The Cygnus C Support Library

Full Configuration

`libc` 1.4
May 1993

Libc

Steve Chamberlain
Roland Pesch
Cygnus Support

sac@cygnus.com, doc@cygnus.com *The Cygnus C Support Library*

'`libc`' includes software developed by the University of California, Berkeley and its contributors.

'`libc`' includes software developed by Martin Jackson, Graham Haley and Steve Chamberlain of Tadpole Technology and released to Cygnus.

'`libc`' uses floating point converstion software developed at AT&T, which includes this copyright information:

> The author of this software is David M. Gay.
>
> Copyright (c) 1991 by AT&T.
>
> Permission to use, copy, modify, and distribute this software for any purpose without fee is hereby granted, provided that this entire notice is included in all copies of any software which is or includes a copy or modification of this software and in all copies of the supporting documentation for such software.
>
> THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED WARRANTY. IN PARTICULAR, NEITHER THE AUTHOR NOR AT&T MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.

Table of Contents

1 System Calls

The C subroutine library depends on a handful of subroutine calls for operating system services. If you use the C library on a system that complies with the POSIX.1 standard (also known as IEEE 1003.1), most of these subroutines are supplied with your operating system.

If some of these subroutines are not provided with your system—in the extreme case, if you are developing software for a "bare board" system, without an OS—you will at least need to provide do-nothing stubs (or subroutines with minimal functionality) to allow your programs to link with the subroutines in `libc.a`.

1.1 Definitions for OS interface

This is the complete set of system definitions (primarily subroutines) required; the examples shown implement the minimal functionality required to allow `libc` to link, and fail gracefully where OS services are not available.

Graceful failure is permitted by returning an error code. A minor complication arises here: the C library must be compatible with development environments that supply fully functional versions of these subroutines. Such environments usually return error codes in a global `errno`. However, the Cygnus C library provides a *macro* definition for `errno` in the header file '`errno.h`', as part of its support for reentrant routines (see ⟨undefined⟩ "Reentrancy," page ⟨undefined⟩).

The bridge between these two interpretations of `errno` is straightforward: the C library routines with OS interface calls capture the `errno` values returned globally, and record them in the appropriate field of the reentrancy structure (so that you can query them using the `errno` macro from '`errno.h`').

This mechanism becomes visible when you write stub routines for OS interfaces. You must include '`errno.h`', then disable the macro, like this:

```
#include <errno.h>
#undef errno
extern int errno;
```

The examples in this chapter include this treatment of `errno`.

`_exit` Exit a program without cleaning up files. If your system doesn't provide this, it is best to avoid linking with subroutines that require it (`exit`, `system`).

`close` Close a file. Minimal implementation:

```
int close(int file){
    return -1;
```

```
}
```

`environ` A pointer to a list of environment variables and their values. For a minimal environment, this empty list is adequate:

```
char *__env[1] = { 0 };
char **environ = __env;
```

`execve` Transfer control to a new process. Minimal implementation (for a system without processes):

```
#include <errno.h>
#undef errno
extern int errno;
int execve(char *name, char **argv, char **env){
  errno=ENOMEM;
  return -1;
}
```

`fork` Create a new process. Minimal implementation (for a system without processes):

```
#include <errno.h>
#undef errno
extern int errno;
int fork() {
  errno=EAGAIN;
  return -1;
}
```

`fstat` Status of an open file. For consistency with other minimal implementations in these examples, all files are regarded as character special devices. The 'sys/stat.h' header file required is distributed in the 'include' subdirectory for this C library.

```
#include <sys/stat.h>
int fstat(int file, struct stat *st) {
  st->st_mode = S_IFCHR;
  return 0;
}
```

`getpid` Process-ID; this is sometimes used to generate strings unlikely to conflict with other processes. Minimal implementation, for a system without processes:

```
int getpid() {
  return 1;
}
```

`isatty` Query whether output stream is a terminal. For consistency with the other minimal implementations, which only support output to `stdout`, this minimal implementation is suggested:

```
int isatty(int file){
   return 1;
}
```

`kill` Send a signal. Minimal implementation:

```
#include <errno.h>
#undef errno
extern int errno;
int kill(int pid, int sig){
  errno=EINVAL;
  return(-1);
}
```

`link` Establish a new name for an existing file. Minimal implementation:

```
#include <errno.h>
#undef errno
extern int errno;
int link(char *old, char *new){
  errno=EMLINK;
  return -1;
}
```

`lseek` Set position in a file. Minimal implementation:

```
int lseek(int file, int ptr, int dir){
    return 0;
}
```

`read` Read from a file. Minimal implementation:

```
int read(int file, char *ptr, int len){
    return 0;
}
```

`sbrk` Increase program data space. As `malloc` and related functions depend on this, it is useful to have a working implementation. The following suffices for a standalone system; it exploits the symbol `end` automatically defined by the GNU linker.

```
caddr_t sbrk(int incr){
  extern char end; /* Defined by the linker */
  static char *heap_end;
  char *prev_heap_end;

  if (heap_end == 0) {
    heap_end = &end;
  }
  prev_heap_end = heap_end;
  heap_end += incr;
  return (caddr_t) prev_heap_end;
}
```

`stat` Status of a file (by name). Minimal implementation:

```
int stat(char *file, struct stat *st) {
  st->st_mode = S_IFCHR;
  return 0;
}
```

`times` Timing information for current process. Minimal implementation:

```
int times(struct tms *buf){
  return -1;
}
```

`unlink` Remove a file's directory entry. Minimal implementation:

```
#include <errno.h>
#undef errno
extern int errno;
int unlink(char *name){
  errno=ENOENT;
  return -1;
}
```

`wait` Wait for a child process. Minimal implementation:

```
#include <errno.h>
#undef errno
extern int errno;
int wait(int *status) {
  errno=ECHILD;
  return -1;
}
```

`write` Write a character to a file. '`libc`' subroutines will use this system routine for output to all files, *including* `stdout`—so if you need to generate any output, for example to a serial port for debugging, you should make your minimal `write` capable of doing this. The following minimal implementation

is an incomplete example; it relies on a `writechar` subroutine (not shown; typically, you must write this in assembler from examples provided by your hardware manufacturer) to actually perform the output.

```
int write(int file, char *ptr, int len){
    int todo;

    for (todo = 0; todo < len; todo++) {
        writechar(*ptr++);
    }
    return len;
}
```

1.2 Reentrant covers for OS subroutines

Since the system subroutines are used by other library routines that require reentrancy, 'libc.a' provides cover routines (for example, the reentrant version of fork is _fork_r). These cover routines are consistent with the other reentrant subroutines in this library, and achieve reentrancy by using a reserved global data block (see ⟨undefined⟩ "Reentrancy," page ⟨undefined⟩).

_open_r A reentrant version of open. It takes a pointer to the global data block, which holds errno.

```
int _open_r(void *reent,
    const char *file, int flags, int mode);
```

_close_r A reentrant version of close. It takes a pointer to the global data block, which holds errno.

```
int _close_r(void *reent, int fd);
```

_lseek_r A reentrant version of lseek. It takes a pointer to the global data block, which holds errno.

```
off_t _lseek_r(void *reent,
    int fd, off_t pos, int whence);
```

_read_r A reentrant version of read. It takes a pointer to the global data block, which holds errno.

```
long _read_r(void *reent,
    int fd, void *buf, size_t cnt);
```

_write_r A reentrant version of write. It takes a pointer to the global data block, which holds errno.

```
long _write_r(void *reent,
    int fd, const void *buf, size_t cnt);
```

_fork_r A reentrant version of fork. It takes a pointer to the global data block, which holds errno.

```
int _fork_r(void *reent);
```

_wait_r A reentrant version of wait. It takes a pointer to the global data block, which holds errno.

```
int _wait_r(void *reent, int *status);
```

_stat_r A reentrant version of stat. It takes a pointer to the global data block, which holds errno.

```
int _stat_r(void *reent,
    const char *file, struct stat *pstat);
```

_fstat_r A reentrant version of fstat. It takes a pointer to the global data block, which holds errno.

```
int _fstat_r(void *reent,
    int fd, struct stat *pstat);
```

`_link_r` A reentrant version of `link`. It takes a pointer to the global data block, which holds `errno`.

```
int _link_r(void *reent,
    const char *old, const char *new);
```

`_unlink_r`
A reentrant version of `unlink`. It takes a pointer to the global data block, which holds `errno`.

```
int _unlink_r(void *reent, const char *file);
```

`_sbrk_r` A reentrant version of `sbrk`. It takes a pointer to the global data block, which holds `errno`.

```
char *_sbrk_r(void *reent, size_t incr);
```

2 Variable Argument Lists

The `printf` family of functions is defined to accept a variable number of arguments, rather than a fixed argument list. You can define your own functions with a variable argument list, by using macro definitions from either '`stdarg.h`' (for compatibility with ANSI C) or from '`varargs.h`' (for compatibility with a popular convention prior to ANSI C).

2.1 ANSI-standard macros, '`stdarg.h`'

In ANSI C, a function has a variable number of arguments when its parameter list ends in an ellipsis (`...`). The parameter list must also include at least one explicitly named argument; that argument is used to initialize the variable list data structure.

ANSI C defines three macros (`va_start`, `va_arg`, and `va_end`) to operate on variable argument lists. '`stdarg.h`' also defines a special type to represent variable argument lists: this type is called `va_list`.

2.1.1 Initialize variable argument list

Synopsis

```
#include <stdarg.h>
void va_start(va_list ap, rightmost);
```

Description

Use `va_start` to initialize the variable argument list *ap*, so that `va_arg` can extract values from it. *`rightmost`* is the name of the last explicit argument in the parameter list (the argument immediately preceding the ellipsis '`...`' that flags variable arguments in an ANSI C function header). You can only use `va_start` in a function declared using this ellipsis notation (not, for example, in one of its subfunctions).

Returns

`va_start` does not return a result.

Portability

ANSI C requires `va_start`.

2.1.2 Extract a value from argument list

Synopsis

```
#include <stdarg.h>
type va_arg(va_list ap, type);
```

Description

`va_arg` returns the next unprocessed value from a variable argument list *`ap`* (which you must previously create with *`va_start`*). Specify the type for the value as the second parameter to the macro, *`type`*.

You may pass a `va_list` object *`ap`* to a subfunction, and use `va_arg` from the subfunction rather than from the function actually declared with an ellipsis in the header; however, in that case you may *only* use `va_arg` from the subfunction. ANSI C does not permit extracting successive values from a single variable-argument list from different levels of the calling stack.

There is no mechanism for testing whether there is actually a next argument available; you might instead pass an argument count (or some other data that implies an argument count) as one of the fixed arguments in your function call.

Returns

`va_arg` returns the next argument, an object of type *`type`*.

Portability

ANSI C requires `va_arg`.

2.1.3 Abandon a variable argument list

Synopsis

```
#include <stdarg.h>
void va_end(va_list ap);
```

Description

Use va_end to declare that your program will not use the variable argument list *ap* any further.

Returns

va_end does not return a result.

Portability

ANSI C requires va_end.

2.2 Traditional macros, 'varargs.h'

If your C compiler predates ANSI C, you may still be able to use variable argument lists using the macros from the 'varargs.h' header file. These macros resemble their ANSI counterparts, but have important differences in usage. In particular, since traditional C has no declaration mechanism for variable argument lists, two additional macros are provided simply for the purpose of defining functions with variable argument lists.

As with 'stdarg.h', the type va_list is used to hold a data structure representing a variable argument list.

2.2.1 Declare variable arguments

Synopsis

```
#include <varargs.h>
function(va_alist)
va_dcl
```

Description

To use the 'varargs.h' version of variable argument lists, you must declare your function with a call to the macro `va_alist` as its argument list, and use `va_dcl` as the declaration. *Do not use a semicolon after* `va_dcl`.

Returns

These macros cannot be used in a context where a return is syntactically possible.

Portability

`va_alist` and *`va_dcl`* were the most widespread method of declaring variable argument lists prior to ANSI C.

2.2.2 Initialize variable argument list

Synopsis

```
#include <varargs.h>
va_list ap;
va_start(ap);
```

Description

With the 'varargs.h' macros, use `va_start` to initialize a data structure *ap* to permit manipulating a variable argument list. *ap* must have the type *va_alist*.

Returns

`va_start` does not return a result.

Portability

`va_start` is also defined as a macro in ANSI C, but the definitions are incompatible; the ANSI version has another parameter besides *ap*.

2.2.3 Extract a value from argument list

Synopsis

```
#include <varargs.h>
type va_arg(va_list ap, type);
```

Description

`va_arg` returns the next unprocessed value from a variable argument list *`ap`* (which you must previously create with *`va_start`*). Specify the type for the value as the second parameter to the macro, *`type`*.

Returns

`va_arg` returns the next argument, an object of type *`type`*.

Portability

The `va_arg` defined in '`varargs.h`' has the same syntax and usage as the ANSI C version from '`stdarg.h`'.

2.2.4 Abandon a variable argument list

Synopsis

```
#include <varargs.h>
va_end(va_list ap);
```

Description

Use va_end to declare that your program will not use the variable argument list *ap* any further.

Returns

va_end does not return a result.

Portability

The va_end defined in ‘varargs.h’ has the same syntax and usage as the ANSI C version from ‘stdarg.h’.

Index

The body of this manual is set in
pncr at 10.95pt,
with headings in **pncb at 10.95pt**
and examples in pcrr.
pncri at 10.95pt and
pcrro
are used for emphasis.