Palm OS® Protein C/C++ Compiler Language & Library Reference

Palm OS® Developer Suite

CONTRIBUTORS

Written by Eric Shepherd.
Engineering contributions by Kenneth Albanowski, Flash Sheridan, and Chris Tate.

Palm OS Protein C/C++ Compiler Language & Library Reference
Document Number 3124-002
November 9, 2004
For the latest version of this document, visit
http://www.palmos.com/dev/support/docs/

PalmSource, Inc.
1240 Crossman Avenue
Sunnyvale, CA 94089
USA
www.palmsource.com

Table of Contents

Part II: C/C++ Compiler Library Reference

About This Book

This book provides reference information about the C/C++ language and runtime libraries used with the Palm OS® compiler tools. The audience for this book is application developers creating Palm OS Protein ARM-native applications and shared libraries using either the C or C++ programming languages for ARM-based handheld devices.

This book assumes you're already familiar with the C and C++ programming languages. Its goal is to familiarize you with the specific capabilities of the compiler provided as part of the Palm OS Developer Suite.

If you're unfamiliar with C or C++, or need a good reference for these languages, we recommend the following books, which are the defacto standard references for the languages:

- *The C Programming Language, 2nd Edition* by Brian W. Kernighan, Dennis Ritchie, and Dennis M. Ritchie. ISBN 0-13-1103628.
- *C: A Reference Manual*, Fifth Edition, by Samuel P. Harbison, Prentice Hall, Inc., 2002, ISBN 0-13-089592.
- *The C++ Programming Language, Special 3rd Edition* by Bjarne Stroustrup. ISBN 0-20-1700735.

How This Book Is Organized

This book is divided into two parts, a language reference and a library reference.

Part I, "C/C++ Compiler Language Reference," has the following organization:

- Chapter 1, "Language Overview," on page 3 describes the technical requirements, language extensions, and limitations of the Palm OS compiler.
- Chapter 2, "Language Elements," on page 11 describes the Palm OS compiler's C/C++ language differences, as compared to the ANSI standard.

Part II, "C/C++ Compiler Library Reference," has the following organization:

- Chapter 3, "STLport/iostream," on page 25 describes the STLport implementation of the C++ standard template library.
- Chapter 4, "Palm OS-Specific Libraries," on page 27 describes general library information.
- Chapter 5, "Runtime Library Functions," on page 29 describes the supported and unsupported runtime functions.
- The chapters that follow, beginning with Chapter 6, "assert.h," on page 39 each describe a specific header file and the supported structures, runtime functions, and macros defined within that header file.

Palm OS Developer Suite Documentation

The following tools books are part of the Palm OS Developer Suite:

Document	Description
Introduction to Palm OS Developer Suite	Provides an overview of all of the Palm OS development tools: • compiler tools • resource tools • testing and debugging tools
Palm OS Protein C/C++ Compiler Tools Guide	Describes how to use the Palm OS compiler tools: • pacc – compiler • paasm – assembler • palink – linker • palib – librarian • PSLib – the Palm OS shared library tool, including information about shared library definition (SLD) files • PElf2Bin – Palm OS post linker • ElfDump – diagnostic tool

Document	Description
Palm OS Resource Tools Guide	Describes how to use the Palm OS resource tools: • GenerateXRD – migration tool • Palm OS Resource Editor – XRD editor • PalmRC – building tool • PRCMerge – building tool • PRCCompare – comparison tool • hOverlay – localization tool • PRCSign and PRCCert – code-signing tools
Palm OS Debugger Guide	Describes how to use the Palm OS Debugger.
Palm OS Resource File Formats	Describes the XML formats used for XML resource definition (XRD) files. XRD files are used to define Palm OS resources and are the input files for the Palm OS resource tools.
Palm OS Cobalt Simulator Guide	Describes how to use the Palm OS Cobalt Simulator.
Palm OS Virtual Phone Guide	Describes how to use Virtual Phone.

Additional Resources

- Documentation

  PalmSource publishes its latest versions of this and other documents for Palm OS developers at

  http://www.palmos.com/dev/support/docs/

- Training

  PalmSource and its partners host training classes for Palm OS developers. For topics and schedules, check

  http://www.palmos.com/dev/training

- Knowledge Base

  The Knowledge Base is a fast, web-based database of technical information. Search for frequently asked questions (FAQs), sample code, white papers, and development documentation at

  http://www.palmos.com/dev/support/kb/

Part I
C/C++ Compiler Language Reference

This part is organized into the following chapters:

1
Language Overview

The Palm OS® Protein C/C++ Compiler is a full-featured, standards-based, optimizing C/C++ compiler.

- The Palm OS compiler supports the C language standard ANSI/ISO/IEC 9899:1999, commonly known as C99, as a freestanding implementation. The compiler uses this language by default for C code.

  It is required that you understand both the ANSI/ISO standard C language and library. The ANSI/ISO 9899:1999 C standards document completely describes the standard C library functions, as do several widely-used references including:

  – *The C Programming Language*, Second Edition, by Brian W. Kernighan and Dennis M. Ritchie, Prentice Hall, Inc., 1988, ISBN 0-13-1103628.

  – *C: A Reference Manual*, Fifth Edition, by Samuel P. Harbison, Prentice Hall, Inc., 2002, ISBN 0-13-089592.

  – Online at www.dinkumware.com/refxc.html.

- The Palm OS compiler supports the C++ language standard ANSI/ISO/IEC 14882:1998(E). The C++ language standard is also documented in other widely-used references including *The C++ Programming Language*, Third Edition, by Bjarne Stroustrup, Addison-Wesley, 2000, ISBN 0-20-1700735.

The Palm OS compiler takes as input one or more C and/or C++ language text files (written according to the standards above) and produces a corresponding number of assembly language source files (see the *Palm OS Protein C/C++ Compiler Tools Guide* for more details).

C Technical Requirements

In addition to the ANSI/ISO/IEC requirements previously mentioned, the C facilities of the Palm OS compiler meet the following additional technical requirements:

- Supports a variety of useful extensions to the base language, particularly those useful to the ARM architecture.
- Supports compiling with extensions removed that are incompatible with the appropriate ANSI specification.
- Produces code for the ARM instruction set for version 4T architecture microprocessors as defined in the *ARM Reference Manual*, revision E.
- Adheres to the C calling conventions of the base standard ABI for the ARM architecture.
- Adheres to the shared library conventions documented in the *ARM-Thumb Shared Library Architecture* (ASHLA, document number MADEIRA-0020-CUST-DDES A-01).
- Produces DWARF version 1.1 debugging information, if debugging output is requested.

C++ Technical Requirements

In addition to the ANSI/ISO/IEC requirements previously mentioned, the C++ facilities of the Palm OS compiler meet the following additional technical requirements:

- Adheres to the C++ calling conventions of the base standard ABI for the ARM architecture.

Limitations

There are restrictions on some of the newer, more complex, and more exotic features of the relevant standards.

Restrictions on C99

The C99 implementation is limited is the following ways:

- Complex arithmetic is not supported, and thus all usages of the `_Complex` or `_Imaginary` types are unsupported. This includes:
  - `float _Complex`
  - `double _Complex`
  - `long double _Complex`
  - `float _Imaginary`
  - `double _Imaginary`
  - `long double _Imaginary`
- Avoid use of the `long double` type in the Simulator environment. It is unsupported and should not be used. There is a binary compatibility problem with i386 gcc and the compiler used to build the Simulator.
- Floating-point environment control is not available, therefore the `__STDC_IEC_559__` and `__STDC_IEC_559_COMPLEX__` macros are not defined.
- Variable length arrays are available, however during debugging, the array length may *not* be available. The allocation of local VLAs is implemented via calls to `malloc()` and `free()`.

Restrictions on C++

The C++ implementation is limited is the following way:

- Exported templates are not supported.
- The `long long` type is an extension to C++, not part of the standard. If you wish to disable support for it, you can use the `-strict` option.

Restrictions on Data Types

Table 1.1 lists the maximum and minimum sizes of the integral data types supported by the Palm OS Protein C/C++ Compiler.

Table 1.1 Maximum and minimum sizes of integer types

Constant	Value	Description
`CHAR_BIT`	8	Number of bits in a byte.
`CHAR_MAX`	255	Maximum value of an object of type `char`.
`CHAR_MIN`	0	Minimum value of an object of type char.
`INT_MAX`	+2147483647	Maximum value of an object of type `int`.
`INT_MIN`	-2147483648	Minimum value of an object of type `int`.
`LONG_MAX`	+2147483647	Maximum value of an object of type `long int`.
`LONG_MIN`	-2147483648	Minimum value of an object of type `long int`.
`LLONG_MAX`	+9223372036854775807	Maximum value of an object of type `long long int`.
`LLONG_MIN`	-9223372036854775808	Minimum value of an object of type `long long int`.
`MB_LEN_MAX`	1	Maximum number of bytes in a multibyte character, regardless of locale. **NOTE:** This value should not be relied upon. Its use is not recommended.
`SCHAR_MAX`	+127	Maximum value of an object of type `signed char`.
`SCHAR_MIN`	-128	Minimum value of an object of type `signed char`.

Table 1.1 Maximum and minimum sizes of integer types

Constant	Value	Description
`SHRT_MAX`	+32767	Maximum value of an object of type `short int`.
`SHRT_MIN`	-32768	Minimum value of an object of type `short int`.
`UCHAR_MAX`	255	Maximum value of an object of type `unsigned char`.
`USHRT_MAX`	65535	Maximum value of an object of type `unsigned short`.
`UINT_MAX`	4294967295	Maximum value of an object of type `unsigned int`.
`ULONG_MAX`	4294967295	Maximum value of an object of type `unsigned long int`.
`ULLONG_MAX`	18446744073709551615	Maximum value of an object of type `unsigned long long int`.

Table 1.2 lists constants that describe the attributes of floating-point data types.

Table 1.2 Constants describing attributes of floating-point numeric types

Constant	Value	Description
`FLT_ROUNDS`	1	Rounding is always performed toward the nearest integral value.
`DBL_DIG`	15	The number of digits of decimal precision for type `double`.
`DBL_MANT_DIG`	53	The number of bits used to represent the mantissa for type `double`.
`DBL_MAX_10_EXP`	308	The maximum decimal exponent for type `double`.

Table 1.2 Constants describing attributes of floating-point numeric types

Constant	Value	Description
`DBL_MAX_EXP`	1024	The maximum binary exponent for type `double`.
`DBL_MIN_10_EXP`	-308	The minimum decimal exponent for type `double`.
`DBL_MIN_EXP`	-1021	The minimum binary exponent for type `double`.
`DECIMAL_DIG`	17	The number of decimal digits to which any floating-point number of type `long double` can be rounded, and back, without changing its value.
`FLT_DIG`	6	The number of decimal digits of precision for type `float`.
`FLT_MANT_DIG`	24	The number of bits in the mantissa for type `float`.
`FLT_MAX_10_EXP`	38	The maximum decimal exponent for type `float`.
`FLT_MAX_EXP`	128	The maximum binary exponent for type `float`.
`FLT_MIN_10_EXP`	-37	The minimum decimal exponent for type `float`.
`FLT_MIN_EXP`	-125	The minimum binary exponent for type `float`.
`FLT_RADIX`	2	The exponent radix.
`LDBL_DIG`	15	The number of decimal digits of precision for type `long double`.
`LDBL_MANT_DIG`	53	The number of bits in the mantissa for type `long double`.

Table 1.2 Constants describing attributes of floating-point numeric types

Constant	Value	Description
`LDBL_MAX_10_EXP`	308	The maximum decimal exponent for type `long double`.
`LDBL_MAX_EXP`	1024	The maximum binary exponent for type `long double`.
`LDBL_MIN_10_EXP`	-308	The minimum decimal exponent for type `long double`.
`LDBL_MIN_EXP`	-1021	The minimum binary exponent for type `long double`.
`DBL_MAX`	1.79769313486231571e+308	The maximum value that can be represented by type `double`.
`FLT_MAX`	3.40282347e+38	The maximum value that can be represented by type `float`.
`LDBL_MAX`	1.79769313486231571e+308	The maximum value that can be represented by type `long double`.
`DBL_EPSILON`	2.2204460492503131e-16	The smallest value such that 1.0 + `DBL_EPSILON` is not equal to 1.0 for type `double`.
`DBL_MIN`	2.22507385850720138e-308	The minimum value that can be represented by type `double`.
`FLT_EPSILON`	1.19209290e-7	The smallest value such that 1.0 + `FLT_EPSILON` is not equal to 1.0, for type `float`.
`FLT_MIN`	1.17549435e-38	The minimum value that can be represented by type `float`.
`LDBL_EPSILON`	2.2204460492503131e-16	The smallest value such that 1.0 + `DBL_EPSILON` is not equal to 1.0 for type `long double`.
`LDBL_MIN`	2.22507385850720138e-308	The minimum value that can be represented by type `long double`.

2
Language Elements

This chapter describes the Palm OS® compiler's C/C++ language differences, as compared to the ANSI standard. The following language elements of C and C++ are described:

- Lexical Elements
- Preprocessor Directives
- Predefined Constants

Lexical Elements

This section describes the following lexical elements of C and C++:

- Character Set
- Comments
- Tokens
- Identifiers
- Keywords
- Constants
- Operators
- Separators

Character Set

The Palm OS compiler only specifically supports the ASCII character set for input, although the compiler is intended to be 8-bit neutral. The following lists the basic character set that is available at both compile and run time:

- The uppercase and lowercase letters of the English alphabet

  a b c d e f g h i j k l m n o p q r s t u v w x y z

  A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- The decimal digits 0 through 9

  0 1 2 3 4 5 6 7 8 9
- The following graphic characters:

  `! " # % & ' ( ) * + , - . / : ; < > ? [ \ ] _ { } ~`
- The caret (^) character
- The split vertical bar (|) character
- The space character (' ')
- The control characters representing newline, horizontal tab, vertical tab, and form feed.

The number sign (#) character is used for preprocessing only, and the underscore (_) character is treated as a normal letter.

Comments

The following comments within C/C++ source code are permitted:

- The `/*` (slash, asterisk) characters, followed by any sequence of characters (including newlines), followed by the `*/` (asterisk, slash) characters.
- The `//` (two slashes) characters followed by any sequence of characters. A newline not immediately preceded by a line-continuation (\) character terminates this form of comment. This kind of comment is commonly called a single-line comment.

You can put comments anywhere the language allows white space.

The Palm OS compiler also recognizes the following comments within C/C++ source code, used to affect warning messages generated by the compiler:

`/*ARGSUSED*/`
: When placed before a function definition, this comment suppresses compiler warnings about unused parameters in functions.

`/*NOTREACHED*/`
: When inserted at the beginning of a block of code that appears unreachable by the compiler, this comment suppresses the “unreachable code” warning.

Tokens

Source code is treated during preprocessing and compilation as a sequence of tokens. There are five different types of tokens:

- Identifiers
- Keywords
- Constants
- Operators
- Separators

Adjacent identifiers, keywords, and literals must be separated with white space. Other tokens should be separated by white space to make the source code more readable. White space includes blanks, horizontal and vertical tabs, newlines, form feeds, and comments.

Identifiers

An identifier consists of an arbitrary number of letters or digits; however, it must not begin with a digit and it must not have the same spelling as a keyword. Identifiers provide names for the following language elements:

- Functions
- Data objects
- Labels
- Enumerated tags
- Variables
- Macros
- Typedefs
- Structure members
- Union members

Keywords

Keywords are identifiers reserved by the language for special use.

- Refer to the C language standard: ANSI/ISO/IEC 9899:1999 specification for a list of the keywords common to the C language.
- Refer to the C++ language standard: ANSI/ISO/IEC 14882:1998 specification for a list of the keywords common to the C++ language.

Extension keywords

The Palm OS compiler also recognizes the following keywords:

`__align(`*`n`*`)`

n may be 1, 2, 4, 8, or 16. When applied to a global object, guarantees that the object is emitted with at least the specified alignment. When applied to a type declaration (e.g., `typedef` or `struct`), applies to all global objects that are instances of that type. *Note:* This keyword does not alter the packing within a structure or modify what code is used to access through a pointer. Use `__pack` or `#pragma pack` for the former, and `__packed` for the latter.

`asm`

The `asm` keyword is used to pass information through the compiler to the assembler. The Palm OS compiler permits assembler code to be inlined using the keywords `asm`, `_asm`, and `__asm`. The `asm` keyword has its normal C99 and C++ behavior; in addition, when used as the first keyword in a function definition, the contents of the function are all taken as assembly instructions and the function is emitted "naked," without a prologue or epilogue that pushes or pops registers from the stack. (A 'bx lr' return instruction is placed after your code, in case you do not explicitly return.) An `asm` function is called in the same way as any function; its arguments are in registers r0-r3 and on the stack, as is defined by ATPCS:

```
asm int func (int a, int b) {
  add r0, r0, r1 // return a+b
}
```

The “inline” qualifier can be used with `asm` functions to indicate that the body of the `asm` function should be inserted at each call-site. (The `asm` function should not explicitly return or use labels. As in the above example, it should fall off the end to return execution to the caller.)

Supported use of `asm` routines is limited to “nop,” as an inline `asm` statement and relatively small `asm` functions that do not use labels.

`__asm`
: Followed by curly brackets, indicates a multi-line inline assembly block. Otherwise, indicates inline assembly until the end of the current line.

`__inline`
: An exact alias of the normal `inline` keyword, in C99 or C++, depending on which is being compiled.

`__int64`
: Alias for `long long` type.

`__pack(`*n*`)`
: *n* may be 0, 1, 2, 4, 8, or 16. Applied to a structure definition, this keyword changes the packing in effect for that structure. This keyword overrides any `#pragma pack()` setting for this structure. If zero (0) is selected, natural alignment is used (not the current `#pragma pack` value).

`__packed`
: Hybrid modifier: when applied to a structure definition, forces the packing to be 1-byte aligned. When applied to a pointer, forces all accesses through that pointer to assume an unaligned pointer. (This is also the case when a pointer to a `__packed` structure is used.)

`__pure`
: In function prototypes modifying the function name, this keyword indicates that the function has no side-effects and relies only on its input parameters. Currently, the Palm OS compiler ignores this keyword.

`__ror32(`*`x`*`, `*`y`*`)`
A built-in operator that returns the 32-bit unsigned integer *x* rotated right by *y* bits.

`__value_in_regs`
When this keyword is applied to a function prototype or declaration, states that the return value of the function, if it is a small structure (16 bytes or less), is passed in processor registers r0-r3. (Normally structure return values are passed by pointer in a hidden first argument.)

This calling convention keyword is potentially useful to interoperate with special routines.

Example:

```
struct div_result {int div, rem;};
struct div_result __value_in_regs do_div (int x, int y);
```

`__weak`
In declarations of external objects (functions or data), this modifier indicates that the object is not required and the linker should fix up references if the object is not available during linkage.

Constants

The value of any constant must be in the range of representable values for its type. The C language contains the following types of constants (also called *literals*):

- Integer (decimal, octal, or hexadecimal notation)
- Floating-point (`double`, `float`, `long double`, or hexadecimal notation)
- Character (one or more characters in apostrophes)
- String (sequence of characters enclosed in double quotes)
- Enumeration

Operators

Operators can be classified as:

- Postfix

- Prefix
- Normal
- Boolean
- Assignment
- C++ Compatibility

Postfix

Postfix operators are operators that are suffixed to an expression, such as, `operand++`.

Prefix

Prefix operators are operators that are prefixed to an expression, such as, `++operand` or `!operand`.

Normal

There are several normal operators that return the result defined for each:

`+`	addition
`-`	subtraction
`*`	multiplication
`/`	division
`%`	modulo
`&`	AND
`|`	OR
`^`	XOR
`>>`	shift right
`<<`	shift left

Boolean

The Boolean operators return either 1 (true) or 0 (false).

`&&`	logical AND
`||`	logical OR
`<`	less than

`>`	greater than
`<=`	less than equal
`>=`	greater than equal
`==`	equal
`!=`	not equal

Assignment

An assignment operator stores the value of the right expression into the left expression:

`=`	`a = b` assigns the value of `b` into `a`
`*=`	`a *= b` is equivalent to `a = a * b`
`/=`	`a /= b` is equivalent to `a = a / b`
`%=`	`a %= b` is equivalent to `a = a % b`
`+=`	`index += 2` is equivalent to `index = index + 2`
`-=`	`index -= 3` is equivalent to `index = index - 3`
`<<=`	`n1 <<= n2` is equivalent to `n1 = n1 << n2`
`>>=`	`n1 >>= n2` is equivalent to `n1 = n1 >> n2`
`&=`	`mask &= 2` is equivalent to `mask = mask & 2`
`^=`	`t1 ^= t2` is equivalent to `t1 = t1 ^ t2`
`|=`	`flag |= ON` is equivalent to `flag = flag | ON`

C++ Compatibility

There are three new compound operators in C++:

`.*`	Binds its second operand, which shall be of type "pointer to member of T" (where T is a completely defined class type) to its first operand, which shall be of class T.
`->*`	Binds its second operand, which shall be of type "pointer to member of T" (where T is a completely defined class type) to its first operand, which shall be of type "pointer to T" or "pointer to a class of which T is an unambiguous and accessible base class."

`::`	Allows a type, an object, a function, an enumerator, or a namespace declared in the global namespace to be referred to even if its identifier has been hidden.

Separators

Separators can include:

`( )`	parenthesis
`[ ]`	brackets
`{ }`	braces
`,`	comma
`;`	semi-colon
`:`	colon

Preprocessor Directives

Preprocessor directives instruct the preprocessor to act on the text of the program. Preprocessor directives begin with the # token followed by a preprocessor keyword. The # token must appear as the first character that is not white space on a line. The # is not part of the directive name and can be separated from the name with white space. Except for some `#pragma` directives, preprocessor directives can appear anywhere in a program.

#pragma

A *pragma directive* is an implementation-defined instruction to the compiler. This section describes the `#pragma` commands that the Palm OS compiler recognizes.

`#pragma once`

Indicates that a source file (usually a header) need not be included again. (Thus an `#include` of the same header has no effect.) If normal header guards are used, the compiler optimizes them into a `#pragma once`:

```
#pragma once // unnecessary

#ifndef MY_HEADER_GUARD

#define MY_HEADER_GUARD

// header contents ...

#endif /* MY_HEADER_GUARD */
```

`#pragma pack(`*`n`*`)`
: Sets current structure packing to *`n`*, where *`n`* is 1, 2, 4, 8, or 16.

`#pragma pack()`
: Resets current structure packing to natural alignment.

`#pragma pack (pop [,`*`name`*`] [,`*`n`*`])`
: If *`name`* is supplied, pops back to the position on the stack with that *`name`*, otherwise pops a single value off the stack. If *`n`* is supplied, sets the alignment to that value after popping.

`#pragma pack (push [,`*`name`*`] [,`*`n`*`])`
: Pushes the current structure packing onto a stack. If *`name`* (an identifier) is supplied, names the prior position on the stack. If *`n`* is supplied, sets the packing to that value, after pushing the original value.

`#pragma weak `*`name`*
: Same as declaring the global object with the external name of *`name`* with the `__WEAK` attribute.

Predefined Constants

This section describes the predefined constants provided by the Palm OS Protein C/C++ Compiler.

`__APGE__`
: Defined as 1.

`__APOGEE__`
: Defined as 1.

`__arm`
: Defined as 1.

`_BOOL`
: Defined in C++ mode when `bool` is a keyword.

`__cplusplus`
: Defined in C++ mode.

`c_plusplus`
: Defined in default C++ mode, but not in `strict` mode.

`__DATE__`
: Defined in all modes to the date of the compilation in the form “Mmm dd yyyy.”

`__EDG__`
: Always defined.

`__EDG_VERSION__`
: Defined to an integral value that represents the version number of the front end. For example, version 2.30 is represented as 230.

`__embedded_cplusplus`
: Defined as 1 in embedded C++ mode.

`__EXCEPTIONS`
: Defined in C++ mode when exception handling is enabled.

`_PACC_VER`
: `0xMmmrrbbb`, where (M=Major, m=minor, r=rev, b=build). For example, `0x1000000D`, for 1.0.0.13.

`__PALMSOURCE__`
: Defined as 1.

`__PSI__`
: Defined as 1.

`__RTTI`
: Defined in C++ mode when RTTI is enabled.

`__SIGNED_CHARS__`
: Defined when plain character is signed. (By default, the character type is unsigned.)

`__STDC__`
: Defined in ANSI C mode and in C++ mode. In C++ mode, the value may be redefined.

`__STDC_HOSTED__`
: Defined in C99 mode with the value zero (0).

`__STDC_VERSION__`
: Defined in ANSI C mode with the value `199901L`.

`__TIME__`
: Defined in all modes to the time of the compilation in the form “hh:mm:ss.”

`_WCHAR_T`
: Defined in C++ mode when `wchar_t` is a keyword.

Part II
C/C++ Compiler Library Reference

This part is organized in the following manner: general library and runtime function information appears first, followed by detailed header file information that documents the supported structures, runtime functions, and macros. Note that header file chapters, which are organized alphabetically, follow the "Runtime Library Functions" chapter, which overviews the supported runtime functions provided by the operating system and the *unsupported* runtime functions not implemented by Palm OS.

3 STLport/iostream

The Palm OS® Protein C/C++ Compiler Suite includes and supports the STLport implementation of the C++ standard template library.

Specific details regarding the implementation of the C++ STLport/iostream material is not currently documented in this manual; for documentation, visit http://www.stlport.org/doc/index.html. However, the following information may be useful:

- iostreams are implemented in terms of `stdio`; `cout` is connected to `stdout`, `cerr` is connected to `stderr`, and `cin` is connected to `stdin`.
- no locale functionality beyond the C locale is supported.
- all other pieces of STL functionality are believed to be supported.

For more information on the functionality provided by the C++ standard library, please consult documentation on the C++ language, such as *The C++ Programming Language*, Third Edition, by Bjarne Stroustrup, or the ANSI/ISO specification, available as ANSI/ISO/IEC document 14882:1998.

4

Palm OS-Specific Libraries

An integral part of the Palm OS® Protein C/C++ Compiler are the standard headers, startup code, and run-time libraries. The supplied run-time libraries serve several purposes:

- cpp — The cpp libraries implement objects common to any C++ standard library (e.g., the standard exception objects).
- eabi — The eabi libraries implement preliminary ARM EABI support on top of Palm OS. They implement the necessary EABI support routines, translating them into Palm OS specific routine calls.
- pacc — The pacc libraries implement objects and routines that are unique or particular to the Palm OS compiler and are not required or useful with any other tool chain.
- STLport — The C++ standard template library features thread safety, improved memory utilization, improved run-time efficiency, and new data structures, including hash tables.
- support — This is an implementation of the floating-point and integral support functions. The Palm OS compiler automatically links with this library, however, the `FloatMgr` library should also be linked.

The Palm OS Implementation of the Standard C Library (libc)

The Palm OS implementation of the standard C library is derived from the NetBSD ARM source base, with some modification due to the non-Unix nature of Palm OS:

- In the future, it may be possible to direct `stdout/stdin` operations through other I/O devices; no timeline for this has been stated.
- The C99 header <`complex.h`> is not supported in this version of `libc`. Applications using complex numbers should use STLport or another ANSI compliant C++ library.
- The C99 header <`fenv.h`> is not supported in this version of `libc`. MathLib does not raise floating exceptions and does not respond to varying rounding modes. Checking `errno` and the return value can handle exceptional cases.
- There is also no <`setjmp.h`> implementation. The <`ErrTryCatch.h`> header can provide much of the same functionality, but the standard C interface is not yet supported.
- In addition, the following POSIX header files are not documented in this reference because they are either fairly self-explanatory or do not contain any runtime library functions that are provided by the operating system.
  - `<climits.h>`
  - `<inttypes.h>`
  - `<limits.h>`
  - `<namespace.h>`
  - `<paths.h>`
  - `<signal.h>`
  - `<stdint.h>`
  - `<termios.h>`

5

Runtime Library Functions

Supported Functions

The following is an alphabetical list of runtime library functions, as defined in the POSIX headers for Palm OS® 6.1, which are explicitly provided by the operating system. For detailed information about any of these functions, see the individual header file chapters that follow, beginning with Chapter 6, "assert.h."

posix/ctype.h

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

posix/math.h

`abs()`	`expf()`	`logf()`
`acos()`	`expl()`	`logl()`
`acosf()`	`expm1()`	`modf()`
`acosh()`	`fabs()`	`modff()`
`acosl()`	`fabsf()`	`modfl()`
`asin()`	`fabsl()`	`nextafter()`
`asinf()`	`floor()`	`pow()`

`asinh()`	`floorf()`	`powf()`
`asinl()`	`floorl()`	`powl()`
`atan()`	`fmod()`	`remainder()`
`atan2()`	`fmodf()`	`rint()`
`atan2f()`	`fmodl()`	`scalbn()`
`atan2l()`	`frexp()`	`sin()`
`atanf()`	`frexpf()`	`sinf()`
`atanh()`	`frexpl()`	`sinh()`
`atanl()`	`hypot()`	`sinhf()`
`cbrt()`	`hypotf()`	`sinhl()`
`ceil()`	`hypotl()`	`sinl()`
`ceilf()`	`ilogb()`	`sqrt()`
`ceill()`	`ldexp()`	`sqrtf()`
`copysign()`	`ldexpf()`	`sqrtl()`
`cos()`	`ldexpl()`	`tan()`
`cosf()`	`log()`	`tanf()`
`cosh()`	`log10()`	`tanh()`
`coshf()`	`log10f()`	`tanhf()`
`coshl()`	`log10l()`	`tanhl()`
`cosl()`	`log1p()`	`tanl()`
`exp()`	`logb()`	

posix/stdio.h

`asprintf()`	`freopen()`	`rewind()`
`clearerr()`	`fscanf()`	`scanf()`
`fclose()`	`fseek()`	`setbuf()`

`fdopen()`
`feof()`
`ferror()`
`fflush()`
`fgetc()`
`fgetln()`
`fgetpos()`
`fgets()`
`fileno()`
`fopen()`
`fprintf()`
`fpurge()`
`fputc()`
`fputs()`
`fread()`
`fseeko()`
`fsetpos()`
`ftell()`
`ftello()`
`fwrite()`
`getc()`
`getchar()`
`gets()`
`getw()`
`perror()`
`printf()`
`putc()`
`putchar()`
`puts()`
`putw()`
`setbuffer()`
`setlinebuf()`
`setvbuf()`
`snprintf()`
`sprintf()`
`sscanf()`
`ungetc()`
`vasprintf()`
`vfprintf()`
`vprintf()`
`vscanf()`
`vsnprintf()`
`vsprintf()`
`vsscanf()`

posix/stdlib.h

`abs()`
`atof()`
`atoi()`
`atol()`
`atoll()`
`bsearch()`
`calloc()`
`div()`
`free()`
`inplace_realloc()`
`labs()`
`ldiv()`
`llabs()`
`malloc()`
`qsort()`
`qsort_r()`
`rand()`
`rand_r()`
`random()`
`realloc()`
`srand()`
`srandom()`
`strtod()`
`strtol()`
`strtoll()`
`strtoul()`
`strtoull()`

posix/string.h

`memchr()`	`strcspn()`	`strncpy()`
`memcmp()`	`strdup()`	`strpbrk()`
`memcpy()`	`strerror()`	`strrchr()`
`memmove()`	`strerror_r()`	`strsep()`
`memset()`	`strlcat()`	`strspn()`
`strcat()`	`strlcpy()`	`strstr()`
`strchr()`	`strlen()`	`strtok()`
`strcmp()`	`strncat()`	`strtok_r()`
`strcoll()`	`strncmp()`	`strxfrm()`
`strcpy()`		

posix/strings.h

`bcopy()`	`strcasecmp()`
`bzero()`	`strncasecmp()`

posix/time.h

`asctime()`	`difftime()`	`mktime()`
`asctime_r()`	`gmtime()`	`strftime()`
`clock()`	`gmtime_r()`	`time()`
`ctime()`	`localtime()`	`time()`
`ctime_r()`	`localtime_r()`	

posix/sys/ioctl.h

`ioctl()`

posix/sys/PalmMath.h

`lceilf()`	`lfloorf()`	`sincosf()`

posix/sys/time.h

`getcountrycode()`	`palm_seconds_to_time_t()`
`getgmtoffset()`	`settime()`
`gettimezone()`	`settimezone()`
`hastimezone()`	`system_real_time()`
`localtime_tz()`	`system_time()`
`mktime_tz()`	`time_t_to_palm_seconds()`

posix/sys/uio.h

`readv()`	`writev()`

Unsupported Functions

The following is an alphabetical list of runtime library functions, sorted by header file name, declared in the POSIX headers that are *not* implemented by the operating system.

posix/ctype.h

`isascii()` (this is handled via a `#define`)	`toascii()` (this is handled via a `#define`)

posix/inttypes.h

`strtoimax()`	`strtoumax()`

posix/locale.h

`setlocale()`

posix/math.h

`erf()`	`islessequal()`	`modf()`
`erfc()`	`islessgreater()`	`nan()`
`exp2()`	`isunordered()`	`nearbyint()`
`fdim()`	`lgamma()`	`nexttoward()`
`fma()`	`llrint()`	`remquo()`
`fmax()`	`llround()`	`round()`
`fmin()`	`log2()`	`scalbln()`
`isgreater()`	`lrint()`	`tgamma()`
`isgreaterequal()`	`lround()`	`trunc()`
`isless()`		

In addition, any of the above functions that have `float` overrides (suffixed with an "f") or `long double` overrides (suffixed with an "l") are also unsupported. For example, `exp2f()` and `exp2l()`.

posix/signal.h

`kill()`	`sigblock()`	`sigpending()`
`killpg()`	`sigdelset()`	`sigprocmask()`
`psignal()`	`sigemptyset()`	`sigreturn()`
`raise()`	`sigfillset()`	`sigsetmask()`
`sigaction()`	`siginterrupt()`	`sigstack()`
`sigaddset()`	`sigismember()`	`sigsuspend()`
`sigaltstack()`	`sigpause()`	`sigvec()`

posix/stdio.h

`ctermid()`	`getc_unlocked()`	`remove()`
`cuserid()`	`getchar_unlocked()`	`rename()`

`flockfile()`	`pclose()`	`tempnam()`
`ftrylockfile()`	`popen()`	`tmpfile()`
`funlockfile()`	`putc_unlocked()`	`tmpnam()`
`funopen()`	`putchar_unlocked()`	

posix/stdlib.h

`a64l()`	`drand48()`	`mktemp()`
`abort()`	`erand48()`	`mrand48()`
`alloca()`	`exit()`	`nrand48()`
`atexit()`	`getbsize()`	`putenv()`
`cfree()`	`getenv()`	`qdiv()`
`cgetcap()`	`getloadavg()`	`radixsort()`
`cgetclose()`	`heapsort()`	`realpath()`
`cgetent()`	`initstate()`	`seed48()`
`cgetfirst()`	`jrand48()`	`setenv()`
`cgetmatch()`	`l64a()`	`setkey()`
`cgetnext()`	`lcong48()`	`setstate()`
`cgetnum()`	`lldiv()`	`sradixsort()`
`cgetset()`	`lrand48()`	`srand48()`
`cgetstr()`	`mergesort()`	`ttyslot()`
`cgetustr()`	`mkdtemp()`	`unsetenv()`
`daemon()`	`mkstemp()`	`valloc()`
`devname()`		

posix/string.h

`memccpy()`

posix/strings.h

`bcmp()`	`index()`
`ffs()`	`rindex()`

posix/termios.h

`tcdrain()`	`tcflush()`	`tcsendbreak()`
`tcflow()`	`tcgetpgrp()`	`tcsetpgrp()`

posix/time.h

`clock_getres()`	`strptime()`	`timer_getoverrun()`
`clock_gettime()`	`time2posix()`	`timer_gettime()`
`clock_settime()`	`timelocal()`	`timer_settime()`
`nanosleep()`	`timeoff()`	`timezone()`
`offtime()`	`timer_create()`	`tzset()`
`posix2time()`	`timer_delete()`	`tzsetwall()`

posix/wchar.h

`fwide()`	`wcsncat()`	`wcstoul()`
`wcscat()`	`wcsncmp()`	`wcswidth()`
`wcschr()`	`wcsncpy()`	`wcwidth()`
`wcscmp()`	`wcspbrk()`	`wmemchr()`
`wcscpy()`	`wcsrchr()`	`wmemcmp()`
`wcscspn()`	`wcsspn()`	`wmemcpy()`
`wcslcat()`	`wcsstr()`	`wmemmove()`
`wcslcpy()`	`wcstod()`	`wmemset()`
`wcslen()`	`wcstol()`	

posix/machine/arm/param.h

`delay()`

posix/sys/bswap.h

`bswap16()`	`bswap32()`	`bswap64()`

posix/sys/socket.h

`socketpair()`

posix/sys/stat.h

`chflags()`	`lchflags()`	`mkfifo()`
`chmod()`	`lchmod()`	`mknod()`
`fchflags()`	`lstat()`	`stat()`
`fchmod()`	`mkdir()`	`umask()`
`fstat()`		

posix/sys/time.h

`adjtime()`	`itimerdecr()`	`ratecheck()`
`adjtime1()`	`itimerfix()`	`setitimer()`
`clock_settime1()`	`lutimes()`	`settimeofday()`
`futimes()`	`microtime()`	`settimeofday1()`
`getitimer()`	`ppsratecheck()`	`utimes()`
`gettimeofday()`		

posix/sys/uio.h

`preadv()`	`pwritev()`

6

assert.h

The `<assert.h>` header defines the `assert()` macro, which is used for debugging purposes. It also refers to another macro, `NDEBUG`, which is defined elsewhere.

Functions and Macros

assert Macro

Purpose Outputs a diagnostic message to standard errorand stops the program if a test fails.

Prototype `assert (`*`condition`*`)`

Parameters → *`condition`*

An expression to test; if the result of the expression is `false`, the diagnostic message is displayed and the program terminates. If the result is `true`, this macro has no effect.

Example In the following example, the program will terminate if the data buffer could not be allocated.

```
char *buffer = malloc(150);
assert(buffer);
```

7

ctype.h

The `<ctype.h>` header defines several functions useful for classifying and converting characters. All of the functions declared in this header are part of the C99 standard.

NOTE: None of the functions in `<ctype.h>` are internationally safe. They work only for 7-bit ASCII characters. Many of these functions have Palm OS specific equvalents that are internationally safe. These are listed in Table 7.1.

Table 7.1 Functions with internationally safe equivalents

Function	Palm OS specific equivalent
`isalnum()`	`TxtCharIsAlNum()`
`isalpha()`	`TxtCharIsAlpha()`
`iscntrl()`	`TxtCharIsCntrl()`
`isdigit()`	`TxtCharIsDigit()`
`isgraph()`	`TxtCharIsGraph()`
`islower()`	`TxtCharIsLower()`
`isprint()`	`TxtCharIsPrint()`
`ispunct()`	`TxtCharIsPunct()`
`isspace()`	`TxtCharIsSpace()`
`isupper()`	`TxtCharIsUpper()`
`isxdigit()`	`TxtCharIsHex()`

Table 7.1 Functions with internationally safe equivalents

Function	Palm OS specific equivalent
`tolower()`	`StrToLower()` and `TxtTransliterate()`
`toupper()`	`TxtTransliterate()`

For details on the internationally safe functions listed above, see the book *Exploring Palm OS: Text and Localization*.

8

errno.h

The `<errno.h>` header provides the global error code variable `errno`.

Global Variables

errno Variable

Purpose Global error code variable.

Declared In `posix/errno.h`

Prototype `extern int errno`

Comments The `errno` variable is used by many functions to return error values. The value of `errno` is defined only after a call to a function for which it is explicitly stated to be set and until it is changed by the next function call. The value of `errno` should only be examined when it is indicated to be valid by a function's return value. Programs should obtain the definition of `errno` by the inclusion of `<errno.h>`. It is unspecified whether `errno` is a macro or an identifier declared with external linkage.

The `errno` variable has a value of zero (0) at the beginning. If an error occurs, then this variable is given the value of the error number. In some cases, the behavior of the math library with regard to `errno` is implementation defined.

Nothing in the `<errno.h>` header is specific to Palm OS®. The specific numeric values associated with the error names are not portable and should be treated as opaque by applications.

9

ioctl.h

The `<ioctl.h>` header defines a function to manipulate the underlying device parameters of special files. It defines the `ioctl()` function, which is a standard Posix function.

10

iso646.h

The `<iso646.h>` header defines several constants that expand to the corresponding tokens, useful for programming in ISO 646 variant character sets.

Operators

Purpose Defines constants that expand to the corresponding tokens.

Declared In `posix/iso646.h`

Constants

`#define and &&`
The operator &&.

`#define and_eq &=`
The operator &=.

`#define bitand &`
The operator &.

`#define bitor |`
The operator |.

`#define compl ~`
The operator ~.

`#define not !`
The operator !.

`#define not_eq !=`
The operator !=.

`#define or ||`
The operator ||.

`#define or_eq |=`
The operator |=.

`#define xor ^`
The operator ^.

`#define xor_eq ^=`
The operator ^=.

11 locale.h

The `<locale.h>` header support in `libc` has not been integrated with Palm OS® and thus should not be used. The macros and functions defined in this header do not work as expected and should be avoided.

12

math.h

The <`math.h`> header defines several mathematical functions.

This header is new with Palm OS® Protein. It is a broad subset of section 7.12 of the C language standard ANSI/ISO/IEC 9899:1999.

MathLib is part of SystemLib. To use MathLib, simply include the <`math.h`> header in your source files.

Supported features

The Palm OS Protein C/C++ Compiler supports the use of infinity and NaN (not-a-number) values.

The following C99 macros are supported in <`math.h`>:

- `FLT_EVAL`
- `FP_ILOGBNAN`
- `FP_ILOGB0`
- `FP_INFINITE`
- `FP_NAN`
- `FP_NORMAL`
- `FP_SUBNORMAL`
- `FP_ZERO`
- `HUGE_VAL`
- `HUGE_VALF`
- `HUGE_VALL`
- `INFINITY`
- `MATH_ERREXCEPT`
- `math_errhandling`
- `MATH_ERRNO`
- `NAN`

Differences from the C99 specification

- All of <`math.h`> as specified in the C language standard ANSI/ISO/IEC 9899:1990 is provided as well as most of the extensions specified in 1999 standard. Parts of <`math.h`> that are not supported are listed under the line:

  ```
  #ifdef __USE_C99_EXTENSIONS__
  ```

  Functions in this section are preprocessed out by default and are not tested or supplied by PalmSource.
- Parallel sets of functions for `float` and `long double` arguments types are defined only for 1989 ANSI C functions.

Constraints

- Existing 68K applications must continue to supply the 68K MathLib if required by the application.
- There are cases in which the behavior of the math library with respect to the `errno` error reporting mechanism are implementation defined. For details on how the Palm OS Protein C/C++ Compiler handles errors in these cases, see "Behavior of errno" on page 53.
- The `float` and `long double` overloads as specified in section 26.5 of the ANSI C++ standard are *not* provided.
- The `float` and `long double` counterparts suffixed by "f" and "l" for the functions defined in section 7.12 of the 1989 ANSI C language standard are supported. A few of the `float` counterparts have Palm OS implementations, but most of these simply cast and return the `double` version.
- A handful of single precision counterparts are provided as a high performance alternative to their `double` equivalents. However, there are some additional deviations from the standard that were made to achieve high performance, including:
  - none of the single precision functions set the global variable `errno`.
  - `sqrtf()` flushes denormals to zero (0).
  - `ceilf(-0)` is 0 not –0 as specified in Annex F.9.6.1 of the ANSI standard.

  - `hypotf()` does not follow the spec for NaNs and infinities.
- The library, `libm.a`, is no longer supported and must be removed from existing projects.

Behavior of errno

There are two situations in which case an infinite result can occur as the result of an operation. The first is when a range error occurs, where the computation using finite arguments causes a result that lies outside the range of values that can be represented by the data type. The second case is when the result is infinite because the mathematics involved actually results in an infinite result (such as when an input argument is infinite). In this case, there is no range error, but the result is still infinite.

Some functions only have two ways to result in an infinite value—either the result is infinite, or the result cannot be represented due to its size. In this case, `errno` can be useful, because it will indicate no error if the result is in fact infinity, or `ERANGE` if a range error occurred.

On the other hand, if the function has multiple ways in which infinity can be the result, `errno` cannot be used as a method for determining whether or not the value is legitimate or not.

For this reason, the Palm OS Protein C/C++ Compiler handles `errno` in these cases as follows:

`errno` is set only when it can be used to definitively distinguish between multiple ways of arriving at the same return value. One should not expect `errno` to be set in cases where it will not help determine the reason why the result was achieved. See Table 12.1.

Table 12.1 errno handling for specific cases

Function Name	Sets errno	Explanation
`hypot()`	`ERANGE`	Overflow occurs when `hypot(finite but large, finite)` is called.
`scalbn()` / `ldexp()`	no	Too many ways for infinity to result

Table 12.1 errno handling for specific cases

Function Name	Sets errno	Explanation
`nextafter()`	`ERANGE`	`nextafter(DBL_MAX, INFINITY)`
`ilogb()`	no	Both 0 and denormals have the exponent 0.
`logb()` / `log10()` / log()	`EDOM`	`log`*x*`(0)` isn't equal to `log`*x*`(denormal)`.
`pow()`	`ERANGE`	`pow(finite > 0, finite large > 0)` indicates that overflow occurred. Required by section 7.12.1 of the standard.
`pow(INFINITY, INFINITY)`	no	Overflow does not occur, since the original values were not finite.
`any function(NaN)`	no	Ideally, the function should propagate the same NaN that was passed in, with no additional side effects
`cos(x=±inf)` / `sin(x=±inf)` / *tan(x=–inf)*	`EDOM`	Distinguishes between x being infinite or NaN. Also, the limit as x approaches infinity does not exist.

13

PalmMath.h

The `<PalmMath.h>` header defines Palm OS specific mathematical functions not specified in the ANSI/ISO standard.

Constants

Math Constants

Purpose These constants are intended to be used as 32-bit floats. These constants should not be used as double precision arguments. However, a new double precision version of each of these may be created by removing the “f” suffix from the end of each decimal string.

Declared In `posix/sys/palmmath.h`

Constants

`#define M_E 2.7182818284590452354f`
: Approximates the mathematical constant *e*.

`#define M_LOG2E 1.4426950408889634074f`
: Approximates the mathematical constant $\log_2(e)$.

`#define M_LOG10E 0.43429448190325182765f`
: Approximates the mathematical constant $\log_{10}(e)$.

`#define M_LN2 0.69314718055994530942f`
: Approximates the mathematical constant $\log_e(2)$.

`#define M_LN10 2.30258509299404568402f`
: Approximates the mathematical constant $\log_e(10)$.

`#define M_PI 3.14159265358979323846f`
: Single precision approximation to π.

`#define M_PI_2 1.57079632679489661923f`
: Single precision approximation to π/2.

`#define M_1_PI 0.31830988618379067154f`
: Single precision approximation to 1/π.

`#define M_PI_4 0.78539816339744830962f`
Single precision approximation to π/4.

`#define M_2_PI 0.63661977236758134308f`
Single precision approximation to 2/π.

`#define M_2_SQRTPI 1.12837916709551257390f`
Single precision approximation to 2/√π.

`#define M_SQRT2 1.41421356237309504880f`
Approximates the mathematical constant √2.

`#define M_SQRT1_2 0.70710678118654752440f`
Approximates the mathematical constant 1/√2.

`#define PI M_PI`
Single precision approximation to π.

`#define PI2 M_PI_2`
Single precision approximation to π/2.

`#define M_PI_3 1.04719755119659774615f`
Single precision approximation to π/3.

`#define M_3_PI_4 2.35619449019234492884f`
Single precision approximation to 3*π/4.

`#define M_5_PI_4 3.92699081698724154807f`
Single precision approximation to 5*π/4.

`#define M_3_PI_2 4.71238898038468985769f`
Single precision approximation to 3*π/2.

`#define M_7_PI_4 5.49778714378213816730f`
Single precision approximation to 7*π/4.

Functions and Macros

lceilf Function

Purpose Computes the nearest 32-bit signed integer not less than *x*.

Declared In `posix/sys/palmmath.h`

Prototype `int32_t lceilf (float x)`

Parameters → *x*

Value of type `float` to be evaluated.

Returns Returns the nearest 32-bit signed integer not less than *x*. In cases where *x* is out of the range of representable integers, +/-`INT_MAX` is returned.

Comments Exceptions are never raised.

Compatibility This function is *not* in the C99 specification.

This function *is* a Palm OS extension (not present in C99 or Unix).

See Also `lfloorf()`

lfloorf Function

Purpose Computes the nearest 32-bit signed integer not greater than *x*.

Declared In `posix/sys/palmmath.h`

Prototype `int32_t lfloorf (float x)`

Parameters → *x*

Value of type `float` to be evaluated.

Returns Returns the nearest 32-bit signed integer not greater than *x*. In cases where *x* is out of the range of representable integers, +/-`INT_MAX` is returned.

Comments Exceptions are never raised.

Compatibility This function is *not* in the C99 specification.

This function *is* a Palm OS extension (not present in C99 or Unix).

See Also `lceilf()`

max_c Macro

Purpose	Returns the larger of two values.
Declared In	`posix/sys/PalmMath.h`
Prototype	`#define max_c(real floating a, real floating b)`
Parameters	→ *a* The first value to compare. → *b* The second value to compare.
Returns	Returns the greater of the two input values.
Compatibility	This macro can be used in C code, unlike the standard `max()` macro, which uses C++ templates.
See Also	`min_c()`

min_c Macro

Purpose	Returns the lesser of two values.
Declared In	`posix/sys/PalmMath.h`
Prototype	`#define min_c(real floating a, real floating b)`
Parameters	→ *a* The first value to compare. → *b* The second value to compare.
Returns	Returns the lesser of the two input values.
Compatibility	This macro can be used in C code, unlike the standard `min()` macro, which uses C++ templates.
See Also	`max_c()`

sincosf Function

Purpose	Computes an approximation to the sine (*`sin_val`*) and cosine (*`cos_val`*) of any angle in a single call.

Declared In `posix/sys/palmmath.h`

Prototype

```
void sincosf (float angle, float *cos_val,
    float *sin_val)
```

Parameters

→ *`angle`*
Must be specified in radians.

→ *`cos_val`*
Cosine value.

→ *`sin_val`*
Sine value.

Returns Returns the approximation to the sine (*`sin_val`*) and cosine (*`cos_val`*) of the specified angle.

Compatibility This function is *not* in the C99 specification.

This function *is* a Palm OS extension (not present in C99 or Unix).

14 stdarg.h

The `<stdarg.h>` header defines several macros useful in the creation of functions that accept a variable number of arguments.

The macros defined in this header are all part of the C99 standard.

15

stddef.h

The `<stddef.h>` header defines the commonly used `offsetof()` macro, which is part of the C99 standard.

16

stdio.h

The `<stdio.h>` header defines functions for performing input and output.

The current expected behavior of the standard I/O library is to direct `stdout` and `stderr` output to a debugger via `DbgMessage()`, and to read bytes from `stdin` via the debugger using `DbgGetChar()`. Attempting to close one of the standard files [`stdin/stdout/stderr`] is not currently supported.

The functions in `stdio.h` are all standard `libc` functions, and are internationally aware, except that when printing floating-point numbers, they do not use a local-sensitive decimal character.

17

stdlib.h

The `<stdlib.h>` header defines several general operation functions and macros. Most of the functions defined in this header are standard libc functions; only `inplace_realloc()` is specific to Palm OS®.

NOTE: Functions that convert from strings to numbers are not multi-byte character aware, and do not take into account locale-sensitive settings such as the character used for decimals. Use the Palm OS specific equivalent functions if you need to be internationally safe.

Functions and Macros

inplace_realloc Function

Purpose Attempts to resize the memory block without moving it.

Declared In `posix/stdlib.h`

Prototype `void *inplace_realloc (void *`*`ptr`*`, size_t `*`size`*`)`

Parameters → *`ptr`*
The previously allocated memory.

→ *`size`*
The size, in bytes, to change to.

Returns Returns a pointer, possibly identical to *`ptr`*, to the allocated memory upon successful completion. Otherwise, a `NULL` pointer is returned, in which case the memory referenced by *`ptr`* is still available and intact.

Compatibility This function is a Palm OS extension (not present in C99 or Unix).

18

string.h

The `<string.h>` header defines several functions useful for manipulating strings (character arrays) and memory buffers.

Several standard string and memory manipulation functions have Palm OS® specific equivalents; the equivalents are provided for backward compatibility, but the C99 versions are preferred for future development. See *Exploring Palm OS: Memory, Databases, and Files* for details on the Palm OS specific functions. Table 18.1 lists the C99 functions and their Palm OS equivalents.

Table 18.1 C99 functions and their Palm OS specific equivalents

C99 Function	Palm OS Equivalent
`memmove()`	`MemMove()`
`memset()`	`MemSet()`
`strcat()`	`StrCat()`
`strcmp()`	`StrCompare()`
`strcoll()`	`StrCompare()`
`strcpy()`	`StrCopy()`
`strerror()`	`SysErrString()`
`strerror_r()`	`SysErrString()`
`strlcat()`	`StrLCat()`
`strlcpy()`	`StrLCopy()`
`strlen()`	`StrLen()`

Table 18.1 C99 functions and their Palm OS specific equivalents

C99 Function	Palm OS Equivalent
`strncat()`	`StrNCat()` **NOTE:** `StrNCat()` has a different meaning for its parameters, so a careful code review is necessary when shifting between `strncat()` and `StrNCat()`.
`strncmp()`	`StrNCompare()`
`strncpy()`	`StrNCopy()`
`strstr()`	`StrStr()`

For details on `SysErrString()`, see *Exploring Palm OS: System Management*.

The functions listed in Table 18.2 are not internationally safe.

Table 18.2 Functions that are not internationally safe

Function Name	Comments
`strchr()`	Not multi-byte aware.
`strcmp()`	Not multi-byte aware, and not locale sensitive.
`strcspn()`	Not multi-byte aware.
`strlcat()`	Not multi-byte aware.
`strlcpy()`	Not multi-byte aware.
`strncat()`	Not multi-byte aware; truncation can occur in the middle of a multi-byte character.
`strncmp()`	Not multi-byte aware or local sensitive.
`strncpy()`	Not multi-byte aware.

Table 18.2 Functions that are not internationally safe

Function Name	Comments
`strpbrk()`	Not multi-byte aware.
`strrchr()`	Not multi-byte aware.
`strsep()`	Not multi-byte aware.
`strspn()`	Not multi-byte aware.
`strstr()`	Not multi-byte aware.
`strtok()`	Not multi-byte aware.
`strtok_r()`	Not multi-byte aware.
`strxfrm()`	Not yet implemented.

19 strings.h

The `<strings.h>` header defines several functions useful for manipulating strings; these functions adhere to the Posix standard.

`strcasecmp()` and `strncasecmp()` are not internationally safe to use; they are neither multi-byte aware nor locale sensitive. These functions are equivalent to `StrCaselessCompare()` and `StrNCaselessCompare()`, respectively. See the book *Exploring Palm OS: Text and Localization* for details.

20
time.h

The `<time.h>` header defines several functions useful for reading and converting the current time and date.

The functions listed in Table 20.1 are not internationally safe to use.

Table 20.1 Functions that are not internationally safe

Function	Comments
`asctime()`	Not multi-byte aware, not locale sensitive, and returns unlocalized text.
`asctime_r()`	Not multi-byte aware, not locale sensitive, and returns unlocalized text.
`ctime()`	Not multi-byte aware, not locale sensitive, and returns unlocalized text.
`ctime_r()`	Not multi-byte aware, not locale sensitive, and returns unlocalized text.
`strftime()`	Not multi-byte aware, not locale sensitive, and returns unlocalized text.

The functions defined in this header are all standard `libc` functions.

21

time.h

The `<time.h>` header defines several Palm OS® specific functions useful for reading and converting the current time and date.

Constants

TZNAME_MAX

Purpose Defines the maximum length of a time zone identifier string.

Declared In `posix/sys/time.h`

Constants `#define TZNAME_MAX 32`

Functions and Macros

getcountrycode Function

Purpose Gets the two-byte country code for the specified time zone.

Declared In `posix/sys/time.h`

Prototype
```
status_t getcountrycode (const char *tzname,
   char *buf, size_t bufsize)
```

Parameters

→ *tzname*
 The time zone.

→ *buf*
 The buffer.

→ *bufsize*
 The size of the buffer.

Returns Returns `P_OK` upon successful completion; otherwise it returns `P_ERROR`.

Compatibility This function is a Palm OS extension (not present in C99 or Unix).

getgmtoffset Function

Purpose Gets the difference in seconds between Greenwich Mean Time (GMT) and local standard time.

Declared In `posix/sys/time.h`

Prototype `int32_t getgmtoffset (const char *tznanme)`

Parameters → *`tzname`*
The time zone.

Returns Returns the current GMT offset, which takes into account daylight saving time. This difference is positive for time zones West of Greenwich and negative for zones East of Greenwich.

Compatibility This function *`is`* a Palm OS extension (not present in C99 or Unix).

gettimezone Function

Purpose Copies the current system time zone name into *`buf`*.

Declared In `posix/sys/time.h`

Prototype `ssize_t gettimezone (char *buf, size_t bufsize)`

Parameters → *`buf`*
The buffer.

→ *`bufsize`*
The size of the buffer.

Returns Returns the number of bytes copied into *`buf`* upon successful completion; otherwise it returns `P_ERROR.`

Compatibility This function is a Palm OS extension (not present in C99 or Unix).

See Also `hastimezone()`, `settimezone()`

hastimezone Function

Purpose Determines if the system has the specified timezone. That is, if a timezone database is installed for the specified timezone.

Declared In `posix/sys/time.h`

Prototype `int hastimezone (const char *tzname)`

Parameters → *`tzname`*
The time zone.

Returns Returns `P_OK` upon successful completion; otherwise it returns `P_ERROR`.

Compatibility This function is a Palm OS extension (not present in C99 or Unix).

See Also `gettimezone()`, `settimezone()`

localtime_tz Function

Purpose Converts the specified UTC time in the time zone to a broken-down time.

Declared In `posix/sys/time.h`

Prototype
```
void localtime_tz (const time_t *timer,
   const char *tzname, struct tm *result)
```

Parameters → *`timer`*
The calendar time.

→ *`tzname`*
The time zone.

← *`result`*
A `tm` structure.

Compatibility This function is a Palm OS extension (not present in C99 or Unix).

mktime_tz Function

Purpose Converts a specified broken-down time in the time zone to UTC time. If the `tm_isdst` member of the `tm` struct is negative, this function tries to determine if the specified time zone is currently in daylight saving time.

Declared In `posix/sys/time.h`

Prototype
```
time_t mktime_tz (struct tm *tm,
   const char *tzname)
```

Parameters → *`tm`*
A `tm` structure.

→ *`tzname`*
The time zone.

Returns Returns the UTC time.

Compatibility This function is a Palm OS extension (not present in C99 or Unix).

palm_seconds_to_time_t Function

Purpose Takes as input the number of seconds since 1/1/1904 (old Palm epoch) and returns the number of seconds since 1/1/1970 (Unix epoch).

Declared In `posix/sys/time.h`

Prototype `time_t palm_seconds_to_time_t (uint32_t` *`seconds`*`)`

Parameters → *`seconds`*
The number of seconds.

Returns Returns the number of seconds since 1/1/1970 (Unix epoch).

Compatibility This function is a Palm OS extension (not present in C99 or Unix).

See Also `time_t_to_palm_seconds()`

settime Function

Purpose Sets the system time to the specified time.

Declared In `posix/sys/time.h`

Prototype `status_t settime (time_t` *`time`*`)`

Parameters → *`time`*
The system time.

Returns Returns `P_OK` upon successful completion.

Compatibility This function is a Palm OS extension (not present in C99 or Unix).

settimezone Function

Purpose Sets the system's time zone.

Declared In `posix/sys/time.h`

Prototype `status_t settimezone (const char *tzname)`

Parameters → *tzname*
The time zone.

Returns Returns `P_OK` upon successful completion; otherwise it returns `P_ERROR`.

Compatibility This function is a Palm OS extension (not present in C99 or Unix).

See Also `gettimezone()`, `hastimezone()`

system_real_time Function

Purpose Gets the value of the real time clock in nanoseconds.

Declared In `posix/sys/time.h`

Prototype `nsecs_t system_real_time (void)`

Returns Returns the value of the real time clock in nanoseconds.

Compatibility This function is a Palm OS extension (not present in C99 or Unix).

system_time Function

Purpose Gets the value of the run time clock in nanoseconds.

Declared In `posix/sys/time.h`

Prototype `nsecs_t system_time (void)`

Returns Returns the value of the run time clock in nanoseconds.

Compatibility This function is a Palm OS extension (not present in C99 or Unix).

time_t_to_palm_seconds Function

Purpose Takes as input the number of seconds since 1/1/1970 (Unix epoch) and returns the number of seconds since 1/1/1904 (old Palm epoch).

Declared In	`posix/sys/time.h`
Prototype	`uint32_t time_t_to_palm_seconds (time_t` *`seconds`*`)`
Parameters	→ *`seconds`* The number of seconds.
Returns	Returns the number of seconds since 1/1/1904 (old Palm epoch).
Compatibility	This function is a Palm OS extension (not present in C99 or Unix).
See Also	`palm_seconds_to_time_t()`

22

uio.h

The `<uio.h>` header defines two functions useful for vector I/O operations, as well as the `iovec` structure they require.

Structures and Types

iovec Struct

Purpose Defines an I/O vector; that is, a buffer address and size.

Declared In `posix/sys/uio.h`

Prototype

```
struct iovec {
   void *iov_base;
   size_t iov_len;
}
```

Fields

`iov_base`
The base address of a memory region for input or output.

`iov_len`
The size of the memory pointed to by *`iov_base`*.

Functions and Macros

readv Function

Purpose Performs the same action as `read()`, but scatters the input data into the *`iovcnt`* buffers specified by the members of the *`iov`* array: `iov[0]`, `iov[1]`, ..., `iov[iovcnt-1]`.

Declared In `posix/sys/uio.h`

Prototype

```
ssize_t readv (int d, const struct iovec *iov,
    size_t iovcnt)
```

Parameters → *`d`*
The position to start reading from.

→ *`iov`*
The array.

→ *`iovcnt`*
The buffer.

Returns Returns the number of bytes actually read and placed in the buffer. Zero (0) is returned if end-of-file is read. Otherwise, -1 is returned and the global variable `errno` is set to indicate the error.

See Also `read()`

writev Function

Purpose Performs the same action as `write()`, but gathers the output data from the *`iovcnt`* buffers specified by the members of the *`iov`* array: `iov[0]`, `iov[1]`, ..., `iov[iovcnt-1]`.

Declared In `posix/sys/uio.h`

Prototype
```
ssize_t writev (int d, const struct iovec *iov,
   size_t iovcnt)
```

Parameters → *`d`*
The position to start gathering from.

→ *`iov`*
The array.

→ *`iovcnt`*
The buffer.

Returns Returns the number of bytes actually written. Otherwise, -1 is returned and the global variable `errno` is set to indicate the error.

See Also `write()`

23

wchar.h

The `<wchar.h>` header is included for compliance purposes only.

None of the C wide-char (`wchar_t`) functionality is supported in Palm OS. (In fact, the `wchar_t` type is not even used by Palm OS since it can vary in size from 8-bits to 32-bits depending on the compiler.) For safe manipulation of text regardless of the device's character encoding, use the Palm OS String and Text Managers; see *Exploring Palm OS: Text and Localization*.

Index