intel®

System V/iRMK™ C Libraries

Order Number: 467226-001

Intel Corporation
3065 Bowers Avenue
Santa Clara, California 95052-8126

In the United States, additional copies of this manual or other Intel liberature may be obtained by writing:

Literature Distribution Center
Intel Corporation
P.O. Box 7641
Mt. Prospect, IL 60056-7641

Or you can call the following toll-free number:

1-800-548-4725

In locations outside the United States, obtain additional copies of Intel documentation by contacting your local Intel sales office. For your convenience, international sales office addresses are printed on the last page of this document. Contact your local sales office to obtain the latest specifications before placing your order.

Intel Corporation makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Intel Corporation assumes no responsibility for any errors that may appear in this document. Intel Corporation makes no commitment to update nor to keep current the information contained in this document. Intel Corporation assumes no responsibility for the use of any circuitry other than circuity embodied in an Intel product. No other circuit patent licenses are implied.

Intel software products are copyrighted by and shall remain the property of Intel Corporation. Use, duplication or disclosure is subject to restrictions stated in Intel's Software License Agreement, or in the case of software delivered to the government, in accordance with the software license agreement as defined in FAR 52.227-7013.

No part of this document may be copied or reproduced in any form or by any means without prior wirtten consent of Intel Corporation. Intel Corporation retains the right to make changes to these specifications at any time, without notice.

The following are trademarks of Intel Corporation and its affiliates and may be used only to identify Intel products. (Registered trademarks are followed by a superscripted ® .)

Above	▲	Intelevision	MICROMAINFRAME	SLD
ACE51	i	int$_e$ligent Identifier	MULTI CHANNEL	SugarCube
ACE96	i®	int$_e$ligent Programming	MULTIMODULE	SUPERCHARGER
ACE186	I²ICE	Intellec®	MultiSERVER	SatisFAXtion
ACE196	ICE	Intellink	NETPORT	SX
ACE960	iCEL	iOSP	ONCE	ToolTALK
ActionMedia	ICEVIEW	iPAT	OpenNET	UNIPATH
BITBUS	iCS	iPDS	OTP	UPI
Code Builder	iDBP	iPSC®	PRO750	VAPI
COMMputer	iDIS	iRMK	PROMPT	Visual Edge
CREDIT	iLBX	iRMX®	Promware	VLSiCEL
Data Pipeline	iMDDX	iSBC®	QUEST	WYPIWYF
DVI	iMMX	iSBX	QueX	ZapCode
ETOX	Inboard	iSDM	Quick-Erase	287
FaxBACK	Insite	iSXM	Quick-Pulse Programming	376
Genius	Intel®	Library Manager	READY-LAN	386
i486	int$_e$l®	MAPNET	RMX/80	387
i750®	Intel386	MCS®	RUPI	4-SITE
i860	int$_e$lBOS	Megachassis	Seamless	486
	Intel Certified			

IBM and PC AT are registered trademarks and PC and PC XT are trademarks of International Business Machines Corporation. XENIX, MS-DOS and Microsoft are registered trademarks of Microsoft Corporation. Ethernet is a registered trademark of Xerox Corporation. Soft-Scope is a registered trademark of Concurrent Sciences, Inc. UNIX is a registered trademark of UNIX System Laboratories, Inc.. Hazeltine and Executive 80 are trademarks of Hazeltine Corporation. TeleVideo is a trademark of TeleVideo Systems Inc. Wyse and WY-75 are registered trademarks of Wyse Technology. MetaWare and High C are registered trademarks of MetaWare, Inc. Phar Lap is a trademark of Phar Lap Software, Inc.

MIX is an acronym for Modular Interface eXtension. MIX® is a registered trademark of MIX Software, Incorporated.

Rev.	Revision History	Date
-001	Original Issue.	12/90

PREFACE

The *System V/iRMK™ C Libraries* manual tells you where to find the information you need to use the C run-time libraries. This manual supplements *C: A Reference Manual* which describes the ANSI parts of the C Libraries.

Audience

This manual is intended for the system programmer or software engineer who is developing applications. This manual assumes familiarity with the:

- System V/iRMK Kernel
- Soft-Scope III Debugger
- C programming language
- ANSI Standard C Libraries
- POSIX standard

About the Manual

This manual is organized into the following sections.

- Chapter 1 tells how to select libraries for your target environment. It also provides configuration and programming information.
- Chapter 2 discusses C library header files.
- Chapter 3 tells how to use non-ANSI library functions.

Conventions

- `Functions`, `macros`, `header file names`, `program tags`, and *`field names`* are indicated in a special font when used in text.

Related Documents

You may also need to refer to the following documents.

- *C: A Reference Manual*
  (Third Edition)
- *iC-386 User's Guide* or the user's guide for your compiler
- *Intel386™ Family System Builder User's Guide*
- *Intel386™ Family Utilities User's Guide*
- *IEEE Standard Portable Operating System Interface for Computer Environments*, IEEE Std 1003.1.1988
- *1989 American National Standard for Information Systems - Programming Language C* (ANSX3.159-1989)

CONTENTS

Chapter 1. Overview

Chapter 2. Header Files

Chapter 2. Header Files (continued)

Chapter 3. Functions

Chapter 3. Functions (continued)

Tables

OVERVIEW 1

Header files and libraries are installed on System V/iRMK™ in the following directories:

Library files: */usr/intel/rmk/system/lib*
Header files: */usr/intel/rmk/system/include*

The installed libraries and header files are customized to meet the requirements of iC-386. They may have to be modified to meet the requirements of other compilers. They make development of applications easier by providing:

- fast and efficient functions for common programming tasks
- interfaces to standard and custom execution environments
- built-in versions of some functions
- fully reentrant code

If your application uses C Library functions, do the following:

- Include the header files to put the library function declarations, type definitions, and related macro definitions in your source text.

  See also: Compiler controls and preprocessor directives, *iC-386 User's Guide* or the user's guide for your compiler

- Compile your source text to produce an object module compatible with the C Libraries.
- Bind the application object modules to the startup code and the libraries containing the functions used by the application.

  See also: BND386, *Intel386™ Family Utilities User's Guide*

In binding your program with the C Libraries, you can choose among the following:

- target-independent functions
- functions that depend on the iRMK Kernel
- functions that compile to the minimum amount of code needed to resolve external references when floating-point functionality is not used

Header Files

Calls to library functions are external references. Before using a library function in your source code, you must declare the function name and any external data types and variables used by the function. The header files contain source text for library function declarations and useful macro definitions.

To ensure that the external declarations in your source code match those in the libraries you use, include the appropriate header files in your source text. If you write your own declarations for library functions, external references to the library functions can be resolved incorrectly, causing errors during binding and unpredictable behavior during execution.

See also: Description of header files, Chapter 2

Including header files makes developing your application program easier. This is because some functions, such as those that accept `float` data type arguments, require prototypes. All function declarations in the C Library header files are correctly prototyped; including the header files ensures an appropriate match between definition and use for the functions. You can write your own external declaration for any library function or variable, but doing so does not guarantee an exact match.

Include each header file by using the `#include` preprocessor.

The header files contain two features that you can use to make your application software more efficient, as follows:

- Some functions declared with prototypes in the header files are also defined as macros in the same header files. A macro is a name that the compiler replaces with a different string (a process called expanding the macro) during preprocessing. The macro expansion replaces the function call with source text and eliminates the need to bind in the library module containing the function.

  If you prefer to use the library function rather than the macro, use the `#undef` preprocessor directive to remove the macro definition. The `#undef` directive must follow the inclusion of the header file and must precede the function call in the source text.

- Some functions are defined in the header files as built-in functions. The compiler generates in-line machine code instructions for these functions. Using a built-in function instead of generating code for a function call eliminates the overhead of call and return.

Macros

Macro definitions in the header files provide convenient names for often-used values and enable you to insert a block of source text in place of some function calls for more efficient object code.

To use an identifier that is also defined as a macro in an included header file, you can un-define the macro using the `#undef` preprocessor directive or suppress the macro name locally. To suppress the macro definition locally, enclose the identifier in parentheses. The left parenthesis prevents the compiler from recognizing the name as a macro, so the macro is not expanded and the name remains in the source text after preprocessing to be recognized as an identifier by the compiler. The following example suppresses the macro definition of `toupper` and instead calls the `toupper()` library function:

```
#include <string.h>
char a;
...
a = (toupper)("a");
```

Startup Code

The startup code initializes the library subsystems, such as input/output and memory allocation, then calls the user-written `main` function. The startup code is provided to you in both object code and ASM386 assembly language source text. You can tailor the startup code to the needs of the target environment by changing and reassembling the assembly language source text.

Startup code must match the memory model of the application that you bind. The */usr/intel/rmk/system/lib/cstartc.obj* object code file contains the startup code for a compact program. The */usr/intel/rmk/system/lib/cstarts.obj* object code file contains the startup code for a small model program. If you create an application that runs in an embedded environment, you must re-assemble the startup source code, */usr/intel/rmk/system/src/cstart.asm.*

Re-Assembling *cstart.asm*

If your application requires different or additional initialization, the module may be re-assembled. Use the following invocation formats.

For compact model applications:

```
/usr/intel/bin/asm386  \
-'"%*DEFINE (controls)(asm386 compact rom)"' \
cstart.asm
```

For small model applications:

```
/usr/intel/bin/asm386 \
-'"%*DEFINE (controls)(asm386 small ram)"' \
cstart.asm
```

Library Files

Use BND386 to bind your program with the library files. Table 1-1 shows how to select the correct libraries for binding with C compiled modules.

Table 1-1. Library Files

Model	Floating Point	Library
Small	Yes	crmks.lib
Small	No	crmksnf.lib
Compact	Yes	crmkc.lib
Compact	No	crmkcnf.lib

The `crmksnf.lib` and `crmkcnf.lib` libraries contain functions that do not use floating point numbers. If your program does not use the numeric processor, you must use these libraries.

The `crmks.lib` and `crmkc.lib` libraries require the 387™ coprocessor support libraries. The `80387n.lib`, `c1387n.lib`, and `eh387n.lib` libraries are part of these libraries. There is no support for emulation of the numeric coprocessor. If you use these libraries, the board running application must have a numeric coprocessor installed.

Programming Tips

The C Library object code includes debug information. If you use a debugger or emulator, you can access symbolic information from the library functions such as entry points, determine which library function is executing, examine the call stack, and so on. To remove the debug information from the library object code, purge the debug information using MAP386.

If your program uses external names that match library names, you must resolve all references from your own code to those names before binding the C Libraries.

See also: MAP386 and binding, *Intel386™ Family Utilities User's Guide*

Run-time Errors

Most of the library functions can produce one or more error return values, that is, a value outside of the normal range of return values for the function. Typical error return values are -1 for a function that returns INT values and NULL for a function that returns a pointer. An error value returned from a function indicates only that an error has occurred.

In addition to the error return value, most of the library functions can set the value of the *errno* external variable to provide more specific information about the cause of the error. The <errno.h> header file defines error macros that expand to the values used for *errno*. You can use the *errno* external variable to find information about the cause of the error.

The value of *errno* is useful only for the most recent occurrence of an error. Once *errno* has been set because of an error, its value does not change until the next error occurs. You can use *errno* effectively in two ways:

1. If a function can both set *errno* and return an error value, perform the following steps to test success or failure of the function:
   - Check the return value of the function.
   - If the return value indicates an error, check the value of *errno* to determine what error has occurred.
2. If a function can set *errno* but cannot return an error value, perform the following steps to test success or failure of the function:
   - Set *errno* to 0 before calling the function.
   - After the function returns, check the value of *errno*. If it is set to a non-zero value, an error has occurred. The value of *errno* provides specific information about the error.

Initialization

The C Libraries must be initialized before use. To initialize the libraries, place the function `initialize_clib` in your program. The function requires a pointer to a structure of type `CLIB_CONFIGURATION_STRUC`. The structure has the following elements:

```
typedef struct CLIB_CONFIGURATION_STRUC {
    void        *arena_ptr;
    int         arena_size;
    char        ttyname[80];
    mode_t      mask;
    KN_HOST_ID  slot;
};
```

Where:

arena_ptr A pointer to a statically allocated memory area used for dynamic memory allocation. This area must be at least 10 K-Bytes.

arena_size An `int` containing the size (in bytes) of the memory area.

ttyname[] A character array containing the name of a System V/386 tty port for usage by `stdin`, `stdout`, and `stderr`.

mask A `mode_t` value containing the file creation mask. This is used by `open()` when the O_CREATE flag is used. mode_t is defined in `sys/types.h`

slot A KN_HOST_ID value containing the slot in which the C Server is located.

C Server

The libraries perform I/O requests using the C Server that runs on the System V/386 Host in the system. The C Server is a daemon that must be started prior to booting the iRMK Host that uses it. To start the daemon, type the command **/usr/intel/rmk/bin/c_server start**. To stop the daemon, enter the command **/usr/intel/rmk/bin/c_server stop**.

The daemon uses directory */tmp* for temporary file creation. The C Server may be used by multiple iRMK Hosts and has a limit of 56 open files at one time.

Memory Management

The functions `malloc()`, `realloc()`, and `free()` allocate memory from a heap. The heap is created and maintained by calls to `sbrk()`. `sbrk()` allocates memory from the user-created memory pool. Note that memory is never returned to the memory pool until the task exists.

Each task using the libraries requires at least 10 K-bytes of memory. When creating the memory pool, thought must be given to the number of tasks that will use the C Libraries.

Environment

Environment variables are supported using `getenv()` and `putenv()`. You can build an environment by calling `putenv()`. Parameters are in the following form:

```
<environment variable>=<ascii value>
<environment variable>=<ascii value>
.......
.......
<environment variable>=<ascii value>
```

Time Functions

The `clock()` function is implemented as the number of seconds between the time the calling job first began execution and the current time. The `clock()` function does not return the actual processor time consumed by the task.

The environment variable `TZ` is used by `ctime()`, `localtime()`, `strftime()`, and `mktime()` to override the default time zone. If the environment variable `TZ` is not set, the libraries use a default timezone of `GMT` for time functions.

The value of `TZ` has the form:

```
[std[offset]][dst[offset][[start/time][end/time]]
```

Where:

`std and dst` — Three or more bytes that name the standard (`std`) or summer (`dst`) time zone. Only `std` is required. If `dst` is missing, then summer time does not apply. Upper or lower case alpha characters are allowed. The following characters are not permitted: commas (,), colons (:), minus signs (-), plus signs (+), and digits (0-9).

`offset` Indicates the value that must be added to the local time to derive the Coordinated Universal Time. `offset` has the form:

```
+hh[:mm[:ss]]
```

Minutes (`mm`, 0-59) and seconds (`ss`, 0-59) are optional. The number of hours (`hh`, 0-24) is required. If no `offset` follows `dst`, summer time is assumed to be one hour ahead of standard time.

The `offset` may be preceded by a "+" or "-" sign. If preceded by a "-" sign, the time zone is east of the Prime Meridian. A "+" sign (or no sign) indicates that time zone is west of the Prime Meridian

`rule` Indicates when to change to summer time and when to change back to standard time. The rule has the form:

```
date/time,date/time
```

`date` indicates the day for the time change. `time` has the same format as `offset`. If no entry is made for time, 02:00:00 is assumed.

Three formats can be used for the date entry. These are:

n=date
n is the zero based Julian day (0-365), leap years can be counted as it is possible to refer to February 29th.

J*n*=date
n is the Julian day (1-365). This entry must be preceded by the J.

M*m*.*n*.*d*=date
Precede the field with an M. Enter *d* day (0-6) of week *n* (1-5) of month *m*. Week 1 is the first week in which the *d*'th day occurs. Day zero is always Sunday. Week 5 specifies the last *d*'th day in month *m*.

The following is an example using a `TZ` variable:

```
TZ="PST8PDT7,100/2,200/2"
```

The field defines standard time (PST) as being offset 8 hours from Coordinated Universal Time. Summer time (PDT) is offset 7 hours. The system clock is adjusted for summer time on day 100 at 02:00. It is adjusted to standard time on day 200 at 02:00.

Task Deletion

The `exit()` and `abort()` functions delete the current task by calling the `_exit()` function. `_exit()` in turn deletes tasks using the **KN_delete_task** system call.

Examples

The */usr/intel/rmk/examples/ic386/clib.cpt* directory contains a compact model example demonstrating the use of `printf()` through the C Server. This example is bound with the 387™ Numerics Library.

The */usr/intel/rmk/examples/ic386/clib.sml* directory contains a small model example demonstrating multiple tasks using `printf()`.

***

HEADER FILES 2

General Information

This chapter describes the C Library header files. The header files contain declarations for the library functions and define related macros and data types. Table 2-1 lists the C Library header files and summarizes their contents. The header files are found in the following product directory: */usr/intel/rmk/system/include*.

Table 2-1. Header Files

Header File	Contents
<assert.h>	Diagnostics
<crmk.h>	Initialization Structure
<ctype.h>	Character Handling
<errno.h>	Errors
<fcntl.h>	File Access Mode and Status Flag Macros
<float.h>	Floating-point Limits
<io.h>	File Input/Output
<limits.h>	Limits on Integral Types
<locale.h>	Localization
<math.h>	Mathematics
<process.h>	Process Execution
<reent.h>	Reentrancy
<search.h>	Searching
<setjmp.h>	Non-local Jumps
<stdarg.h>	Variable Argument Lists
<stddef.h>	Common Definitions
<stdio.h>	Stream Input/Output
<stdlib.h>	Utility Functions
<string.h>	String Handling
<sys/types.h>	Primitive System Data Types
<term.h>	Terminal Functions
<time.h>	Date and Time

Description

The following pages describe the contents of each header file, in alphabetical order. A note at the top of some pages indicates that the header is specified in the ANSI C or IEEE POSIX standard. Some headers are not defined in any standard, and some contain a mixture of ANSI, POSIX, and/or supplementary, non-standard declarations. Each header description indicates where to find detailed information on its contents. Although some macros and types are defined in more than one header file, these definitions are protected (that is, if a macro or type is already defined, the second definition is ignored). Multiple inclusions of the headers do not harm your application.

<assert.h>

Diagnostics
ANSI

The `<assert.h>` header file defines the "function-like" `assert` macro.

See also: `assert` and `NDEBUG`, *C: A Reference Manual*

<crmk.h>

Information structure for C Library configuration

```
#ifndef _crmkh
#define _crmkh
/*lint -library */

#ifndef _typesh
#include <sys/types.h>
#endif /* _typesh */

#pragma fixedparams("initialize_clib")

/* C Library Initialization Structure */
#pragma align("_CLIB_CONFIGURATION_STRUC_")
typedef struct _CLIB_CONFIGURATION_STRUC_ {
   void       *arena_ptr;    /* ptr for memory allocation */
   int         arena_size;   /* size of arena */
   char        tty_name[80]; /* tty name for stdin, stdout, */
                             /* stderr */
   mode_t      unmask;       /*slot for UNIX C server */
   KN_HOST_ID  slot;
} CLIB_CONFIGURATION_STRUC;


int initialize_clib(CLIB_CONFIGURATION_STRUC *);

#endif /* _crmkh */
```

<ctype.h>

Character handling
ANSI

The header <ctype.h> declares functions and defines macros for character handling.

The ANSI functions declared in <ctype.h> are:

```
isalnum
isalpha
iscntrl
isdigit
isgraph
islower
isprint
ispunct
isspace
isupper
isxdigit
tolower
toupper
```

The supplementary functions include `isascii` and `isodigit`, for "mapping" integers to the ranges of ASCII codes and octal digits.

`isascii` is an SVID-defined function that tests whether a character-coded integer value is an ASCII code (i.e., between 0 and 0xFF inclusive).

`isodigit` tests whether an ASCII character (as defined above) is also an octal digit (ASCII 0 through 7).

The other supplementary functions are `_tolower` and `_toupper`, for case conversion. These are SVID-defined functions that work faster than the corresponding `tolower` and `toupper` functions, but whose behavior is undefined for "wrong-case" input.

The macros include `toascii` and macro versions of the functions mentioned in this section.

See also: Character Processing, *C: A Reference Manual*

The <errno.h> header file defines several macros that indicate the kind of error encountered and the macro errno to access the value of the last error condition. Table 2-2 lists error conditions.

Table 2-2. Error Macros

Macro	Error Condition
E2BIG	Size of argument too large
EACCES	Permission denied
EAGAIN	No more processes
EBADF	Bad file descriptor
EBUSY	Mount device busy
ECHILD	No children
EDEADLK	Resource deadlock avoided
EDEADLOCK	Locking violation
EDOM	Math argument out of domain of function
EEXIST	File already exists
EFAULT	Bad address
EFBIG	File too large
EFREE	Bad free pointer
EINTR	Interrupted function call
EINVAL	Invalid argument or operation
EIO	I/O error
EISDIR	Is a directory
EMFILE	Too many open files for process
EMLINK	Too many links
ENAMETOOLONG	Filename too long
ENFILE	System file table overflow
ENODEV	No such device
ENOENT	File or path not found
ENOEXEC	File is not executable
ENOLCK	No locks available
ENOMEM	Not enough space
ENOSPC	No space left on device
ENOSYS	Function not implemented

Table 2-2 Error Macros (continued)

Macro	Error Condition
ENOTBLK	Block device required
ENOTDIR	Not a directory
ENOTEMPTY	Directory not empty
ENOTTY	Inappropriate I/O control operation
ENXIO	No such device or address
EPERM	Operation not permitted
EPIPE	Broken pipe
ERANGE	Math result not representable
EROFS	Read-only file system
ESIGNAL	Bad signal vector
ESPIPE	Illegal seek
ESRCH	No such process
ETXTBSY	Text file busy
EXDEV	Cross-device link

<fcntl.h>

File access mode and
status flag macros
POSIX

The `<fcntl.h>` header file defines POSIX file access mode and status flag macros for use in the flag values used by calls to `open`. Tables 2-3 and 2-4 list these macros.

Table 2-3. File Access Mode Macros

Macro	Access Mode
O_RDONLY	Open for read only
O_WRONLY	Open for write only
O_RDWR	Open for read and write

Table 2-4. File Status Flag Macros

Macro	File Status
O_APPEND	File offset set to end of file before each write; data is appended.
O_CREAT	Create and open a new file, deleting any existing file with the pathname as that specified in the call to open (), unless O_EXCL is also used.
O_EXCL	Used only with O_CREAT, causes the call to open() to fail if a file with the same pathname as that specified in the call to open() already exists.
O_NONBLOCK	open() returns without waiting for the device to be ready or available.
O_TRUNC	Truncates any existing regular file with the same pathname as specified in the call to open() to zero bytes, leaving the existing mode and owner unchanged by this call.

See also: Use of file access mode and file status macros, `open` function in Chapter 3

<float.h>

Floating-point limits
ANSI

The <float.h> header file defines characteristics of the floating-point data types. Table 2-5 lists the ANSI floating-point constants and their values in C.

Table 2-5. Floating-point Limits

Macro	Value
FLT_RADIX	2
FLT_ROUNDS	0
FLT_DIG	6
FLT_EPSILON	5.96046e-8
FLT_MANT_DIG	24
FLT_MAX	3.40282e38
FLT_MAX_EXP	127
FLT_MAX_10_EXP	38
FLT_MIN	1.17549e-38
FLT_MIN_EXP	-125
FLT_MIN_10_EXP	-37
DBL_DIG	15
DBL_EPSILON	1.110223024625156e-16
DBL_MANT_DIG	53
DBL_MAX	1.797693134862315e308
DBL_MAX_EXP	1023
DBL_MAX_10_EXP	308
DBL_MIN	2.225073858507202e-308
DBL_MIN_EXP	-1021
DBL_MIN_10_EXP	-307
LDBL_DIG	15
LDBL_EPSILON	1.110223024625156e-16
LDBL_MANT_DIG	53
LDBL_MAX	1.797693134862315e308
LDBL_MAX_EXP	1023
LDBL_MAX_10_EXP	308
LDBL_MIN	2.225073858507202e-308
LDBL_MIN_EXP	-1021
LDBL_MIN_10_EXP	-307

See also: Types, *C: A Reference Manual*

<io.h>

File input/output

The <io.h> header file declares several functions for file input and output.

The functions are:

TTY	isatty
File Access	create open umask unlink
File Read and Write	close lseek read write

See also: Description of the above functions, Chapter 3

<limits.h>

Limits on integral types
ANSI

The `<limits.h>` header file contains constants that specify the ranges of integer and character types. Table 2-6 lists these constants and their values.

Table 2-6. Integral Type Range Macros

Macro	Value	Interpretation
CHAR_BIT	8	Number of bits per byte
SCHAR_MIN	-128	Minimum value of signed char
SCHAR_MAX	+127	Maximum value of signed char
UCHAR_MAX	255	Maximum value of unsigned char
CHAR_MIN	SCHAR_MIN	Default minimum value of char
	or 0	When nosignedchar control is in effect
CHAR_MAX	SCHAR_MAX	Default maximum value of char
	or UCHAR_MAX	When nosignedchar control is in effect
SHRT_MIN	-32768	Minimum value of short int
SHRT_MAX	32767	Maximum value of short int
USHRT_MAX	65535	Maximum value of unsigned short
INT_MIN	-2147483648	Minimum value of int
INT_MAX	2147483647	Maximum value of int
UINT_MAX	4294967295	Maximum value of unsigned int
LONG_MIN	-2147483648	Default minimum value of long int
	or -2**63	When long64 control is in effect
LONG_MAX	2147483647	Default maximum value of long int
	or 2**63-1	When long64 control is in effect
ULONG_MAX	4294967295	Default maximum value of unsigned long
	or 2**64-1	When long64 control is in effect

See also: Compiler controls and ranges of integral values, *iC-386 User's Guide* `<limits.h>` header, *C: A Reference Manual*

<locale.h>

Localization
ANSI

The <locale.h> header file declares a type and two functions, and defines several related macros. These facilities allow the libraries to support different alphabets, and various number, monetary quantity, date and time formats.

The type is the lconv structure, and the functions are setlocale and localeconv. The first of these functions changes certain locale-specific features of the libraries. The second obtains locale-specific information about monetary and numeric quantities.

The ANSI macros defined in <locale.h> and the result of using them are:

LC_COLLATE	behavior of strcoll and strxfrm functions change
LC_CTYPE	character handling functions change
LC_MONETARY	information returned by localeconv changes
LC_NUMERIC	decimal-point and non-monetary information returned by localeconv changes
LC_TIME	behavior of strftime function changes
LC_ALL	all behavior changes

The supplementary macro defined in <locale.h> is:

LC_MAX	maximum number of locales supported

The NULL macro is also defined in <locale.h>.

See also: Sections 13.5 and 13.6, *C: A Reference Manual*

The <math.h> header file contains function declarations, macro definitions, and a type definition for floating-point mathematics.

The type definition provides the following structure for use with a user-supplied error handler matherr:

```
struct exception  {
                   int     type;
                   char    *name;
                   double  arg1;
                   double  arg2;
                   double  retval;
                  };
```

Table 2-7 lists the macros. Of these macros, only HUGE_VAL is defined in the ANSI C standard; the others are supplementary SVID-defined constant macros.

See also: Error-handling structure type and macros, matherr function in Chapter 3

Table 2-7. Floating-point Error Codes

Macro	Indicates
DOMAIN	Domain error
HUGE_VAL	Floating-point function's result is too large for return data type
OVERFLOW	Numeric overflow range error
PLOSS	Partial (possibly unavoidable) loss of significance
SING	Math function undefined for an argument value (a "singularity")
TLOSS	Total loss of significance
UNDERFLOW	Numeric underflow range error

The ANSI functions declared in <math.h> are:

Trigonometric	acos asin atan atan2 cos sin tan
Hyperbolic	cosh sinh tanh
Exponential	exp frexp ldexp
Logarithmic	log log10 modf
Power	pow
Square Root	sqrt
Rounding	ceil floor
Absolute Value	fabs
Floating Point Remainder	fmod

See also: Mathematical Functions, *C: A Reference Manual*

The <math.h> header also includes declarations for the following supplementary Bessel, error-handling, and square functions:

```
j0,j1,jn
matherr
square
y0,y1,yn
```

The Bessel functions are SVID-defined, as are the prototype and related structure definition for the matherr function stub.

See also: Description of the above supplementary functions, Chapter 3

<process.h>

Process execution

The `<process.h>` header file declares functions supporting process execution protocols.

The functions are:

```
_exit
sleep
```

See also: Description of the above functions, Chapter 3

<search.h>

Searching
SVID

The `<search.h>` header file declares the SVID-defined functions `lfind` and `lsearch`.

See also: Description of above functions, Chapter 3

<setjmp.h>

Non-local jumps
ANSI

The `<setjmp.h>` header file declares a type and two functions for bypassing the normal call and return discipline.

The type is `jmp_buf`. Declare an array of type `jmp_buf` for holding information needed to restore a calling environment.

The functions are `setjmp` and `longjmp`.

See also: `<setjmp.h>`, *C: A Reference Manual*

<stdarg.h>

Variable argument lists
ANSI

The `<stdarg.h>` header file declares a type and defines three macros, to process arguments in functions which accept a variable number of arguments.

The type is `va_list`. The macros are `va_start`, `va_arg`, and `va_end`.

See also: `<stdarg.h>`, *C: A Reference Manual*

<stddef.h>

Common definitions

ANSI

The `<stddef.h>` header file contains definitions of commonly used standard C types and macros.

The types are `ptrdiff_t`, `size_t`, and `wchar_t`.

The macros are `NULL` (null pointer constant), `NULL` (string terminator constant), and `offsetof`.

See also: `<stddef.h>`, *C: A Reference Manual*

<stdio.h>

Stream input/output
ANSI

The <stdio.h> header file defines types and macros and declares functions for stream input and output.

The ANSI types defined in <stdio.h> are:

```
FILE
fpos_t
size_t
```

Table 2-8 lists the ANSI I/O macros.

Table 2-8. ANSI I/O Macros

Macro	Usage
NULL	Null pointer macro
EOF	End-of-file
_IOFBF	Stream fully buffered
_IOLBF	Flush buffer when full and when newline is written
_IONBF	Stream unbuffered
SEEK_SET	Seek relative to beginning of file
SEEK_CUR	Seek relative to current position in file
SEEK_END	Seek relative to end-of-file
BUFSIZ	Size of buffer
TMP_MAX	Number of unique temporary filenames
FILENAME_MAX	Maximum length of filename
L_tmpnam	Maximum length of temporary filename
FOPEN_MAX	Maximum number of open files
stdin	Standard input stream
stdout	Standard output stream
stderr	Standard error output stream

The ANSI functions declared in <stdio.h> are:

Operations on Files	remove rename tmpfile tmpnam
File Access	fclose fflush fopen freopen setbuf setvbuf
Formatted I/O	fprintf fscanf printf scanf sprintf sscanf vfprintf vprintf vsprintf
Character I/O	fgetc fgets fputc fputs getc getchar gets putc putchar puts ungetc
Direct I/O	fread fwrite

File Positioning	`fgetpos`
	`fsetpos`
	`fseek`
	`ftell`
	`rewind`

Error Handling	`clearerr`
	`feof`
	`ferror`
	`perror`

The `<stdio.h>` header defines macro versions of the following ANSI functions: `feof`, `ferror`, `getc`, `getchar`, `putc`, and `putchar`. (It also defines macro versions of the `fgetchar` and `fileno` functions listed below.)

See also: Input/Ouput Facilities, *C: A Reference Manual*

The supplementary functions declared are:

```
fcloseall
fdopen
fgetchar
fileno
flushall
fputchar
getw
putw
rmtmp
tempnam
```

The `fdopen` and `fileno` facilities are POSIX-conforming; the other supplementary facilities are non-standard.

See also: Description of the above supplementary functions, Chapter 3

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

The ANSI types are size_t and the following structures:

```
typedef struct {
                    int quot;
                    int rem;
                } div_t;

typedef struct {
                    long quot;
                    long rem;
                } ldiv_t;
```

Table 2-9 lists the ANSI macros.

Table 2-9. ANSI Utility Macros

Macro	Usage
NULL	Null pointer
MB_CUR_MAX	Maximum number of characters in multibyte character
EXIT_FAILURE	Unsuccessful termination status
EXIT_SUCCESS	Successful termination status
RAND_MAX	Maximum random number value

The ANSI functions are:

String Conversion	`atof`
	`atoi`
	`atol`
	`strtod`
	`strtol`
	`strtoul`

Pseudo-random Sequence Generation	`rand` `srand`

Memory Management	`calloc`
	`free`
	`malloc`
	`realloc`

Communication with the Environment	`abort`
	`atexit`
	`exit`
	`getenv`
	`putenv`

Searching and Sorting	`bsearch` `qsort`

Integer Arithmetic	`abs`
	`div`
	`labs`
	`ldiv`

Multibyte Characters and Strings	`mblen`
	`mbstowcs`
	`mbtowc`
	`wcstombs`
	`wctomb`

See also: Description of the above functions in *C: A Reference Manual*

Besides the ANSI functions listed above, the header file <stdlib.h> declares the following supplementary functions:

```
ecvt
fcvt
ftoa
gcvt
getopt
itoa
itoh
ltoa
ltoh
ltos
onexit
sbrk
ultoa
utoa
```

See also:

Description of the supplementary functions above (except for onexit) in Chapter 3.

The onexit function and its argument and return type onexit_t are declared in <stdlib.h> to support applications written before atexit replaced onexit in the ANSI C standard. The onexit function calls the standard atexit function, and negates its return value.

<string.h>

String handling
ANSI

The <string.h> header file declares functions for manipulating character arrays and other objects treated as character arrays, and defines the size_t type and the NULL macro. The ANSI functions include:

Copying	memcpy
	memmove
	strcpy
	strncpy
Concatenation	strcat
	strncat
Comparison	memcmp
	strcmp
	strcoll
	strncmp
	strxfrm
Search	memchr
	strchr
	strcspn
	strpbrk
	strrchr
	strspn
	strstr
	strtok
Miscellaneous	memset
	strerror
	strlen

See also: Description of the above functions in the *C: A Reference Manual*

Besides the ANSI functions listed above, the <string.h> header file declares the following supplementary functions:

```
cstr
memccpy
memicmp
strcmpi
strdup
stricmp
strlwr
strnicmp
strnset
strrev
strset
strupr
udistr
```

See also: Description of supplementary functions above, Chapter 3

<sys/types.h>

Primitive system data types
POSIX

The <sys/types.h> header file defines the following POSIX types.

Table 2-10. Primitive Types

Type	Usage
mode_t	Some file attributes
off_t	File sizes

As allowed by POSIX, additional system types are defined in <sys/types.h>. These include the ANSI size_t and time_t types, and fpos_t, which is used by the new getpos and fsetpos functions.

The <term.h> header file declares functions that extract and use capabilities from the terminal capability data base, */etc/termcap*.

```
#ifndef _term
#define _term

/*lint -library */

#pragma fixedparams("tgetent","tgetnum","tgetflag","tgetstr")
#pragma fixedparams("togoto","tputs")

int tgetent(char *, char *);
int tgetnum(char *);
int tgetflag(char *);
char *tgetstr(char *,char **);
char *tgoto(char *, int, int);
void tputs(char *, int, int(*)());

#endif /* _term */
```

<time.h>

Date and time

ANSI

The <time.h> header file defines four ANSI-standard types and five macros (two ANSI, three supplementary) and declares several functions for manipulating "time-stamps" associated with files.

the ANSI types are:

```
clock_t
size_t
time_t

struct tm {
            int tm_sec;
            int tm_min;
            int tm_hour;
            int tm_mday;
            int tm_mon;
            int tm_year;
            int tm_wday;
            int tm_yday;
            int tm_isdst;
           };
```

Table 2-11 lists the macros defined in <time.h>.

Table 2-11. Date and Time Macros

Macro	Usage
CLOCKS_PER_SEC	Processor clock ticks per second
NULL	Null pointer
daylight	Daylight savings time
timezone	Time zone
tzname	Time zone

For the System V/iRMK™ Kernel, the value of CLOCKS_PER_SEC is 100.

The ANSI functions declared in <time.h> include:

```
asctime
clock
ctime
difftime
gmtime
localtime
mktime
strftime
time
```

A supplementary POSIX-defined function, `tzset` (set time zone), is also declared in <time.h>.

See also: Time and Date Functions;
NULL, Standard Language Additions;
C: A Reference Manual

`tzset`, time macros;
Chapter 3

***

FUNCTIONS 3

Description

This chapter describes non-ANSI library functions implemented for the System V/iRMK Kernel. Where applicable, appropriate standards are referenced.

See also: *IEEE Standard Portable Operating System Interface for Computer Environments*, IEEE Std 1003.1.1988

close

Close a file
POSIX 6.3.1
Target dependent

Synopsis

```
#include <io.h>
int close (int fildes);
```

Where:

fildes is an open file descriptor.

Discussion

Use `close` to close the file associated with the file descriptor *fildes*. The file descriptor is then available for reuse.

Returns

If the call is successful, zero is returned. If the call is not successful, -1 is returned and *errno* is set.

creat

Create a file or
rewrite an existing file
POSIX 5.3.2
Target dependent

Synopsis

```
#include <io.h>
int creat (const char *path, mode_t mode);
```

Where:

path identifies the file to be created.

mode is the file mode.

Discussion

Use `creat` to create a new file and open it in the specified mode. If the file pointed to by *`path`* exists, `creat` truncates the existing file before creating and opening a new file.

Returns

On successful completion, `creat` returns the file descriptor. If an error occurs, `creat` returns `-1` and sets *`errno`*.

Cross-references

`open`

cstr, udistr

Convert C string to UDI string
Convert UDI string to C string

Synopsis

```
#include <string.h>
char *cstr (char *c_str, const char *udi_str)
char *udistr (char *udi_ptr, const char *c_ptr);
```

Where:

c_str is a pointer to a null-terminated (C) string.

udi_str is a pointer to a count-prefixed (UDI) string.

Discussion

Use `cstr` to convert the count-prefixed (Universal Development Interface-style) string pointed to by *udi_str* to a null-terminated (C language) string and store it starting at the location pointed to by *c_str*. The location pointed to by *c_str* must be large enough to hold the string and the NULL string terminator. Since count-prefixed strings are restricted to zero to 255 characters (range of the one-byte count), plus the terminating `NULL` character, *c_str* may point to an area one to 256 bytes long.

Use `udistr` to convert a `NULL`-terminated string pointed to by *c_ptr* to a count-prefixed (UDI-style) string and store it in the memory area pointed to by *udi_ptr*. The location pointed to by *udi_str* must be large enough to hold the string and the leading one-byte length field. The behavior of `udistr` for strings longer than 255 bytes is undefined.

You can use `strlen` to determine the required length of the destination buffer. For `udistr`, the buffer must be one byte longer than the value returned by `strlen`, since `strlen` returns the number of characters in the string excluding the terminating `NULL` character. The converted count-prefixed string requires the additional byte to hold the count.

The two pointers *c_ptr* and *udi_ptr* normally point to separate string buffers. However, if the arguments are identical, `udistr` and `c_str` still work correctly, converting the indicated string in place.

Returns

These functions return pointers to the converted string. There is no error return.

Cross-reference

`strlen`, *C: A Reference Manual*

ecvt, fcvt

Floating-point number to
character string conversion

Synopsis

```
#include <stdlib.h>

char *ecvt (double value, int count, int *dec, int *sign);
char *fcvt (double value, int count, int *dec, int *sign);
```

Where:

value is the floating-point number to be converted.

count is the desired number of digits in the converted string (excluding NULL).

dec is a pointer to a variable containing the implied position of the decimal point in the converted string.

sign is a pointer to a variable containing the sign of the floating-point value.

Discussion

Use `ecvt` or `fcvt` to convert *value* to a `NULL`-terminated character string. The converted string contains only digits and the terminating `NULL` character.

The *count* argument indicates "the number of digits to be stored after the implied decimal point." If there are more digits than the *count* argument, the low-order digit is rounded; if there are fewer digits of precision, the string is padded with zeros.

The converted string contains only digits. To find the position of the implied decimal point and sign, use *dec* and *sign* after the call to `ecvt` or `fcvt`.

The *dec* argument points to an integer that indicates the decimal position relative to the beginning of the string; a negative or zero value indicates a position preceding the first digit in the string.

The *sign* argument points to an integer that indicates the sign of the floating-point string. It is zero for a positive value and non-zero for a negative value.

The `ecvt` and `fcvt` functions are not re-entrant. Use `sprintf` for portability.

Returns

The `ecvt` and `fcvt` functions return a pointer to the converted string; there is no error return.

Cross-reference

`sprintf`, *C: A Reference Manual*

fcloseall

Close all open files

Synopsis

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

Discussion

Use `fcloseall` to close all currently open files. The `fcloseall` function does not close `stdin`, `stdout`, or `stderr`.

Returns

The `fcloseall` function returns the number of files closed, which may be zero or greater. There is no error return.

Cross-references

`fopen`
`stderr`
`stdin`
`stdout`

fdopen

Open a stream on
a file descriptor
POSIX 8.2.2

Synopsis

```
#include <stdio.h>
FILE *fdopen (int fildes, char *mode);
```

Where:

fildes is the file descriptor.

mode is one of the file opening modes for `fopen`, except that the "W" and "W+" modes do not cause truncation of the file.

Discussion

Use `fdopen` to open a stream and associate it with the file descriptor *`fildes`*. The file to be associated with *`fildes`* must already be open.

You cannot open a stream in a mode incompatible with the mode of the file. For example, if the file is open for writing, you cannot open the stream for reading or for updating.

Returns

On successful completion, `fdopen` returns a pointer to the stream; otherwise `fdopen` returns a `NULL` pointer, which indicates that an invalid file *`mode`* was specified.

Cross-references

See also: `fopen`, Input/Output Facilities; *C: A Reference Manual*

fgetchar

Get a character from stdin

Synopsis

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

Discussion

Use `fgetchar` to read a character from the standard input stream, `stdin`.

Returns

On successful completion, `fgetchar` returns the next character from `stdin`; otherwise, `fgetchar` returns `EOF`. Note that since `EOF` is a legal `int` value, you should use `feof` or `ferror` to check for an actual error.

Cross-references

`feof`, `ferror`, `stdin`; *C: A Reference Manual*

fileno

Map a stream pointer
to a file descriptor
POSIX 8.2.1

Synopsis

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

Where:

stream is a pointer to an open stream.

Discussion

Use `fileno` to get the file descriptor associated with the given *stream*. This function lets you use the file descriptor I/O calls (for example, `read`, `write`, `lseek`) on streams.

Note that to mix the two I/O systems (`open` vs. `fopen`, `read` vs. `fread`, etc.), you must flush all I/O buffers when going from the buffered system (for example, `fwrite`) to the unbuffered system (for example, `write`). If you omit this step, you are likely to lose data.

Returns

On successful completion, `fileno` returns the file descriptor; there is no error return.

Cross-references

`lseek`
`read`
`write`

flushall

Flush all file buffers

Synopsis

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

Discussion

Use `flushall` to write output stream buffers to the associated files and clear open input streams of their contents. The `flushall` function does not close the streams.

Returns

The `flushall` function returns the number of streams successfully flushed; there is no error return.

fputchar

Write a character to stdout

Synopsis

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

Where:

`c` is the character to be written.

Discussion

Use `fputchar` to write a character to `stdout`. This function is the same as `fputc(c,stdout)`.

Returns

On successful completion, `fputchar` returns the character written; otherwise, `fputchar` returns `EOF`. Note that since `EOF` is a legal `int` value, you should use `ferror` to check for an actual error.

Cross-references

`ferror`
`fputc`

ftoa

Convert floating-point
number to character string

Synopsis

```
#include <stdlib.h>
char* ftoa (double value, char *string, unsigned int iplaces,
            unsigned int fplaces);
```

Where:

value is the floating-point number to be converted.

string is a pointer to the string to be created by the conversion.

iplaces is the desired number of significant integer digits.

fplaces is the desired number of significant fractional digits.

Discussion

Use `ftoa` to convert the input value to a `NULL`-terminated character string string in the format `[-]iii.fffE[-]eee`.

The first argument, *value*, contains the number to be converted. The second, *string*, points to a character array into which `ftoa` writes the converted string. The *iplaces* and *fplaces* arguments, respectively, specify the number of significant integer and fractional digits.

The value of the number is truncated, not rounded. The algorithm that `ftoa` uses is accurate to eighteen significant digits. If *iplaces* plus *fplaces* exceeds eighteen, they are adjusted so that only eighteen significant digits are used.

For portability, use `sprintf`'s `%e` conversion specifier. Use the optional field width and precision to control the number of fractional digits. Note that `sprintf`'s `%e` conversion specifier produces a string in the format `[-]d.dddE+ee`; you cannot get more than one integer digit.

Returns

On completion, `ftoa` returns the number of characters in the converted string preceding the terminating `NULL` character. There is no error return.

Cross-reference

`sprintf`, *C: A Reference Manual*

gcvt

Floating-point number to
character string conversion

Synopsis

```
#include <stdlib.h>
char *gcvt (double value, int digits, char *buffer);
```

Where:

value is the floating-point number to be converted.

digits is the desired number of digits in the converted string (excluding \0).

buffer is a pointer to the buffer for the converted string.

Discussion

Use `gcvt` to convert *`value`* to a `NULL`-terminated character string which is *`digits`* characters in length. The `gcvt` function stores the string at the location pointed to by *`buffer`*. The buffer must be large enough to hold the converted string and terminating `NULL` character.

If possible, `gcvt` formats the string as in the decimal (`%f`) format of the `printf` function; otherwise, `gcvt` stores it in the exponential (`%e`) format. For portability, use `sprintf`'s `%g` conversion specifier.

Returns

The `gcvt` function returns a pointer to the converted character string. There is no error return.

Cross-reference

`sprintf`, *C: A Reference Manual*

getenv

Searches the environment
list for a string
POSIX 4.6.1

Synopsis

```
#include <stdlib.h>
char *getenv(char *name);
```

Where:

name is the string specifying an environment variable.

Discussion

The `getenv` function searches the environment list for a string of the form *name*=`value`.

Returns

Upon successful completion, the function returns a pointer to `value`. A NULL pointer is returned if *name* cannot be found.

getw

Read a word from a stream
SVID

Synopsis

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

Where:

stream is a pointer to the stream structure associated with the input stream.

Discussion

Use `getw` to read the next input word (i.e., integer) from the stream opened by `fopen` or `creat`. The apparent behavior of this function may vary due to word length and byte ordering in the environment in which the stream was written using `putw`.

Returns

On successful completion, `getw` returns the input word; otherwise, `getw` returns `EOF` as an error or end-of-file indicator.

Since the error and end-of-file indicators are both `EOF`, which can also be a valid data word, you should use `feof` and `ferror` to distinguish among end-of-file, an error, and a valid return of `EOF`.

Cross-references

`creat, ferror, feof, fopen, putw;` *C: A Reference Manual*

isatty

Determine whether the
file is a terminal device
POSIX 4.7.2
Target dependent

Synopsis

```
#include <io.h>
int isatty (int fildes);
```

Where:

fildes is a file descriptor.

Discussion

Use `isatty` to determine whether the file descriptor *`fildes`* is associated with a terminal device.

Returns

If the file descriptor is associated with a terminal device, `isatty` returns 1; otherwise, `isatty` returns 0. If *`fildes`* is an invalid file descriptor, `isatty` returns 0 and sets *`errno`*.

itoa

Convert int to
character string

Synopsis

```
#include <stdlib.h>
char* itoa (int value, char *string, int radix);
```

Where:

value is the integer to be converted.

string is a pointer to the string.

radix is the radix of *value*.

Discussion

Use `itoa` to convert the input integer *value* to the equivalent `NULL`-terminated character string and store the result in *string*. Specify the radix of *value* with the *radix* argument, which must be in the range 2-36. If *value* is negative and *radix* is 10, the first character of the stored string is the minus sign (-).

The *string* buffer must be large enough to hold the ASCII representation of the largest integer on the target system. For example, on a 386™ processor-based system, the largest signed values represented in a 32-bit integer are -2,147,483,648 and +2,147,483,647. In base 2, their binary representations are 1 and thirty-one trailing 0s, and 0 and thirty-one trailing 1's, respectively. With the sign and terminating `NULL`, the minimum buffer size would be thirty-four bytes.

Digits in the converted string representing values 10 through 36 are the characters `a` through `x`.

For portability, use `sprintf`'s `%o`, `%d`, or `%x` conversion specifiers, if *radix* is 8, 10, or 16, respectively.

Returns

The `itoa` function returns a pointer to the string. There is no error return.

Cross-reference

`sprintf`, *C: A Reference Manual*

itoh

Convert int to
character string

Synopsis

```
#include <stdlib.h>
char* int itoh (int n, char *buffer);
```

Where:

n is the integer to be converted.

buffer is a pointer to the string.

Discussion

Use `itoh` to convert the input integer *n* into the equivalent `NULL`-terminated, hexadecimal string in the buffer to which *`buffer`* points. The buffer to which *`buffer`* points must be large enough to hold the largest integer on the target system. The `itoh` function does not place a leading `NULL` in the buffer. The `itoh` function converts all hexadecimal characters to upper case.

For portability, use `sprintf`'s `%x` conversion specifier.

Returns

The `itoh` function returns a pointer to the string. There is no error return.

Cross-reference

`sprintf`, *C: A Reference Manual*

j0, j1, jn
y0, y1, yn

Bessel functions
SVID

Synopsis

```
#include <math.h>

double j0 (double x);
double j1 (double x);
double jn (int n, double x);

double y0 (double x);
double y1 (double x);
double yn (int n, double x);
```

Where:

n is the order of the Bessel function.

x is the argument to the Bessel function. For `y0`, `y1`, and `yn`, *x* must be positive.

Discussion

Use `j0`, `j1`, and `jn` to calculate the Bessel function of the first kind of the zeroth, first, and *n*th orders, respectively.

Use `y0`, `y1`, and `yn` to calculate the Bessel function of the second kind of the zeroth, first, and *n*th order, respectively, where *n* is any non-zero positive integer.

Returns

The Bessel functions return the value of `yn`(*x*) for any non-zero, positive *x* and integer `n` greater than zero.

If input argument *x* is too large in magnitude, the `j0`, `j1`, `y0`, and `y1` functions return zero and set *errno*. Also, a `TLOSS` error indicator is sent to the standard error output stream.

Negative and zero *x* arguments cause the `y0`, `y1`, and `yn` functions to return the `_HUGE_VAL` and to set *errno*. Also, a `DOMAIN` error indication is sent to the standard error output stream.

You can use `matherr` to change this error-handling behavior.

lfind, lsearch

lfind - Linear search
lsearch - Linear search and update
SVID

Synopsis

```
#include <search.h>

void *lfind (const void *key, const void *base,
          unsigned *nelp, unsigned width,
          int (*compar)(const void *, const void *));

void *lsearch (const void *key, const void *base,
          unsigned *nelp, unsigned width,
          int (*compar)(const void *, const void *));
```

Where:

key is a pointer to the value to be searched for.

base is a pointer to the first element in the array.

nelp is a pointer to the number of elements in the array.

width is a pointer to the size, in bytes, of each element in the array.

compar is the function to compare each element in the array with the *key*.

Discussion

Use the `lfind` or `lsearch` function to perform a linear search of an array of elements beginning at *base* and searching to the first occurrence of *key*. The value to which *nelp* points is the number of elements in the array and the size of each element is *width* bytes.

The comparison function *compar* tests each element (but not necessarily every byte) to identify the proper element. The comparison function is user-supplied. Its two arguments point to the elements to be compared. The comparison function must return zero if the elements are identical, and non-zero otherwise.

Unlike the ANSI `bsearch` facility, `lfind` and `lsearch` do not require the array to be sorted.

According to the SVID, *key* and *base* should be cast to pointer to `void`; the return value should be cast to pointer.

Returns

The `lfind` function returns a pointer to the first match. If `lfind` does not find a match, it returns a `NULL` pointer.

The `lsearch` function also returns a pointer to the first match. If `lsearch` does not find a match, it adds *key* to the end of the array, and returns a pointer to it. Since the `lsearch` function does not allocate space for a new element, you must ensure that space is available and that the new element does not overwrite an existing element.

Cross-reference

`bsearch`, *C: A Reference Manual*

ltoa, ltos

Convert long to character string

Synopsis

```
#include <stdlib.h>

char* ltoa (long num, char *string, int radix);
char* ltos (long num, char *string, int radix);
```

Where:

num is the integer to be converted.

string is the pointer to the string.

radix is the radix of *num*.

Discussion

Use `ltoa` or `ltos` to convert the supplied `long integer` value in *num* to the equivalent ASCII string in the *string* buffer using base *radix*, which must be in the range 2-36 decimal.

For `ltos`, *radix* can be an integer value from 2 to 16 or -2 to -16 decimal. The absolute value of *radix* is the number base of the input argument. A negative *radix* indicates that the input value is a `signed long`; a positive radix indicates an `unsigned long` input.

The buffer must be large enough to hold the largest possible number. The string is `NULL`-terminated.

For portability, use `sprintf`'s `%lo`, `%ld`, or `%lx` conversion specifiers, if *radix* is 8, 10, or 16, respectively.

Returns

The `ltoa` and `ltos` functions return a pointer to the string. There are no error returns.

Cross-reference

`sprintf`, *C: A Reference Manual*

ltoh

Convert long to character string

Synopsis

```
#include <stdlib.h>
char* ltoh (unsigned long num, char *string);
```

Where:

num is the integer to be converted.

string is the pointer to the string.

Discussion

Use `ltoh` to convert the `long integer` value in *num* into the equivalent hexadecimal string in the *string* buffer. The `ltoh` function does not place leading `NULL` characters in the buffer. The buffer must be large enough to hold the largest possible integer. The string to which *string* points is `NULL`-terminated.

For portability, use `sprintf`'s `%lx` conversion specifier.

Returns

The `ltoh` function returns a pointer to the string. There is no error return.

Cross-reference

`sprintf`, *C: A Reference Manual*

matherr

Floating-point math
error handling
SVID

Synopsis

```
#include <math.h>
int  matherr (struct exception *exp);
```

Where:

exp points to a structure containing information about the error.

Discussion

The `matherr` function is a user-supplied function that handles errors in mathematical functions. When an error occurs in a math function listed below, a pointer to the `exception` structure defined in `<math.h>` is passed to `matherr`.

Table 3-1. Functions That Call matherr

acos	exp	log10	tan
asin	floor	modf	tanh
atan	fmod	pow	y0
atan2	j0	sin	y1
ceil	j1	sinh	y2
cos	j2	sqrt	
cosh	log	square	

If you do not provide a `matherr` function, math errors are handled as described for each math function. If you want to handle certain exceptions differently, examine the `exception.type` element to determine the type of error that occurred, and perhaps the string pointed to by the `exception.name` element to determine which function detected the error. Based on this information, your implementation of `matherr` can take appropriate action.

The error type macros listed in Table 3-2 are standard values for the `exception.type` field. (Of these macros, only `HUGE_VAL` is defined in the ANSI C standard; the others are supplementary SVID-defined constant macros.)

Table 3-2. Floating-point Error Codes

Macro	Indicates
DOMAIN	Domain error
HUGE_VAL	Floating-point function's result is too large for return data type
OVERFLOW	Numeric overflow range error
PLOSS	Partial (possibly unavoidable) loss of significance
SING	Math function undefined for an argument value (a singularity)
TLOSS	Total loss of significance
UNDERFLOW	Numeric underflow range error

Each error type corresponds to one of the Numeric Data Processor (NDP) error status bits, as shown in Table 3-3.

See also: For more details on there errors, Section 3.2 of the *80387 Programmer's Reference*

Table 3-3. NDP Error Status Bits

Error	Error Status Bit
DOMAIN	Invalid operation
SING	Zero divide
OVERFLOW	Overflow
UNDERFLOW	Underflow
TLOSS	Denormalized operand
PLOSS	Precision

Returns

The `matherr` function stub provided in the C Libraries always returns zero.

memccpy

Copy characters in memory
SVID

Synopsis

```
#include <string.h>
void *memccpy (void *dest, const void *src, int c, int n);
```

Where:

dest points to the destination string.

src points to the source string.

c is the character signalling the end of the source string.

n is the number of bytes to copy.

Discussion

Use `memccpy` to copy characters from the location pointed to by *`dest`* to the location pointed to by *`src`*. The `memccpy` *`function`* copies characters until it has copied the character-coded integer value *`c`* or has copied *`n`* characters, whichever comes first.

Returns

On successful completion, `memccpy` returns a pointer to the byte in *`dest`* that follows the character; otherwise, `memccpy` returns a `NULL` pointer. There is no error return.

Cross-reference

`memcpy`, *C: A Reference Manual*

memicmp

Case-insensitive
memory comparison

Synopsis

```
#include <string.h>
int memicmp (const void *ptr1, const void *ptr2, unsigned len);
```

Where:

ptr1	points to the source string.
ptr2	points to the destination string.
len	is the number of characters to compare.

Discussion

The `memicmp` function is a case-insensitive version of the ANSI function `memcmp`. As such, `memicmp` compares *len* characters, starting at *ptr1*, with *len* characters at *ptr2*. The result indicates whether the first string is less than, equal to, or greater than the second string, ignoring the case of each string. The digits in the strings are compared lexicograpically; that is, as characters and not as values (e.g., 2 is greater than 13, but 02 is less than 13).

Returns

If the first string is lexicographically less than the second (ignoring case), `memicmp` returns a negative integer; if greater (ignoring case), a positive integer; otherwise `memicmp` returns zero.

Cross-references

`memccpy`; Chapter 3
`memchr`, `memcmp`, `memcpy`, `memset`; *C: A Reference Manual*

open

Open a file
POSIX 5.3.1
Target dependent

Synopsis

```
#include <fcntl.h>
#include <io.h>

int open (const char *path, int oflag [, mode_t mode]);
```

Where:

path points to the pathname of the file to be opened.

oflag indicates how the file is to be opened for reading and/or writing.

mode is the access mode to be set for a new file. This argument is legal, and required, only when *oflag* includes O_CREAT, described below.

Discussion

Use open to get a file descriptor which is associated with the file identified by *path*. The access modes and status flags of the open file descriptor are set according to *oflag*.

For *oflag*, you must specify one of the following access modes, defined in <fcntl.h>:

O_RDONLY Open for reading only.

O_WRONLY Open for writing only.

O_RDWR Open for reading and writing.

See also: POSIX file access mode macros, <fcntl.h> header file in Chapter 2

In addition to the required access mode, you can also use one or more of the following file status flags in *oflag*:

O_APPEND Perform all writes at the end of the file.

O_CREAT Creates a new file, deleting any existing file named *path* unless you specify O_EXCL.

O_EXCL Returns an error value instead of deleting any existing file.

O_TRUNC Truncates any existing file named *path* to zero bytes.

To use more than one status flag, you must add (+) or bitwise inclusive-OR (|) them together in the call to open.

Specify the third argument (mode_t *mode*) only if *oflag* includes O_CREAT. This argument is required with O_CREAT, but has no effect if the file identified by *path* already exists (see the discussion of O_EXCL). The *mode* argument sets the file permission bits for the file, except those set in the process's file mode creation mask (see the discussion of umask).

The open function checks the file permission specified in *mode* against the global permission mask of the program and removes from the argument any file permission that the global permission mask prohibits. Use umask to set the process's file mode creation mask.

Returns

On successful completion, open returns the lowest numbered unused file descriptor. The file descriptor is used to reference the file in calls to the isatty, close, dup, dup2, lseek, read, and write functions. If an error occurs, open returns -1 and sets *errno*.

Cross-references

```
creat
close
<fcntl.h>
fstat
isatty
lseek
read
write
umask
```

putenv

Change or add a value
to the environment
SVID

Synopsis

```
#include <stdlib.h>
int putenv (char *string);
```

Where:

string is the string specifying an environment variable and the value it should take.

Discussion

Use `putenv` to define an environment variable and its value. The string argument points to a character string in the form `name=value`. The `putenv` function alters an existing variable or creates a new one to make the value of the environment variable `name` equal to `value`. The memory pointed to by string can be used for other purposes after return from `putenv` (`putenv` makes a copy).

Returns

If the call is successful, zero is returned. If the call is not successful, a non-zero value is returned.

putw

Put a word in an output stream

SVID

Synopsis

```
#include <stdio.h>
int putw (int w, FILE *stream);
```

Where:

w	is the word (i.e., integer) to be written.
stream	is a pointer to the stream structure associated with the output stream.

Discussion

Use `putw` to write the word *w* (i.e., integer) to the specified *stream*. The least-significant byte of the word is written first.

Returns

On successful completion, `putw` returns the word written, which may be `EOF`. You can use `feof` and `ferror` to distinguish between an error and a valid return of `EOF`.

Cross-references

`feof`
`ferror`

read

Read from a file
POSIX 6.4.1
Target dependent

Synopsis

```
#include <io.h>
int read (int fildes, void *buf, unsigned nbyte);
```

Where:

fildes is the file descriptor of the file to be read from.

buf points to the buffer to receive the bytes read.

nbyte is the number of bytes to be read.

Discussion

Use `read` to read up to *`nbyte`* bytes from the file identified by *`fildes`*, and to place the characters in the buffer pointed to by *`buf`*.

Reading proceeds from the file position indicated by the file offset associated with *`fildes`*. The `read` function increments the file offset by the number of bytes written. If the file position indicated before the read operation begins is after the end of file, no bytes are read.

Returns

On successful completion, `read` returns the number of bytes placed in *`buf`*; otherwise, `read` returns -1 and sets *`errno`*.

rmtmp

Remove all temporary files

Synopsis

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

Discussion

Use `rmtmp` to close and delete any files opened by `tmpfile`.

Returns

The `rmtmp` function returns the number of files deleted.

Cross-reference

`tmpfile`

sbrk

String break
Target dependent

Synopsis

```
#include <stdlib.h>
void *sbrk (unsigned size);
```

Where:

size is the number of bytes to be acquired; must be greater than zero.

Discussion

Use `sbrk` to acquire a memory area from the target operating system.

The `malloc` function calls `sbrk` when there is not enough memory available in the heap to satisfy the allocation request. If you get memory with `sbrk`, you cannot free or reallocate it with `free` or `realloc`.

Returns

If successful, the call returns address of the memory area. If it is not successful, `NULL` is returned.

Cross-references

`free`
`malloc`
`realloc`

sleep

Suspend task execution
Target dependent

Synopsis

```
#include <process.h>
unsigned int sleep (unsigned int seconds);
```

Where:

seconds is the number of seconds to suspend a task.

Discussion

Use `sleep` to suspend a task for a specified number of seconds.

Returns

This function returns the number of seconds slept,

square

Square a number

Synopsis

```
#include <math.h>
double square (double val);
```

Where:

val is the number to be squared.

Discussion

Use `square` to calculate the square of the number *val* (i.e., *val* * *val*).

Returns

The `square` function returns the value of *val* * *val*. Errors cause `matherr` to be called.

Cross-reference

`matherr`

strcmpi, stricmp

Case-insensitive string comparison

Synopsis

```
#include <string.h>

int strcmpi (const char *s1, const char *s2);
int stricmp (const char *s1, const char *s2);
```

Where:

s1, s2 point to the strings to be compared.

Discussion

These functions are case-insensitive versions of the ANSI `strcmp` function. As such, they compare the first `NULL`-terminated string to the second and return a value based on whether the first string is lexicographically less than, greater than, or the same as the second string (ignoring case).

Use `strcmpi` or `stricmp` to lexicographically compare two strings, pointed to by *s1* and *s2*, ignoring distinctions between lower case and upper case.

Returns

The `strcmpi` and `stricmp` functions return an integer greater than, equal to, or less than zero, depending on whether the string pointed to by *s1* is lexicographically greater than, equal to, or less than the string pointed to by *s2*, ignoring case in both strings. There are no error returns.

Cross-reference

`strcmp`, *C: A Reference Manual*

strdup

Duplicate a string in memory

Synopsis

```
#include <string.h>
char *strdup (const char *s);
```

Where:

s points to a character string to be copied.

Discussion

Use `strdup` to copy the character string pointed to by `s`. Memory space for the copy is obtained from `malloc`. Use `free` to return the memory space when the program no longer needs it.

Returns

The `strdup` function returns a pointer to the duplicate string placed in memory. It returns `NULL` if `malloc` cannot allocate the required memory.

Cross-references

`free`, `malloc`; *C: A Reference Manual*

strftime

Format time string

Synopsis

```
#include <time.h>
size_t strftime (char *buffer, size_t maxsize,
     const char *format, const struct tm *timeptr);
```

Where:

buffer contains the time information string.

maxsize is the maximum number of characters to be put into *buffer*.

format specifies the format of the time information string.

timeptr points to the time information structure.

Discussion

The `strftime` function places characters into the array pointed to by *buffer* under the control of the string pointed to by *format* up to *maxsize* number of characters. The *format* argument consists of ordinary characters and directives. The `strftime` function copies ordinary characters unchanged into *buffer*.

A *format* directive consists of a percent (%) character followed by a special character as shown in Table 3-4. The `strftime` function replaces each directive with the appropriate string from the structure pointed to by *timeptr*.

Table 3-4. Time String Format Directives

Directive	Conversion string
%a	abbreviated weekday name
%A	full weekday name
%b	abbreviated month name
%B	full month name
%c	locale's appropriate date and time representation as in asctime
%d	day of month as a decimal number (01 through 31)
%H	hour as a decimal number (00 through 23)
%I	hour as a decimal number (01 through 12)
%j	day of year as decimal number (001 through 366)
%m	month as a decimal number (01 through 12)
%M	minute as a decimal number (00 through 59)
%p	either "AM" or "PM"
%S	second as a decimal number (00 through 59)
%U	week number of the year as a decimal number (00 through 53) Sunday is considered the first day of the week
%w	weekday as a decimal number 0 (Sunday) through 6
%W	week number of the year as a decimal number (00 through 53) Monday is considered the first day of the week
%x	date (Mmm dd yyyy)
%X	time (hh:mm:ss)
%y	year without the century as decimal number (00 through 99)
%Y	year with the century as decimal number
%Z	time-zone name, or by no characters if none is defined
%%	%

If the directive is not one of the recognized characters, the behavior of `strftime` is undefined.

For the conversion, `strftime` uses only the first three characters of the `TZ` string.

When the string resulting from the conversion is longer than *max* characters, including the terminating `NULL` character, `strftime` places only the first *maxsize*-1 characters in the *buffer*. The *buffer* string is then null-terminated.

Returns

The `strftime` function returns the number of characters placed in the *buffer*, not including the terminating `NULL` character, if this total (including the `NULL` character) is not greater than *max*. Otherwise, `strftime` returns `Ø` and places *maxsize*-1 characters in *buffer*, leaving the string `NULL`-terminated.

strlwr, strupr

Convert string to
lower or upper case

Synopsis

```
#include <string.h>

char *strlwr (char *s);
char *strupr (char *s);
```

Discussion

Use `strlwr` to convert any upper-case alphabetic characters in the string pointed to by *s* to lower case.

Use `strupr` to convert any lower-case alphabetic characters in the string pointed to by *s* to upper case.

These functions modify strings without moving them, so their input and return values are the same. These functions are identical to the ANSI `tolower` and `toupper` functions, but applied to an entire string rather than a single character.

Returns

The `strlwr` and `strlwr` functions return a pointer to the modified string. There are no error returns.

Cross-references

`tolower`, `toupper`; *C: A Reference Manual*

strnicmp

Case-insensitive
string comparison

Synopsis

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

Where:

s1, s2 point to the strings to be compared.

n is the maximum number of characters in the strings to be compared.

Discussion

This function is a case-insensitive version of the ANSI `strncmp` function. As such, it compares up to *n* characters of the first `NULL`-terminated string to the second and returns a value based on whether the first string is lexicographically less than, greater than, or the same as the second string (ignoring case).

Returns

The `strnicmp` function returns an integer less than, greater than or equal to zero depending on whether the first *n* characters of the string pointed to by *s1* is less than, greater than or equal to the first *n* characters of the string pointed to by *s2*.

Cross-reference

`strncmp`, *C: A Reference Manual*

strnset

Set some characters
in a string

Synopsis

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

Where:

s points to the string to be set.

c is the character-coded integer value to be assigned to characters in the string.

n is the number of characters to be set.

Discussion

The `strnset` function sets at most *n* characters of the string *s* to *c*.

Returns

The `strnset` function returns a pointer to the string. There is no error return.

Cross-reference

`strset`

strrev

Reverse order of characters in string

Synopsis

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

Where:

s points to the string to be reversed.

Discussion

Use `strrev` to reverse the order of characters in the string pointed to by `s`, leaving the terminating null character at the end.

Returns

The `strrev` function returns the pointer to the modified string. There is no error return.

strset

Set characters
in a string

Synopsis

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

Where:

s points to the string to be set.

c is the character-coded integer value to be assigned to the characters in the string.

Discussion

Use `strset` to set all the characters in the string pointed to by s, except the required terminating `NULL` character, to c.

Returns

The `strset` function returns a pointer to the string. There is no error return.

Cross-reference

`strnset`

tempnam

Choose directory for
file creation
SVID

Synopsis

```
#include <stdio.h>
char *tempnam(cont char *dir, const char *pfx);
```

Where:

dir points to the name of the directory in which the file is to be created. Note that if *dir* is NULL or does not identify a valid directory, /tmp is used as a last resort. TMPDIR can also be used to set this aspect of the user's environment.

pfx points to a string of up to five characters.

Discussion

The tempnam function generates a unique name for a temporary file in the directory of the user's choice. The pointer *dir* is used to identify the chosen directory. The function also permits the user to specify the first characters of the temporary file's name. The pointer *pfx* points to a character string (up to five characters) used as the first characters of the file name.

Returns

If the call is successful, a pointer to a unique name is returned. If the call is not successful a NULL pointer is returned.

tgetent, tgetnum, tgetflag, tgetstr, tgoto, tputs

Terminal Functions

Synopsis

```
tgetent(char *bp, char *name)

tgetnum(char *id)

tgetflag(char *id)

char * tgetstr(char *id, char **area)

char * tgoto(char *cm, int destcol, int destline)

tputs(char*cp, int affcnt, int(*outc)())
```

Where:

bp	is a pointer to a character buffer. This buffer must be at least 1024 bytes in length.
name	is a pointer to a string that contains a terminal name.
id	is a pointer to a character string. The character string names a terminal capability.
area	is a pointer to a character array.
cm	is a pointer to a cursor addressing string.
destcol	is an int to which a column number is returned.
destline	is an int to which a line number is returned.
cp	is a pointer to a string that is to be processed.
affcnt	identifies the number of lines affected by the operation.
outc()	a function called for each character in a string.

Discussion

These functions extract and use capabilities from the terminal capability data base, */etc/termcap*.

`tgetent` extracts the */etc/termcap* entry for terminal *name* and places it in the buffer at *bp*. The call returns -1 if it can not open the `termcap` file, 0 if the terminal name does match an entry in the file, and 1 if the call is successful.

If *name* is found, and the value does not begin with a slash, and the terminal type *name* is the same as the environment string TERM, the TERMCAP string is used instead of reading the termcap file. If it does begin with a slash, the string is used as a path name rather than */etc/termcap*. This can speed entry into programs that call tgetent, well as help debug new terminal descriptions.

tgetnum gets the numeric value of capability *id*.

tgetflag returns 1 if the specified capability is present in the terminal's environment, zero if it is not.

tgetstr gets the string value of capability *id*. This string is placed in the buffer at *area*. tgetstr decodes abbreviations described in termcap, except for cursor addressing and padding information.

tgoto returns a cursor addressing string decoded from *cm* to go to column *destcol* in line *destline*. It uses the external variable UP (from the up capability) and BC (if BC is given rather than BS) to avoid placing \n, CONTROL-D, or NULL in the returned string. If a return sequence is given that is not understood, than tgoto returns OOPS.

tputs decodes the padding information (extra spaces, etc.) in the string *cp*. *affcnt* gives the number of lines affected by the operation, or 1 if this is not a applicable. outc() is called to process each character.

time macros

daylight, timezone, tzname

Header and Definitions

```
#include <time.h>

#define daylight (_tzset_ptr()->_daylight)
#define timezone (_tzset_ptr()->_timezone)
#define tzname (_tzset_ptr()->_tzname)
```

Discussion

Use the `daylight` macro to access the flag that represents whether or not daylight savings time is in effect. The flag is zero for daylight savings time, and non-zero otherwise. The default value is 1, indicating that daylight savings time is not in effect.

Use the `timezone` macro to access the value that represents the difference in seconds between Coordinated Universal Time and local time.

Use the `tzname` macro to access a pair of pointers to strings which are the name of the time zone and the name of the daylight savings time. For example, `tzname[0]` could point to `PST` and `tzname[2]` could point to `PDT`. The default strings are `UCT` and `NULL`.

Cross-reference

`tzset`
POSIX standard, Sections 8.1.1, 8.3.2, B.2.3, B.8.1.1

tzset

Set the time zone
POSIX 8.3.2
Target dependent

Synopsis

```
#include <time.h>
void tzset (void);
```

Discussion

The `tzset` function uses the environment variable `TZ` to set the values of three global variables: `daylight`, `timezone`, and `tzname`. These variables are used by `localtime` to calculate the local time.

Note that `tzset` must be called before any of the three named functions, or for those functions to reflect any changes made to the `TZ` environment variable.

The parameter associated with `TZ` takes the form `PST8PDT`. The first three characters are the name of the time zone (`for example, PST`). The digit (or digits) can be signed and is the difference in hours from Coordinated Universal Time. The final three characters are optional; their inclusion indicates that daylight savings time is currently in effect, and provides the name of the time zone during daylight savings time. If `TZ` is not set, `tzset` assumes `UCT`.

The variables are as follows:

`timezone`	This value is the difference in seconds between Coordinated Universal Time and local time.
`daylight`	This value is zero if daylight savings time is in effect and non-zero otherwise.
`tzname[]`	These two pointers point to the two strings which are the name of the time zone and the name of the daylight savings time.

The default values for these variables are `28800` for `timezone`, `1` for `daylight`, and `PST` and `PDT` for `tzname`.

Returns

The `tzset` function does not return any value.

ultoa, utoa

Convert unsigned long to string
Convert unsigned int to string

Synopsis

```
#include <stdlib.h>

char *ultoa (unsigned long value, char *string, int radix);
int utoa (unsigned int value, char *string, int radix);
```

Where:

value is the value to be converted.

string is a pointer to the string.

radix is the radix of `value`.

Discussion

Use `ultoa` to convert the `unsigned long` value *`value`* to the equivalent `NULL`-terminated character string and store the result in *`string`*.

Use `utoa` to convert the `unsigned int` value *`value`* to the equivalent `NULL`-terminated character string and store the result in *`string`*.

Specify the radix of value with the *`radix`* argument, which must be in the range 2-36 decimal.

The *`string`* buffer must be large enough to hold the ASCII representation of the largest integer on the target system. For example, on a 386 processor-based system, the largest signed values represented in a 32-bit integer are -2,147,483,648 and +2,147,483,647. In base 2, their binary representations are 1 and thirty-one trailing 0s, and 0 and thirty-one trailing 1's, respectively. With the sign and terminating `NULL`, the minimum buffer size would be thirty-four bytes.

For portability, use `sprintf`'s `%lo`, `%ld`, or `%lx` conversion specifiers, if *`radix`* is 8, 10, or 16, respectively, when calling `ultoa`. Use `sprintf`'s `%o`, `%d`, or `%x` conversion specifiers, if *`radix`* is 8, 10, or 16, respectively, when calling `utoa`.

Digits in the converted string representing values 10 through 36 are the characters `a` through `x`.

Returns

The `ultoa` and `utoas` functions return pointers to the converted strings. There are no error returns.

Cross-reference

`sprintf`, *C: A Reference Manual*

umask

Set file mode creation mask
POSIX 5.3.3
Target dependent

Synopsis

```
#include <io.h>

mode_t umask (mode_t cmask);
```

Where:

cmask is the new file mode creation mask.

Discussion

Use `umask` to change the mask used by `creat`, `mkdir`, and `open` to turn off permission bits in the *mode* argument supplied to those functions. Specify *cmask* by using the permission bit flag macros defined in `<sys/stat.h>` and described in Table 2-12. Bit positions that are set in *cmask* are cleared in the mode of the created file.

Individual read, write, and execute or search permissions can be (re)set with the following flag macros, defined in `<sys/stat.h>`.

S_IRUSR	read permission bit for file owner class
S_IWUSR	write permission bit for file owner class
S_IXUSR	execute or search permission bit for file owner class
S_IRGRP	read permission bit for file group class
S_IWGRP	write permission bit for file group class
S_IXGRP	execute or search permission bit for file group class
S_IROTH	read permission bit for file other class
S_IWOTH	write permission bit for file other class
S_IXOTH	execute or search permission bit for file other class

You can use multiple permissions masks or flag macros by inclusive-ORing them together with the | operator.

Returns

The `umask` function returns the previous value of the file mode creation mask.

Cross-references

`creat`
`open`

unlink

Remove directory entries
POSIX 5.5.1
Target dependent

Synopsis

```
#include <io.h>
int unlink (const char *path);
```

Where:

path identifies a file to be deleted.

Discussion

Use `unlink` to delete a file or directory with the pathname pointed to by *path*. If the file is open, it is marked for deletion, then deleted when all connections are closed.

Returns

If the call is successful, zero is returned. If the call is not successful, -1 is returned and *errno* is set.

write

Write to a file
POSIX 6.4.2
Target dependent

Synopsis

```
#include <io.h>
int write (int fildes, const void *buf, unsigned nbyte);
```

Where:

fildes is an open file descriptor.

buf points to the buffer containing the bytes to be written to the file.

nbyte is the number of bytes to be written to the file.

Discussion

Use `write` to write *`nbytes`* bytes from the buffer pointed to by *`buf`* to the file identified by the open file descriptor *`fildes`*.

Writing proceeds from the file position indicated by the file offset associated with *`fildes`*. The `write` function increments the file offset by the number of bytes written. If the result is greater than the length of the file, the file is extended.

If there is not enough room to write `nbyte` bytes, the `write` function returns the number of bytes actually written.

The `O_APPEND` flag used with `creat` or `open` causes the offset to be set to the end of the file before writing begins.

Returns

On successful completion, `write` returns the number of bytes written to the file associated with `fildes`. This number is always less than or equal to `nbyte`. If `write` returns a number less than `nbyte`, an error occurred but some bytes were written. If `write` is unable to process any characters it returns -1 and sets *errno*.

Cross-references

`creat`
`open`

_exit

Terminates current job

Synopsis

```
#include <process.h>
void _exit(void);
```

Discussion

The _exit function terminates the current job.

Returns

If the call is successful (the system is not corrupted), the job is deleted and there is no return value. If the call is not successful (the system is corrupted), an infinite loop results.

***

INDEX

D (continued)

E

F

F (continued)

G

H

S (continued)

T

U

V

W

Y

Request For Reader's Comments

Intel's Technical Publications Departments attempt to provide publications that meet the needs of all Intel Product users. This form lets you participate directly in the publication process. Your comments will help us correct and improve our publications. Please take a few minutes to respond.

Please restrict your comments to the usability, accuracy, organization, and completeness of this publication. If you have any comments on the product that this publication describes, please contact your Intel representative.

1. Please describe any errors you found in this publication (include page number).

2. Does this publication cover the information you expected or required? Please make suggestion for improvement.

3. Is this the right type of publication for your needs? Is it at the right level? What types of publications are needed?

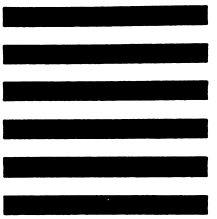
4. Did you have any difficulty understanding descriptions or wording? Where?

5. Please rate this publication on a scale of 1 to 5 (5 being the best rating). _____

Name _____ Date _____
Title _____
Company Name/Department _____
Address _____
City _____ State _____ Zipcode _____
(Country) _____ Phone _____

Please check here if you require a written reply ☐

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



BUSINESS REPLY MAIL
FIRST CLASS MAIL PERMIT NO. 79 HILLSBORO OR

POSTAGE WILL BE PAID BY ADDRESSEE

ICD TECHNICAL PUBLICATIONS HF3-72
INTEL CORPORATION
5200 NE ELAM YOUNG PARKWAY
HILLSBORO OR 97124-9978

Please fold here and close the card with tape. Do not staple.

WE'D LIKE YOUR COMMENTS....

This document is one of a series describing Intel products. Your comments on the other side of this form will help us produce better manuals. Each reply will be reviewed. All comments and suggestions become the property of Intel Corporation.

If you are in the United States and are sending only this card, postage is prepaid.

If you are sending additional material or if you are outside the United States, please insert this card and any enclosures in an envelope. Send the envelope to the above address, adding "United States of America" if you are outside the United States.

Thanks for your comments.

International Sales Offices

AUSTRALIA
Intel Australia Pty. Ltd.
Unit 13
Allambie Grove Business Park
25 Frenchs Forest Road East
Frenchs Forest, NSW 2086

BRAZIL
Intel Semicondutores do Brazil LTDA
Av. Paulista, 1159-CJS 404/405
01311 - Sao Paulo - S.P.

CANADA
Intel Semiconductor of Canada, Ltd.
4585 Canada Way, Suite 202
Burnaby V5G 4L6
British Columbia

Intel Semiconductor of Canada, Ltd.
2650 Queensview Drive
Suite 250
Ottawa K2B 8H6
Ontario

Intel Semiconductor of Canada, Ltd.
190 Attwell Drive
Suite 500
Rexdale M9W 6H8
Ontario

Intel Semiconductor of Canada, Ltd.
620 St. Jean Boulevard
Pointe Claire H9R 3K2
Quebec

CHINA/HONG KONG
Intel PRC Corporation
15/F, Office 1, Citic Bldg.
Jian Guo Men Wai Street
Beijing, PRC

Intel Semiconductor Ltd.
10/F East Tower
Bond Center
Queensway, Central
Hong Kong

DENMARK
Intel Denmark A/S
Glentevej 61, 3rd Floor
2400 Copenhagen NV

FINLAND
Intel Finland OY
Ruosilantie 2
00390 Helsinki

FRANCE
Intel Corporation S.A.R.L.
1, Rue Edison-BP 303
78054 St. Quentin-en-Yvelines
Cedex

WEST GERMANY
Intel Semiconductor GmbH
Dornacher Strasse 1
8016 Feldkirchen bei Muenchen

Intel Semiconductor GmbH
Hohenzollern Strasse 5
3000 Hannover 1

Intel Semiconductor GmbH
Abraham Lincoln Strasse 16-18
6200 Wiesbaden

Intel Semiconductor GmbH
Zettachring 10A
7000 Stuttgart 80

INDIA
Intel Asia Electronics, Inc.
4/2, Samrah Plaza
St. Mark's Road
Bangalore 560001

ISRAEL
Intel Semiconductor Ltd.
Atidim Industrial Park-Neve Sharet
P.O. Box 43202
Tel-Aviv 61430

ITALY
Intel Corporation Italia S.p.A.
Milanofiori Palazzo E
20090 Assago
Milano

JAPAN
Intel Japan K.K.
5-6 Tokodai, Tsukuba-shi
Ibaraki, 300-26

Intel Japan K.K.
Daiichi Mitsugi Bldg.
1-8889 Fuchu-cho
Fuchu-shi, Tokyo 183

Intel Japan K.K.
Bldg. Kumagaya
2-69 Hon-cho
Kumagaya-shi, Saitama 360

Intel Japan K.K.
Kawaasa Bldg., 8-9F
2-11-5, Shinyokohama
Kohoku-ku, Yokohama-shi
Kanagawa, 222

Intel Japan K.K.
Ryokuchi-Eki Bldg.
2-4-1 Terauchi
Toyonaka-shi, Osaka 560

Intel Japan K.K.
Shinmaru Bldg.
1-5-1 Marunouchi
Chiyoda-ku, Tokyo 100

Intel Japan K.K.
Green Bldg.
1-16-20 Nishiki
Naka-ku, Nagoya-shi
Aichi 450

KOREA
Intel Technology Asia, Ltd.
16th Floor, Life Bldg.
61 Yoido-Dong, Youngdeungpo-Ku
Seoul 150-010

NETHERLANDS
Intel Semiconductor B.V.
Postbus 84130
3099 CC Rotterdam

NORWAY
Intel Norway A/S
Hvamveien 4-PO Box 92
2013 Skjetten

SINGAPORE
Intel Singapore Technology, Ltd.
101 Thomson Road #21-05/06
United Square
Singapore 1130

SPAIN
Intel Iberia S.A.
Zurbaran, 28
28010 Madrid

SWEDEN
Intel Sweden A.B.
Dalvagen 24
171 36 Solna

SWITZERLAND
Intel Semiconductor A.G.
Zuerichstrasse
8185 Winkel-Rueti bei Zuerich

TAIWAN
Intel Technology Far East Ltd.
8th Floor, No. 205
Bank Tower Bldg.
Tung Hua N. Road
Taipei

UNITED KINGDOM
Intel Corporation (U.K.) Ltd.
Pipers Way
Swindon, Wiltshire SN3 1RJ