters to lower case characters.

### Syntax

#include <ctype.h>

int tolower( int *c* );

*c* A single byte character (an integer in the range 0x00 to 0xff)

### Description

The `tolower` routine converts *c* to lower case if it was an upper case character. Otherwise, it returns *c* unchanged.

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

### Return value

If *c* is an upper case character, the `tolower` routine returns the corresponding lower case character. For other values, it returns *c* unchanged.

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

### See also

The is routines toupper

**Example**

```
#include    <ctype.h>

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

void    main( void )
{
    int i;

    for ( i = 0 ; buffer[i] != '¥0' ; ++i )
    {
        buffer2[i] = tolower( buffer1[i] );
    }
    /*
        buffer2[] will have the following contents.
        "0123456789abcdefgabcdefg"
    */

}
```

# toupper

**Macro/Function**

### Function

Converts lower case characters to upper case characters.

### Syntax

#include <ctype.h>

int toupper( int *c* );

*c* A single byte character (an integer in the range 0x00 to 0xff)

### Description

The `toupper` routine converts *c* to upper case if it was a lower case character. Otherwise, it returns *c* unchanged.

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

### Return value

If *c* is a lower case character, the `toupper` routine returns the corresponding upper case character. For other values, it returns *c* unchanged.

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

### See also

The is routines tolower

**Example**

```
#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] );
    }
    /*
        buffer2[] will have the following contents.
        "0123456789ABCDEFGABCDEFG"
    */

}
```

# ultoa **Function**

## Function

Converts an integer of type `unsigned long` to a character string in the specified radix.

## Syntax

#include <stdlib.h>

char *ultoa( unsigned long *number*, char **s*, int *base* );

| | |
|---|---|
| *number* | Value to be converted |
| *s* | Buffer to store the converted string |
| *base* | The radix in which to express *number* |

## Description

The `ultoa` function converts *number* to a null terminated string, and stores the result of that conversion in *s*. The radix in which to express *number* is specified in *base*. The value of *base* must be in the range 2 to 36. The `ultoa` function sets *s* to the null string if *base* is less than 2 or greater than 36.

A buffer large enough to hold the converted string must be allocated for *s*. The maximum length of a string created by `ultoa`, including the null character, is 33 bytes.

## Return value

The `ultoa` function returns a pointer to the string *s*.

## See also

itoa ltoa

## Example

```
#include    <stdlib.h>

char    buf[33];

void    main( void )
{
    ultoa( 2147483648, buf, 10 );
}
```

# va_arg va_end va_start **Macro**

## Function

These macros implement variable argument lists.

## Syntax

#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 the 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 allow operations on variable argument lists to be implemented easily when creating functions that take a variable number of arguments.

The `va_start` macro sets *ap* to point to the start of variable argument list. The `va_start` macro must be called first.

The `va_arg` macro extracts the current argument as the type specified by *type*, and advances *ap* to the next argument. The *type* argument indicates the type that `va_arg` will return. The *ap* argument must be the same *ap* as the *ap* that was initialized by `va_start`.

After all the arguments from the argument list have been read, the `va_end` macro arranges that later processing will occur correctly. The `va_end` macro must be called last. The behavior that follows is undefined if the `va_macro` is not called.

## Return value

The `va_start` and `va_end` macros do not return values. The `va_arg` macro returns the argument currently pointed to by *ap*.

## See also

vsprintf

### Example

```
#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

This function formats data under the control of a format string and writes that formatted data to a character string.

## Syntax

#include <stdio.h>

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

*buffer* Buffer to hold the output string

*format* Format string

*arglist* Argument list pointer

## Description

The `vsprintf` function operates identically to `sprintf` except that instead of taking an argument list, they take *arglist*, which is a pointer to an argument list. The `vsprintf` function converts *arglist* according to the conversion specifiers in the format string pointed to by *format*, and write the output to the string pointed to by *buffer*.

See the description of `sprintf` for details on conversion specifiers and other aspects.

## Return value

The `vsprintf` function returns the number of bytes output to *buffer*. It returns EOF if any errors occur.

## See also

sprintf va_arg va_end va_start

**Example**

```
#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 );
}
```

# Chapter 3

# Standard Input/Output Routines Reference

This chapter describes the library routines that handle standard input/output. The routines are ordered alphabetically.

If a call to a routine includes pointers to ROM (const char *, const void *, etc.) among its arguments and the /WIN option is not specified, a special variant of the routine must be used. For further details on the naming conventions for these variants, see the appendix “Routines Accessing ROM.”

If a call to a routine includes a pointer to a stream (FILE *) among its arguments, the only possibilities for that stream are stdin, stdout, and stderr.

# fgetc

**Function**

### Function

Gets a character from a stream.

### Syntax

#include <stdio.h>

int fgetc( FILE * *stream* );

*stream* Pointer to a stream

### Description

The `fgetc` function returns the next character from the specified input stream.

### Return value

On success, `fgetc` returns the character it read converted to integer without sign extension. If the end of file is encountered or an error is detected, `fgetc` returns EOF.

### See also

fputc getc getchar ungetc

### Example

```
#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

Gets a string from a stream.

## Syntax

#include <stdio.h>

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

| | |
|---|---|
| *s* | Pointer to the area that will store the string |
| *n* | Number of characters to read |
| *stream* | Pointer to a stream |

## Description

The `fgets` function reads a string from *stream* and stores it in *s*. The read will terminate when *n*-1 characters are read or when a carriage return character is read. The `fgets` function will save the carriage return character at then end of *s*. It will add a null terminator to the end of the characters read into *s*.

## Return value

On success, `fgets` returns *s*. If the file ends or a file error occurs, then `fgets` returns NULL.

## See also

fputs gets

## Example

```
#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

Sends formatted output to a stream.

## Syntax

#include <stdio.h>

int fprintf( FILE * *stream*, char * *format* **[** ,*argument*, ... **]** );

*stream* Pointer to a stream

*format* Format string

*argument* Argument corresponding to a conversion type specifier

## Description

The `fprintf` function takes a list of arguments, converts them in accordance with corresponding conversion type specifiers in the format string specified by *format*, and outputs the formatted data to *stream*. The number of conversion type specifiers must be the number of arguments.

Refer to the `sprintf` description for details on the conversion type specifiers.

## Return value

The `fprintf` function returns the number of bytes output. If an error occurs, it will return EOF.

## See also

fscanf printf putc sprintf

## Example

```
#include <stdio.h>

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

# fputc

**Function**

## Function

Outputs a character to a stream.

## Syntax

#include       <stdio.h>

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

*c*       A character

*stream*       Pointer to a stream

## Description

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

## Return value

On success, `fputc` returns the character *c*.  If an error occurs, it will return EOF.

## See also

fgetc  putc

## Example

```
#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 string to a stream.

### Syntax

#include <stdio.h>

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

s A string

*stream* Pointer to a stream

### Description

The `fputs` function outputs the null-terminated string *s* to the specified output stream. The `fputs` function does not add a carriage return character, and it does not output the final null terminator.

### Return value

On success, `fputs` returns a true value.  On failure, it returns EOF.

### See also

fgets  gets  puts

### Example

```
#include <stdio.h>

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

# fscanf

**Function**

## Function

Scans and formats input from an input stream.

## Syntax

#include <stdio.h>

int fscanf( FILE * *stream*, char * *format* **[** ,*address*, ... **]** );

| | |
|---|---|
| *stream* | Pointer to a stream |
| *format* | Format string |
| *address* | Argument corresponding to a conversion type specifier |

## Description

The `fscanf` function scans a sequence of input fields from the stream, reading one character at a time. It then formats each field in accordance with the conversion type specifiers in the format string specified by *format*. Finally it stores the formatted input at the addresses indicated by the arguments following *format*. The number of formatting specifiers, addresses, and input fields must all be the same.

The `fscanf` function may stop scanning certain fields before it encounters the normal field terminating character (space). It may also stop input for various reasons.

Refer to the `sscanf` description for details on the conversion type specifiers.

## Return value

The `fscanf` function returns the number of input fields correctly scanned, converted, and stored. The return value will not include fields that did not store values.

## See also

printf scanf sscanf

### Example

```
#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" );
}
```

# getc

**Macro/Function**

### Function

Gets a character from a stream.

### Syntax

#include <stdio.h>

int getc( FILE * *stream* );

*stream* Pointer to a stream

### Description

The `getc` routine reads the next character from the specified input stream, and increments the stream's file pointer to point to the next character.

### Return value

On success, `getc` returns the read character converted to an integer without sign extension. If the file ends or an error occurs, then getc will return EOF.

### See also

fgetc getchar gets putc putchar ungetc

### Example

```
#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

Gets a character from the standard input (`stdin`).

### Syntax

#include <stdio.h>

int getchar( void );

### Description

The `getchar` routine returns the next character from the input stream `(stdin)`. The value of `getchar` is the same as `getc(stdin)`.

### Return value

On success, `getchar` returns the read character converted to an integer without sign extension. If the file ends or an error occurs, then `getchar` will return EOF.

### See also

fgetc getc gets putc putchar scanf ungetc

### Example

```
#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 string from the standard input (`stdin`).

### Syntax

#include <stdio.h>

char * gets( char * *s* );

*s* Pointer to an area that will store the string

### Description

The `gets` function reads a string terminated by a carriage return character from the standard input stream (`stdin`) and stores it in *s*. The carriage return character will be replaced by a null character in *s*.

The input string to `gets` may contain white space (spaces, tabs). The `gets` function will stop reading when it encounters a carriage return character, and will copy all characters read until that point to *s*.

### Return value

On success, `gets` returns *s*. On an error, it will return NULL.

### See also

fgets fputs getc puts scanf

### Example

```
#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

Sends formatted output to the standard output.

### Syntax

#include <stdio.h>

int printf( char * *format* **[**,*argument*, ...**]** );

*format* Format string

*argument* Argument corresponding to a conversion type specifier

### Description

The `printf` function converts the arguments in accordance with corresponding conversion type specifiers in the format string specified by *format*, and outputs the formatted data to the standard output. The number of conversion type specifiers must be the number of arguments.

Refer to the `sprintf` description for details on the conversion type specifiers.

### Return value

The `printf` function returns the number of bytes output. If an error occurs, it will return EOF.

### See also

fprintf fscanf putc puts scanf sprintf vprintf vsprintf

### Example

```
#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 character to a stream.

### Syntax

#include        <stdio.h>

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

*c*        A character

*stream*    Pointer to a stream

### Description

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

### Return value

On success, `putc` returns the output character *c*.  If an error occurs, it will return EOF.

### See also

fprintf  fputc  fputs  getc  getchar  printf  putchar

### Example

```
#include <stdio.h>

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

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

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

# putchar

**Macro/Function**

### Function

Outputs a character to the standard output (`stdout`).

### Syntax

#include <stdio.h>

int putchar( int *c* );

*c* A character

### Description

The `putchar` routine outputs the character *c* to the standard output. The value of `putchar(c)` is the same as `putc(c,stdout)`.

### Return value

On success, `putchar` returns the output character *c*. If an error occurs, it will return EOF.

### See also

getc getchar printf putc puts

### Example

```
#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 string to the standard output (`stdout`).

## Syntax

#include <stdio.h>

int puts( char * *s* );

*s* A string

## Description

The `puts` functions outputs the null-terminated string *s* to the standard output stream `(stdout)`, and then outputs a carriage return character at the end.

## Return value

On success, `puts` returns a true value. If an error occurs, it will return EOF.

## See also

fputs gets printf putchar

## Example

```
#include <stdio.h>

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

# scanf

**Function**

## Function

Scans the standard input stream, and inputs with formatting.

## Syntax

#include <stdio.h>

int scanf(, char * *format* **[**,*address*, ...**]** );

*format* Format string

*address* Argument corresponding to a conversion type specifier

## Description

The `scanf` function scans a sequence of input fields from the standard input stream `(stdin)`, reading one character at a time. It then formats each field in accordance with the conversion type specifiers in the format string specified by *format*. Finally it stores the formatted input at the addresses indicated by the arguments following *format.* The number of formatting specifiers, addresses, and input fields must all be the same.

Refer to the `sscanf` description for details on the conversion type specifiers.

## Return value

The `scanf` function returns the number of input fields correctly scanned, converted, and stored. The return value will not include fields that did not store values.

If `scanf` reads the end of file, then the return value will be EOF. If not even one field is stored, then the return value will be 0.

## See also

fscanf getc printf sscanf

### Example

```
#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 character back in an input stream.

### Syntax

#include <stdio.h>

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

*c* A character

*stream* Pointer to a stream

### Description

The `ungetc` function returns (pushes back) the character *c* to its specified source input stream *stream*. The *stream* must not have been opened as read-only. The character *c* will be returned from the *stream* with the next `getc` or `fread` call. One character can be pushed back while in any state. If `ungetc` is called twice without calling `getc`, then the first character pushed back will be deleted. If `fflush` is called, then all pushed back characters will be deleted from memory.

### Return value

On success, `ungetc` returns the pushed-back character code. If the operation fails, then `ungetc` will return EOF.

### See also

getc

## Example

```
#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.

## Syntax

#include <stdio.h>

int vfprintf( FILE * *stream*, char * *format*, va_list *arglist* );

*stream* Pointer to a stream

*format* Format string

*arglist* Pointer to argument list

## Description

The `vfprintf` function operates the same as `printf`, but instead of taking an argument list, it takes a pointer to an argument list.

The `vfprintf` function takes a pointer to a list of arguments, converts them in accordance with corresponding conversion type specifiers in the format string specified by *format*, and outputs the formatted data to *stream*. The number of conversion type specifiers must be the number of arguments.

Refer to the `sprintf` description for details on the conversion type specifiers.

## Return value

The `vfprintf` function returns the number of bytes output. If an error occurs, it will return EOF.

## See also

fprintf va_arg va_end va_start vprintf vsprintf

### Example

```
#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.

## Syntax

#include <stdio.h>

int vprintf( char * *format*, va_list *arglist* );

*format* Format string

*arglist* Pointer to argument list

## Description

The `vprintf` function operates the same as `printf`, but instead of taking an argument list, it takes a pointer to an argument list.

The `vprintf` function takes a pointer to a list of arguments, converts them in accordance with corresponding conversion type specifiers in the format string specified by *format*, and outputs the formatted data to *stream*. The number of conversion type specifiers must be the number of arguments.

Refer to the `sprintf` description for details on the conversion type specifiers.

## Return value

The `vprintf` function returns the number of bytes output. If an error occurs, it will return EOF.

## See also

printf va_arg va_end va_start vfprintf vsprintf

### Example

```
#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' );
}
```

# *Appendix*

# Routines Accessing Rom

OLMS-66K series microcontrollers use separate address spaces for program memory (ROM) and data memory (RAM). The CC665S language specifications assign data objects to these two address spaces according to the presence or absence of the const modifier. Objects with the modifier go into ROM; those without, into RAM.

Let us consider how functions which take pointers as arguments—`strcpy`(char **string1*, char **string2*), for example—access variables with the const modifier.

**■ Example ■**

```
char        ram_data[128];
const char  rom_data[] = "sample";
fn()
{
    strcpy( ramdata, rom_data );
}
```

Using CC665S's /WIN option assigns variables with the const modifier to the ROM WINDOW area, where the functions can access them with data memory addressing, so there is no problem. Omitting the /WIN option, however, places the two pointers in different address spaces which cannot be accessed simultaneously. Without fail, the code in the example will produce erratic results.

RTL665S copes with this problem of two different address spaces by providing special versions of the ANSI/ISO 9899 C standard library routines for calls accessing ROM.

- If routines taking pointers as arguments have names matching those in the standard, the pointers are always for the data memory space.

- Routines with names made up of a name from the standard plus a suffix starting with an underscore (_) include pointers to program memory (ROM) among their arguments. The suffixes have the following meanings.

**Suffixes and Their Meanings**

| | | Memory Space Accessed | |
|---|---|---|---|
| **Suffix** | **Number of Pointer Arguments** | **First Pointer Argument** | **Subsequent Arguments** |
| _c | 1 | ROM | |
| _cc | Two or more | ROM | ROM |
| _cd | Two or more | ROM | RAM |
| _dc | Two or more | RAM | ROM |

Let us consider some examples.

First, `atol`, a function with one pointer argument, has the following variants.

| | |
|---|---|
| atol(s) | s is a pointer to RAM. |
| atol_c(s) | s is a pointer to ROM. |

`strcmp`, a function with two pointer arguments, has the following variants.

| | |
|---|---|
| strcmp(s1 , s2) | s1 and s2 are both pointers to RAM. |
| strcmp_cc(s1 , s2) | s1 and s2 are both pointers to ROM. |
| strcmp_cd(s1 , s2) | s1 is a pointer to ROM; s2, a pointer to RAM. |
| strcmp_dc(s1 , s2) | s1 is a pointer to RAM; s2, a pointer to ROM. |

**■ Example ■**

The following program shows examples of proper usage, improper usage, and improper casts. The explanation assumes that CC665S's /WIN option is not specified.

```
#include  <string.h>
          char          *ramstr1
          char          *ramstr2
const     char          *romstr1
const     char          *romstr2

void      func( void )
{
          .
          .
          .
          /* Correct usage      */
          strcmp( ramstr1 , ramstr2 );
          strcmp_cc( romstr1 , romstr2 );
          strcmp_cd( romstr1 , ramstr2 );
          strcmp_dc( ramstr1 , romstr2 );
          .
          .
          .
          /* Incorrect usage        */
          strcmp( romstr1 , romstr2 );
          strcmp_cc( ramstr1 , ramstr2 );
          .
          .
          .

          /* Improper casts         */
          strcmp( (char *)romstr1 , (char *)romstr2 );
          .
          .
          .
}
```

Casts of the type shown in the last example are particularly dangerous. The source statements are grammatically correct, so CC665S does not issue any error message. Since the program then interprets pointers to one area (ROM) as pointers to a totally separate area (RAM), it will produce erratic results without fail.

# Routines for Accessing ROM with Pointers

The following is a listing of the ANSI/ISO 9899 C standard library routines and their variants.

| Routine | Syntax |
|---|---|
| atof | double atof(char **s*);<br>double atof_c(const char **s*); |
| atoi | int atoi(char **s*);<br>int atoi_c(const char **s*); |
| atol | long atol(char **s*);<br>long atol_c(const char **s*); |
| bsearch | void *bsearch( void **key*, void **base*, size_t *nelem*, size_t *size*,<br>int( **cmp*)( void *, void *) );<br>void *bsearch_cc( const void **key*, const void **base*, size_t *nelem*, size_t *size*,<br>int( **cmp_cc* )( const void *, const void *) );<br>void *bsearch_cd( const void **key*, void **base*, size_t *nelem*, size_t *size*,<br>int( **cmp_cd* )( const void *, void *) );<br>void *bsearch_dc( void **key*, const void **base*, size_t *nelem*, size_t *size*,<br>int( **cmp_dc* )( void *, const void *) ); |
| fprintf | int fprintf( FILE **stream*, char **format* **[**, *argument*, ...**]** );<br>int fprintf_dc( FILE **stream*, const char **format* **[**, *argument*, ...**]** ); |
| fputs | int fputs( char **s*, FILE **stream* );<br>int fputs_c( const char **s*, FILE **stream* ); |
| fscanf | int fscanf( FILE **stream*, char **format* **[**, address, ...**]** );<br>int fscanf_dc( FILE **stream*, const char **format* **[**, address, ...**]** ); |

| Routine | Syntax |
|---|---|
| memchr | void *memchr( void **region*, int *c*, size_t *count* );<br>void *memchr_c( const void **region*, int *c*, size_t *count* ); |
| memcmp | int memcmp( void **region1*, void **region2*, size_t *count* );<br>int memcmp_cc( const void **region1*, const void **region2*, size_t *count* );<br>int memcmp_cd( const void **region1*, void **region2*, size_t *count* );<br>int memcmp_dc( void **region1*, const void **region2*, size_t *count* ); |
| memcpy | void *memcpy( void **dest*, void **src*, size_t *count* );<br>void *memcpy_dc( void **dest*, const void **src*, size_t *count* ); |
| printf | int printf( char **format* [, *argument*, ...] );<br>int printf_c( const char **format* [, *argument*, ...] ); |
| puts | int puts( char **s* );<br>int puts_c( const char **s* ); |
| scanf | int scanf( char **format* [, *address*, ...] );<br>int scanf_c( const char **format* [, *address*, ...] ); |
| sprintf | int sprintf( char **buffer*, char **format* [, *argument*, ...] );<br>int sprintf_dc( char **buffer*, const char **format* [, *argument*, ...] ); |
| sscanf | int sscanf( char **string*, char **format* [, *address*, ...] );<br>int sscanf_cc( const char **string*, const char **format* [, *address*, ...] );<br>int sscanf_cd( const char **string*, char **format* [, *address*, ...] );<br>int sscanf_dc( char **string*, const char **format* [, *address*, ...] ); |
| strcat | char *strcat( char **string1*, char **string2* );<br>char *strcat_dc( char **string1*, const char **string2* ); |
| strchr | char *strchr( char **string*, int *c* );<br>const char *strchr_c( const char **string*, int *c* ); |

| Routine | Syntax |
|---|---|
| strcmp | int strcmp( char **string1*, char **string2* );<br>int strcmp_cc( const char **string1*, const char **string2* );<br>int strcmp_cd( const char **string1*, char **string2* );<br>int strcmp_dc( char **string1*, const char **string2* ); |
| strcpy | char *strcpy( char **string1*, char **string2* );<br>char *strcpy_dc( char **string1*, const char **string2* ); |
| strcspn | size_t strcspn( char **sting1*, char **string2* );<br>size_t strcspn_cc( const char **sting1*, const char **string2* );<br>size_t strcspn_cd( const char **sting1*, char **string2* );<br>size_t strcspn_dc( char **sting1*, const char **string2* ); |
| strlen | size_t strlen( char **string* ).<br>size_t strlen_c( const char **string* ). |
| strncat | char *strncat( char **string1*, char **string2*, size_t *count* );<br>char *strncat_dc( char **string1*, const char **string2*, size_t *count* ); |
| strncmp | int strncmp( char **string1*, char **string2*, size_t *count* );<br>int strncmp_cc( const char **string1*, const char **string2*, size_t *count* );<br>int strncmp_cd( const char **string1*, char **string2*, size_t *count* );<br>int strncmp_dc( char **string1*, const char **string2*, size_t *count* ); |
| strncpy | char *strncpy( char **string1*, char **string2*, size_t *count* );<br>char *strncpy_dc( char **string1*, const char **string2*, size_t *count* ); |
| strpbrk | char *strpbrk( char **string1*, char **string2* );<br>const char *strpbrk_cc( const char **string1*, const char **string2* );<br>const char *strpbrk_cd( const char **string1*, char **string2* );<br>char *strpbrk_dc( char **string1*, const char **string2* ); |

| **Routine** | **Syntax** |
|---|---|
| strrchr | char *strrchr( char *_string_, int _c_ );<br>char *strrchr_c( const char *_string_, int _c_ ); |
| strspn | size_t strspn( char *_string1_, char *_string2_ );<br>size_t strspn_cc( const char *_string1_, const char *_string2_ );<br>size_t strspn_cd( const char *_string1_, char *_string2_ );<br>size_t strspn_dc( char *_string1_, const char *_string2_ ); |
| strstr | char *strstr( char *_string1_, char *_string2_ );<br>const char *strstr_cc( const char *_string1_, const char *_string2_ );<br>const char *strstr_cd( const char *_string1_, char *_string2_ );<br>char *strstr_dc( char *_string1_, const char *_string2_ ); |
| strtod | double strtod( char *_s_, char **_endptr_ );<br>double strtod_c( const char *_s_, const char **_endptr_ ); |
| strtok | char *strtok( char *_string1_, char *_string2_ );<br>char *strtok_dc( char *_string1_, const char *_string2_ ); |
| strtol | long strtol( char *_s_, char **_endptr_, int _base_ );<br>long strtol_c( const char *_s_, const char **_endptr_, int _base_ ); |
| strtoul | unsigned long strtoul( char *_s_, char **_endptr_, int _base_ );<br>unsigned long strtoul_c( const char *_s_, const char **_endptr_, int _base_ ); |
| vfprintf | int vfprintf( FILE *_stream_, char *_format_, va_list _arglist_ );<br>int vfprintf_dc( FILE *_stream_, const char *_format_, va_list _arglist_ ); |
| vprintf | int vprintf( char *_format_, va_list _arglist_ );<br>int vprintf_c( const char *_format_, va_list _arglist_ ); |
| vsprintf | int vfprintf( char *_buffer_, char *_format_, va_list _arglist_ );<br>int vfprintf_dc( char *_buffer_, const char *_format_, va_list _arglist_ ); |

# Addendum

# Low-Level Routines

Programs using RTL665S's standard I/O routines must link in certain low-level routines.

This addendum describes these low-level routines called by the standard I/O routines.

# Introduction

Low-level routines are hardware-dependent routines that are normally called indirectly via library routines. Since the routines described in chapter 3 “Standard Input/Output Routines Reference” of the RTL665S Run-Time Library Reference call these low-level routines internally, the latter must be specified at link time. The following chart lists the library routines calling these low-level routines.

| Standard I/O Routines | Necessary Low-Level Routines |
|---|---|
| fgetc, fgets, fscanf, getc, getchar, gets, scanf | read |
| fprintf, fputc, fputs, printf, putc, putchar, puts, vfprintf, vprintf | write |

We supply sample versions of these low-level routines (read and write) to support standard input and output. Since such routines are highly hardware dependent, however, these sample routines may not always work. It is the user’s responsibility to modify or even rewrite the routines to match the user’s environment.

When modifying or rewriting these low-level routines, use the specifications starting on the next page.

# Specifications for Low-Level Routines

## read

### Function

Reads from a file.

### Syntax

| | |
|---|---|
| int | read(int *handle*, unsigned char **buffer*, int *len*); |
| *handle* | Handle for an open file |
| *buffer* | Pointer to memory area for storing the data |
| *len* | Maximum number of bytes to read |

### Description

`read` attempts to read *len* bytes from the file associated with *handle* into the buffer pointed to by *buffer*.

The sample routine takes *len* bytes from the serial port's receive buffer and stores them in the buffer pointed to by *buffer*.

### Return value

`read` returns an integer indicating the number of bytes placed in the buffer.

The sample routine contains absolutely no error processing. Expand it to return 0 on end-of-file (Ctrl-Z) and to return –1 on error.

### See also

write

# write

## Function

Writes to a file.

## Syntax

| | |
|---|---|
| int | write(int *handle*, unsigned char **buffer*, int *len*); |
| *handle* | Handle for an open file |
| *buffer* | Pointer to memory area holding the data |
| *len* | maximum number of bytes to write |

## Description

`write` attempts to write *len* bytes from the buffer pointed to by *buffer* to the file associated with *handle*.

The sample routine takes *len* bytes from the buffer pointed to by *buffer* and stores them in the serial port's transmit buffer.

## Return value

`write` returns an integer indicating the number of bytes written.

The sample routine contains absolutely no error processing. Expand it to return –1 on error.

## See also

read