

1 / *Technical Standard*

2 **Extended API Set Part 1 (SANITY DRAFT)**

3 *The Open Group*



4

5 © *April 2006, The Open Group*

6 All rights reserved.

7 No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in
8 any form or by any means, electronic, mechanical, photocopying, recording or otherwise,
9 without the prior permission of the copyright owners.

10 Technical Standard

11 Extended API Set Part 1 (SANITY DRAFT)

12 Document Number: <doc_no>

13 Published in the U.K. by The Open Group, April 2006.

14 Any comments relating to the material contained in this document may be submitted to:

15 The Open Group
16 Thames Tower
17 37-45 Station Road
18 Reading
19 Berkshire, RG1 1LX
20 United Kingdom

21 or by Electronic Mail to:

22 OGSpecs@opengroup.org

/ Contents

24	Chapter 1	Introduction.....	1
25	1.1	Scope.....	1
26	1.2	Relationship to Other Formal Standards	1
27	Chapter 2	Changes to the Base Definitions Volume.....	3
28	2.1	Section 1.5.1, Codes	3
29	2.2	Section 3.362, Stream.....	3
30	2.3	Chapter 13, Headers.....	3
31	Chapter 3	Changes to the Shell and Utilities Volume.....	5
32	Chapter 4	Changes to the System Interfaces Volume.....	7
33	4.1	Section 2.5, Standard I/O Streams	7
34	4.2	fclose() and fflush()	7
35	4.3	Reference Pages.....	8
36		<i>alphasort()</i>	9
37		<i>dirfd()</i>	11
38		<i>dprintf()</i>	13
39		<i>fmemopen()</i>	14
40		<i>getdelim()</i>	17
41		<i>mbsnrtowcs()</i>	19
42		<i>mkdtemp()</i>	21
43		<i>open_memstream()</i>	23
44		<i>psiginfo()</i>	25
45		<i>stpncpy()</i>	26
46		<i>stpncpy()</i>	27
47		<i>strndup()</i>	28
48		<i>strlen()</i>	29
49		<i>strsignal()</i>	30
50		<i>wcpcpy()</i>	31
51		<i>wcpncpy()</i>	32
52		<i>wscasecmp()</i>	33
53		<i>wcsdup()</i>	34
54		<i>wcsncasecmp()</i>	35
55		<i>wcsnlen()</i>	36
56		<i>wcsnrtombs()</i>	37
57		Index.....	39

59 The Open Group

60 The Open Group is a vendor-neutral and technology-neutral consortium, whose vision of
61 Boundaryless Information Flow will enable access to integrated information within and between
62 enterprises based on open standards and global interoperability. The Open Group works with
63 customers, suppliers, consortia, and other standards bodies. Its role is to capture, understand,
64 and address current and emerging requirements, establish policies, and share best practices; to
65 facilitate interoperability, develop consensus, and evolve and integrate specifications and Open
66 Source technologies; to offer a comprehensive set of services to enhance the operational
67 efficiency of consortia; and to operate the industry's premier certification service, including
68 UNIX certification.

69 Further information on The Open Group is available at www.opengroup.org.

70 The Open Group has over 15 years' experience in developing and operating certification
71 programs and has extensive experience developing and facilitating industry adoption of test
72 suites used to validate conformance to an open standard or specification.

73 More information is available at www.opengroup.org/certification.

74 The Open Group publishes a wide range of technical documentation, the main part of which is
75 focused on development of Technical and Product Standards and Guides, but which also
76 includes white papers, technical studies, branding and testing documentation, and business
77 titles. Full details and a catalog are available at www.opengroup.org/bookstore.

78 As with all *live* documents, Technical Standards and Specifications require revision to align with
79 new developments and associated international standards. To distinguish between revised
80 specifications which are fully backwards-compatible and those which are not:

- 81 • A new *Version* indicates there is no change to the definitive information contained in the
82 previous publication of that title, but additions/extensions are included. As such, it *replaces*
83 the previous publication.
- 84 • A new *Issue* indicates there is substantive change to the definitive information contained in
85 the previous publication of that title, and there may also be additions/extensions. As such,
86 both previous and new documents are maintained as current publications.

87 Readers should note that Corrigenda may apply to any publication. Corrigenda information is
88 published at www.opengroup.org/corrigenda.

89 This Document

90 This document has been prepared by The Open Group Base Working Group. The Open Group
91 Base Working Group is considering submitting a number of API sets to the Austin Group as
92 input to the revision of the Base Specifications, Issue 6.

93 This is the first document in that set.

94

Trademarks

95 Boundaryless Information Flow TM is a trademark and UNIX[®] and The Open Group[®] are
96 registered trademarks of The Open Group in the United States and other countries.

97 All other trademarks are the property of their respective owners.

98

Acknowledgements

99 The contributions of the following to the development of this document are gratefully
100 acknowledged:

- 101 • The Open Group Base Working Group

Introduction

1

2 1.1 Scope

3 The purpose of this document is to define a set of new API extensions to further increase
4 application capture and hence portability for systems built upon the Single UNIX Specification,
5 Version 3.

6 The scope of this set of extensions has been to consider interfaces from existing open source
7 implementations, such as the GNU C library.

8 1.2 Relationship to Other Formal Standards

9 No decision has been made on whether these interfaces will be added to a future Technical
10 Standard of The Open Group, how these interfaces would announce themselves in the name
11 space, or whether related interfaces should be merged with existing reference pages. This
12 Technical Standard is being forwarded to the Austin Group for consideration as input to the
13 revision of the Base Specifications, Issue 6.

Changes to the Base Definitions Volume

14

15 It is proposed that these additions comprise a new Option Group called Extended Interfaces.

16 2.1 Section 1.5.1, Codes

17 Add a new margin code as follows:

18 UX Extended Interfaces

19 The functionality described is optional. The functionality described is also an extension to
20 the ISO C standard.

21 Where applicable, functions are marked with the UX margin legend in the SYNOPSIS
22 section. Where additional semantics apply to a function, the material is identified by use of
23 the UX margin legend.

24 **Notes:**

- 25 1. This section is repeated in XBD, XSH, and XCU and therefore will in XBD (Section 1.5.1),
26 XSH (Section 1.8.1), and XCU (Section 1.8.1).
- 27 2. The use of UX as a margin code is a placeholder and may change in the final publication.

28 2.2 Section 3.362, Stream

29 Add *fmemopen()* and *open_memstream()* to the list of functions that can create a stream, marked
30 with the UX margin legend and shaded.

31 2.3 Chapter 13, Headers

32 The following header file reference pages will need the following additions, marked with the UX
33 margin legend and shaded as part of the Extended Interfaces Option Group.

34 **<dirent.h>**

35 The following shall be declared as functions and may also be defined as macros. Function
36 prototypes shall be provided.

```
37 int alphasort(const struct dirent **, const struct dirent **);
38 int dirfd (DIR *);
39 int scandir (const char *, struct dirent ***,
40             int (*) (const struct dirent *),
41             int (*) (const struct dirent **, const struct dirent **));
```

42 **<signal.h>**

43 The following shall be declared as functions and may also be defined as macros. Function
44 prototypes shall be provided.

```
45       void psignal (int, const char *);  
46       void psiginfo (siginfo_t *, const char *);
```

47 **<stdio.h>**

48 The following shall be declared as functions and may also be defined as macros. Function
49 prototypes shall be provided.

```
50       int dprintf (int, const char *, ...);  
51       FILE *fmemopen(void *, size_t, const char *);  
52       ssize_t getdelim (char **, size_t *, int, FILE *);  
53       ssize_t getline (char **, size_t *, FILE *);  
54       FILE *open_memstream(char **, size_t *);
```

55 **<stdlib.h>**

56 The following shall be declared as functions and may also be defined as macros. Function
57 prototypes shall be provided.

```
58       char *mkdtemp(char *);
```

59 **<string.h>**

60 The following shall be declared as functions and may also be defined as macros. Function
61 prototypes shall be provided.

```
62       char *stpcpy (char *, const char *);  
63       char *stpncpy (char *, const char *, size_t);  
64       char *strndup (const char *, size_t);  
65       size_t strlen (const char *, size_t);  
66       char *strsignal(int signum);
```

67 **<wchar.h>**

68 The following shall be declared as functions and may also be defined as macros. Function
69 prototypes shall be provided.

```
70       size_t mbsnrtowcs (wchar_t *, const char **, size_t, size_t, mbstate_t *);  
71       wchar_t *wcpncpy (wchar_t *, const wchar_t *);  
72       wchar_t *wcpncpy (wchar_t *, const wchar_t *, size_t);  
73       int wcscasecmp (const wchar_t *, const wchar_t *);  
74       wchar_t *wcsdup (const wchar_t *);  
75       int wcsncasecmp (const wchar_t *, const wchar_t *, size_t);  
76       size_t wcsnlen (const wchar_t *, size_t);  
77       size_t wcsnrtombs (char *, const wchar_t **, size_t, size_t, mbstate_t *);
```

Changes to the Shell and Utilities Volume

78

79 It is proposed that the following changes are made to Chapter 4, Utilities, the *ls* command.

80 **Note:** All page and line numbers in this proposal refer to the Shell and Utilities volume of
81 IEEE Std 1003.1-2001, 2004 Edition.

82 SYNOPSIS

83 In the SYNOPSIS section on Page 571, Line 22014 add the `-S` option by changing the SYNOPSIS
84 from:

85 UX `ls [-CFRacdilqrtul] [-H | -L] [-fgmnopsx] [file...]`

86 to:

87 UX `ls [-CFRSacdilqrtul] [-H | -L] [-fgmnopsx] [file...]`

88 OPTIONS

89 In the OPTIONS section after Page 571, Line 22054 add a description of the new `-S` option as
90 follows:

91 `-S` Sort with the primary key being file size (in decreasing order) and the secondary
92 key being filename in the collating sequence (in increasing order).

93 On Page 572, Lines 22065-22067 specify the interaction between the `-f` and `-S` options by
94 changing the description of the `-f` option from:

95 UX `-f` Force each argument to be interpreted as a directory and list the name found in
96 each slot. This option shall turn off `-l`, `-t`, `-s`, and `-r`, and shall turn on `-a`; the order
97 is the order in which entries appear in the directory.

98 to:

99 UX `-f` Force each argument to be interpreted as a directory and list the name found in
100 each slot. This option shall turn off `-l`, `-t`, `-S`, `-s`, and `-r`, and shall turn on `-a`; the
101 order is the order in which entries appear in the directory.

102 On Page 572, Line 22082 note the interaction between `-S` and `-r` by changing the description of
103 the `-r` option from:

104 `-r` Reverse the order of the sort to get reverse collating sequence or oldest first.

105 to:

106 `-r` Reverse the order of the sort to get reverse collating sequence oldest first, or
107 smallest file size first depending on the other options given.

108 On Page 572, Lines 22092-22094 add `-t` and `-S` to the list of mutually-exclusive options by
109 changing from:

110 Specifying more than one of the options in the following mutually-exclusive pairs shall not be
111 UX considered an error: `-C` and `-l` (ell), `-m` and `-l` (ell), `-x` and `-l` (ell), `-C` and `-l` (one), `-H` and `-L`,
112 `-c` and `-u`. The last option specified in each pair shall determine the output format.

113 to:

114 Specifying more than one of the options in the following mutually-exclusive pairs shall not be
115 UX considered an error: **-C** and **-I** (ell), **-m** and **-I** (ell), **-x** and **-I** (ell), **-C** and **-I** (one), **-H** and **-L**,
116 **-c** and **-u**, **-t** and **-S**. The last option specified in each pair shall determine the output format.

117 **RATIONALE**

118 Add a new paragraph after Page 577, Line 22291:

119 The **-S** option was added to the standard in Issue 7, but had been provided by several
120 implementations for many years. The description given in the standard documents historic
121 practice, but does not match much of the documentation that described its behavior. Historical
122 documentation typically described it as something like:

123 **-S** Sort by size (largest size first) instead of by name. Special character devices (listed
124 last) are sorted by name.

125 even though the file type was never considered when sorting the output. Character special files
126 do typically sort close to the end of the list because their file size on most implementations is
127 zero. But they are sorted alphabetically with any other files that happen to have the same file
128 size (zero), not sorted separately and added to the end.

Changes to the System Interfaces Volume

129

130 It is proposed that the following changes are made to Section 2.5, Standard I/O Streams.

131 **Note:** The text described in this proposal refers to the System Interfaces volume of IEEE Std 1003.1,
132 2004 Edition.

133 4.1 Section 2.5, Standard I/O Streams

134 Change the first sentence to:

135 UX A stream is associated with an external file (which may be a physical device) or memory buffer
136 UX by “opening” a file or buffer. This may involve “creating” a new file.

137 Add the following to the end:

138 UX A stream associated with a memory buffer shall have the same operations for text files that a
139 stream associated with an external file would have. In addition, the stream orientation shall be
140 determined in exactly the same fashion.

141 Input and output operations on a stream associated with a memory buffer by a call to
142 *fmemopen()* shall be constrained by the implementation to take place within the bounds of the
143 memory buffer. In the case of a stream opened by *open_memstream()* or *open_wmemstream()*, the
144 memory area shall grow dynamically to accommodate write operations as necessary. For output,
145 data is moved from the buffer provided by *setvbuf()* to the memory stream during a flush or
146 close operation.

147 4.2 *fclose()* and *fflush()*

148 Add the following to the “shall fail” section within the ERRORS section:

149 [ENOMEM] The underlying stream was created by *open_memstream()* or
150 *open_wmemstream()* and insufficient memory is available.

151 Update the [ENOSPC] error condition to:

152 [ENOSPC] There was no free space remaining on the device containing the file or in the
153 buffer used by the *fmemopen()* function.

154 **4.3 Reference Pages**

155 Add the following new system interface descriptions in alphabetical order with the existing
156 system interface descriptions in Chapter 3, System Interfaces.

157 **NAME**

158 alphasort, scandir — scan a directory

159 **SYNOPSIS**

```
160 ux      #include <dirent.h>
161
162      int alphasort(const struct dirent **d1, const struct dirent **d2);
163
164      int scandir(const char *dir, struct dirent ***namelist,
165                 int (*sel)(const struct dirent *),
166                 int (*compar)(const struct dirent **, const struct dirent **));
```

166 **DESCRIPTION**

167 The *alphasort()* function can be used as the comparison function for the *scandir()* function to sort
 168 the directory entries into alphabetical order, as if by the *strcoll()* function. Its parameters are the
 169 two directory entries, *d1* and *d2*, to compare.

170 The *scandir()* function shall scan the directory *dir*, calling the function referenced by *sel* on each
 171 directory entry. Entries for which the function referenced by *sel* returns non-zero shall be stored
 172 in strings allocated as if by a call to *malloc()*, and sorted using *qsort()* with the comparison
 173 function *compar()*, and collected in array *namelist* which shall be allocated as if by a call to
 174 *malloc()*. If *sel* is a null pointer, all entries shall be selected.

175 **RETURN VALUE**

176 Upon successful completion, *alphasort()* shall return an integer greater than, equal to, or less
 177 than 0, according to whether the name of the directory entry pointed to by *d1* is lexically greater
 178 than, equal to, or less than the directory pointed to by *d2* when both are interpreted as
 179 appropriate to the current locale. There is no return value reserved to indicate an error.

180 Upon successful completion, the *scandir()* function shall return the number of entries in the
 181 array and a pointer to the array through the parameter *namelist*. Otherwise, the *scandir()*
 182 function shall return -1 .

183 **ERRORS**

184 The *scandir()* function shall fail if:

185 [EACCES] Search permission is denied for the component of the path prefix of *dir* or read
 186 permission is denied for *dir*.

187 [ELOOP] A loop exists in symbolic links encountered during resolution of the *dir*
 188 argument.

189 [ENAMETOOLONG] The length of the *dir* argument exceeds {PATH_MAX} or a pathname
 190 component is longer than {NAME_MAX}.

192 [ENOENT] A component of *dir* does not name an existing directory or *dir* is an empty
 193 string.

194 [ENOMEM] Insufficient storage space is available.

195 [ENOTDIR] A component of *dir* is not a directory.

196 The *scandir()* function may fail if:

197 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
 198 resolution of the *dir* argument.

199 [EMFILE] {OPEN_MAX} file descriptors are currently open in the calling process.

200 [ENAMETOOLONG]
201 As a result of encountering a symbolic link in resolution of the *dir* argument,
202 the length of the substituted pathname string exceeded {PATH_MAX}.

203 [ENFILE] Too many files are currently open in the system.

204 EXAMPLES

205 An example to print the files in the current directory:

```
206 #include <dirent.h>
207 #include <stdio.h>
208 ...
209 struct dirent **namelist;
210 int i,n;
211
212     n = scandir(".", &namelist, 0, alphasort);
213     if (n < 0)
214         perror("scandir");
215     else {
216         for (i = 0; i < n; i++) {
217             printf("%s\n", namelist[i]->d_name);
218             free(namelist[i]);
219         }
220     free(namelist);
221     ...
```

222 APPLICATION USAGE

223 These functions are part of the Extended Interfaces Option Group and need not be available on
224 all implementations.

225 RATIONALE

226 None.

227 FUTURE DIRECTIONS

228 None.

229 SEE ALSO

230 *compar()*, *malloc()*, *qsort()*, *strcoll()*, the Base Definitions volume of IEEE Std 1003.1-2001,
231 <dirent.h>

232 CHANGE HISTORY

233 First released in Issue X.

234 **NAME**

235 dirfd — extract the file descriptor used by a DIR stream

236 **SYNOPSIS**237 UX `#include <dirent.h>`238 `int dirfd(DIR *dirp);`

239

240 **DESCRIPTION**

241 The *dirfd()* function shall return a file descriptor referring to the same directory as the *dirp*
 242 argument. This file descriptor shall be closed by a call to *closedir()*. The behavior of future calls
 243 to *readdir()* and *readdir_r()* is undefined if the application attempts to alter the file position
 244 indicator using the returned file descriptor. The behavior of future calls to *closedir()*, *readdir()*,
 245 and *readdir_r()* is undefined if the application attempts to close the file descriptor.

246 **RETURN VALUE**

247 Upon successful completion, the *dirfd()* function shall return an integer which contains a file
 248 descriptor for the stream pointed to by *dirp*. Otherwise, it shall return -1 and may set *errno* to
 249 indicate the error.

250 **ERRORS**251 The *dirfd()* function may fail if:252 [EINVAL] The *dirp* argument does not refer to a valid directory stream.253 [ENOTSUP] The implementation does not support the association of a file descriptor with
254 a directory.255 **EXAMPLES**

256 None.

257 **APPLICATION USAGE**

258 The *dirfd()* function is part of the Extended Interfaces Option Group and need not be available
 259 on all implementations.

260 The *dirfd()* function is intended to be a mechanism by which an application may obtain a file
 261 descriptor to use for the *fchdir()* function.

262 **RATIONALE**

263 This interface was introduced because the Base Definitions volume of IEEE Std 1003.1-2001 does
 264 not make public the **DIR** data structure. Applications tend to use the *fchdir()* function on the file
 265 descriptor returned by this interface, and this has proven useful for security reasons; in
 266 particular, it is a better technique than others where directory names might change.

267 The description uses the term “a file descriptor” rather than “the file descriptor”. The
 268 implication intended is that an implementation that does not use an *fd* for *diropen()* could still
 269 *open()* the directory to implement the *dirfd()* function. Such a descriptor must be closed later
 270 during a call to *closedir()*.

271 An implementation that does not support file descriptors referring to directories may fail with
 272 [ENOTSUP].

273 If it is necessary to allocate an *fd* to be returned by *dirfd()*, it should be done at the time of a call
 274 to *opendir()*.

275 **FUTURE DIRECTIONS**

276 None.

277 **SEE ALSO**278 *closedir()*, *diropen()*, *fchdir()*, *fileno()*, *open()*, *opendir()*, *readdir()*, *readdir_r()*, the Base Definitions
279 volume of IEEE Std 1003.1-2001, **<dirent.h>**, **<stdio.h>**280 **CHANGE HISTORY**

281 First released in Issue X.

282 **NAME**

283 dprintf — formatted output conversion to a file descriptor

284 **SYNOPSIS**

285 ux #include <stdio.h>

286 int dprintf(int *filides*, const char **format*, ...);

287

288 **DESCRIPTION**289 The *dprintf()* function shall be equivalent to the *fprintf()* function, except that *dprintf()* shall
290 write output to the file associated with the file descriptor specified by the *filides* argument rather
291 than place output on a stream.292 **RETURN VALUE**293 Upon successful completion, the *dprintf()* function shall return the number of bytes transmitted.
294 If an output error was encountered, it shall return a negative value.295 **ERRORS**296 Refer to *fprintf()*.297 In addition, the *dprintf()* function may fail if:298 [EBADF] The *filides* argument is not a valid file descriptor.299 **EXAMPLES**

300 None.

301 **APPLICATION USAGE**302 The *dprintf()* function is part of the Extended Interfaces Option Group and need not be available
303 on all implementations.304 **RATIONALE**

305 None.

306 **FUTURE DIRECTIONS**

307 None.

308 **SEE ALSO**309 *fprintf()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>310 **CHANGE HISTORY**

311 First released in Issue X.

312 NAME

313 fmemopen — open a memory buffer stream

314 SYNOPSIS

315 ux #include <stdio.h>

316 FILE *fmemopen(void *restrict buf, size_t size,
317 const char *restrict mode);
318

319 DESCRIPTION

320 The *fmemopen()* function shall associate the buffer given by the *buf* and *size* arguments with a
321 stream. The *buf* argument shall be either a null pointer or point to a buffer that is at least *size*
322 bytes long.323 The *mode* argument is a character string having one of the following values:324 *r* or *rb* Open the stream for reading.325 *w* or *wb* Open the stream for writing.326 *a* or *ab* Append; open the stream for writing at the first null byte.327 *r+* or *rb+* or *r+b* Open the stream for update (reading and writing).328 *w+* or *wb+* or *w+b* Open the stream for update (reading and writing). Truncate the buffer
329 contents.330 *a+* or *ab+* or *a+b* Append; open the stream for update (reading and writing); the initial
331 position is at the first null byte.

332 The character 'b' shall have no effect.

333 If a null pointer is specified as the *buf* argument, *fmemopen()* shall allocate *size* bytes of memory
334 as if by a call to *malloc()*. This buffer shall be automatically freed when the stream is closed.
335 Because this feature is only useful when the stream is opened for updating (because there is no
336 way to get a pointer to the buffer) the *fmemopen()* call may fail if the *mode* argument does not
337 include a '+'.
338339 The stream maintains a current position in the buffer. This position is initially set to either the
340 beginning of the buffer (for *r* and *w* modes) or to the first null byte in the buffer (for *a* modes). If
341 no null byte is found in append mode, the initial position is set to one byte after the end of the
342 buffer.342 If *buf* is a null pointer, the initial position shall always be set to the beginning of the buffer.343 The stream also maintains the size of the current buffer contents. For modes *r* and *r+* the size is
344 set to the value given by the *size* argument. For modes *w* and *w+* the initial size is zero and for
345 modes *a* and *a+* the initial size is either the position of the first null byte in the buffer or the value
346 of the *size* argument if no null byte is found.347 A read operation on the stream cannot advance the current buffer position behind the current
348 buffer size. Reaching the buffer size in a read operation counts as “end-of-file”. Null bytes in the
349 buffer have no special meaning for reads. The read operation starts at the current buffer position
350 of the stream.351 A write operation starts either at the current position of the stream (if mode has not specified
352 'a' as the first character) or at the current size of the stream (if mode had 'a' as the first
353 character). If the current position at the end of the write is larger than the current buffer size, the
354 current buffer size is set to the current position. A write operation on the stream cannot advance
355 the current buffer size behind the size given in the *size* argument.

356 When a stream open for writing is flushed or closed, a null byte is written at the current position
 357 or at the end of the buffer, depending on the size of the contents. If a stream open for update is
 358 flushed or closed and the last write has advanced the current buffer size, a null byte is written at
 359 the end of the buffer if it fits.

360 An attempt to seek a memory buffer stream to a negative position or to a position larger than the
 361 buffer size given in the *size* argument shall fail.

362 RETURN VALUE

363 Upon successful completion, *fmemopen()* shall return a pointer to the object controlling the
 364 stream. Otherwise, a null pointer shall be returned, and *errno* shall be set to indicate the error.

365 ERRORS

366 The *fmemopen()* function shall fail if:

367 [EINVAL] The *size* argument specifies a buffer size of zero.

368 The *fmemopen()* function may fail if:

369 [EINVAL] The value of the *mode* argument is not valid.

370 [EINVAL] The *buf* argument is a null pointer and the *mode* argument does not include a
 371 '+' character.

372 [ENOMEM] The *buf* argument is a null pointer and the allocation of a buffer of length *size*
 373 has failed.

374 [EMFILE] {FOPEN_MAX} streams are currently open in the calling process.

375 EXAMPLES

```
376 #include <stdio.h>
377 static char buffer[] = "foobar";
378 int
379 main (void)
380 {
381     int ch;
382     FILE *stream;
383
384     stream = fmemopen(buffer, strlen (buffer), "r");
385     if (stream == NULL)
386         /* handle error */;
387
388     while ((ch = fgetc(stream)) != EOF)
389         printf("Got %c\n", ch);
390
391     fclose(stream);
392     return (0);
393 }
```

391 This program produces the following output:

```
392 Got f
393 Got o
394 Got o
395 Got b
396 Got a
397 Got r
```

398 APPLICATION USAGE

399 The *fmemopen()* function is part of the Extended Interfaces Option Group and need not be
400 available on all implementations.

401 RATIONALE

402 This interface has been introduced to eliminate many of the errors encountered in the
403 construction of strings, notably overflowing of strings. This interface prevents overflow.

404 FUTURE DIRECTIONS

405 None.

406 SEE ALSO

407 *fdopen()*, *fopen()*, *freopen()*, *malloc()*, the Base Definitions volume of IEEE Std 1003.1-2001,
408 <stdio.h>

409 CHANGE HISTORY

410 First released in Issue X.

411 **NAME**412 getdelim, getline — read a delimited record from *stream*413 **SYNOPSIS**414 UX `#include <stdio.h>`415 `ssize_t getdelim(char **lineptr, size_t *n, int delimiter,`
416 `FILE *stream);`417 `ssize_t getline(char **lineptr, size_t *n, FILE *stream);`
418419 **DESCRIPTION**420 The *getdelim()* function shall read from *stream* until it encounters a character matching the
421 *delimiter* character. The argument *delimiter* (when converted to a **char**) shall specify the character
422 that terminates the read process.423 The *delimiter* argument is an **int**, the value of which the application shall ensure is a character
424 representable as an **unsigned char** or equal value to the macro EOF. If the *delimiter* argument has
425 any other value, the behavior is undefined.426 The application shall ensure that **lineptr* is a valid argument that could be passed to the *free()*
427 function. If **n* is non-zero, the application shall ensure that **lineptr* points to an object of size at
428 least **n* bytes.429 The size of the object pointed to by **lineptr* shall be increased to fit the incoming line, if it isn't
430 already large enough. The characters read shall be stored in the string pointed to by the *lineptr*
431 argument.432 The *getline()* function shall be equivalent to the *getdelim()* function with the *delimiter* character
433 equal to the <newline> character.434 **RETURN VALUE**435 Upon successful completion, the *getdelim()* function shall return the number of characters
436 written into the buffer, including the delimiter character if one was encountered before EOF.
437 Otherwise, it shall return `-1` and set *errno* to indicate the error.438 **ERRORS**

439 These functions shall fail if:

440 [EINVAL] When *lineptr* or *n* are a null pointer.

441 [ENOMEM] Insufficient memory is available.

442 These functions may fail if:

443 [EINVAL] *stream* is not a valid file descriptor.444 [EOVERFLOW] More than {SSIZE_MAX} characters were read without encountering the
445 *delimiter* character.

446 **EXAMPLES**

```
447     #include <stdio.h>
448     #include <stdlib.h>
449
450     int main(void)
451     {
452         FILE * fp;
453         char * line = NULL;
454         size_t len = 0;
455         ssize_t read;
456         fp = fopen("/etc/motd", "r");
457         if (fp == NULL)
458             exit(1);
459         while ((read = getline(&line, &len, fp)) != -1) {
460             printf("Retrieved line of length %zu :\n", read);
461             printf("%s", line);
462         }
463         if (line)
464             free(line);
465         fclose(fp);
466         return 0;
467     }
```

467 **APPLICATION USAGE**

468 These functions are part of the Extended Interfaces Option Group and need not be available on
469 all implementations.

470 Setting **lineptr* to a null pointer and **n* to zero are allowed and a recommended way to start
471 parsing a file.

472 **RATIONALE**

473 These functions are widely used to solve the problem that the *fgets()* function has with long
474 lines. The functions automatically enlarge the target buffers if needed. These are especially
475 useful since they reduce code needed for applications.

476 **FUTURE DIRECTIONS**

477 None.

478 **SEE ALSO**

479 *fgets()*, *free()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdio.h>

480 **CHANGE HISTORY**

481 First released in Issue X.

482 **NAME**483 `mbsnrto wcs` — convert a multi-byte string to a wide-character string484 **SYNOPSIS**485 `UX` `#include <wchar.h>`486 `size_t mbsnrto wcs(wchar_t *restrict dst, const char **restrict src,`
487 `size_t nmc, size_t len, mbstate_t *restrict ps);`
488489 **DESCRIPTION**490 The `mbsnrto wcs()` function works like the `mbsrtowcs()` function, except that the conversion of
491 characters pointed to by `src` is limited to at most `nmc` bytes (the size of the input buffer).492 If `dst` is not a null pointer, then `mbsnrto wcs()` shall attempt to convert `nmc` bytes from the multi-
493 byte string pointed to by `src` into a wide-character string starting at `dst`. No more than `len` wide
494 characters shall be written to `dst`. The shift state, pointed at by `ps`, is updated by the conversion.
495 Each conversion shall take place, as if by repeated calls to `mbrtowc(dst, *src, n, ps)`, where `n` is a
496 positive number. As long as this call succeeds, it is repeated, each time incrementing `dst` by one
497 and `*src` by the number of bytes converted.

498 Conversion shall stop early if any of the following cases occurs:

- 499
1. An invalid sequence of bytes was encountered in the `src` buffer. Under these conditions `*src`
500 is left pointing to the bytes which caused the conversion to halt. `-1` is returned, and `errno` is
501 set to `[EILSEQ]`.
 2. Either the `nmc` limit has been reached, or `len` non-null wide characters have already been
502 stored in `dst`. Here, `*src` is left to point to the next multi-byte sequence that has not been
503 converted, and the total number of wide characters written to `dst` is returned.
 3. The conversion of the multi-byte buffer pointed to by `src` has been completed by
504 encountering a null byte. In this case `*src` is set to a null pointer, `*ps` is returned to its initial
505 state, and the number of wide characters written to `dst`, excluding the terminating null
506 character, is returned.

509 When `dst` is a null pointer, the conversion proceeds as above, except that no wide characters are
510 written to memory, and the `len` argument is ignored, so no destination length limit is imposed.511 In either case, if `ps` is a null pointer, `mbsnrto wcs()` shall use its own internal `mbstate_t` object,
512 which is initialized at program start-up to the initial conversion state. Otherwise, the `mbstate_t`
513 object pointed to by `ps` shall be used to completely describe the current conversion state of the
514 associated character sequence.515 It is the responsibility of the calling program to ensure that `dst` is large enough to hold at least `len`
516 wide characters.517 **RETURN VALUE**518 The `mbsnrto wcs()` function shall return the number of characters successfully converted, not
519 including the terminating null (if any). If an error occurs, `mbsnrto wcs()` shall return `-1` and may
520 set `errno` to indicate the error.521 **ERRORS**522 The `mbsnrto wcs()` function may fail if:523 `[EILSEQ]` An invalid multi-byte sequence was encountered.

524 **EXAMPLES**

525 None.

526 **APPLICATION USAGE**527 The *mbsnrto wcs()* function is part of the Extended Interfaces Option Group and need not be
528 available on all implementations.529 **RATIONALE**

530 None.

531 **FUTURE DIRECTIONS**

532 None.

533 **SEE ALSO**534 *iconv()*, *mbsrtowcs()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**wchar.h**>535 **CHANGE HISTORY**

536 First released in Issue X.

537 **NAME**

538 mkdtemp — create a unique directory

539 **SYNOPSIS**540 ux `#include <stdlib.h>`541 `char *mkdtemp(char *template);`

542

543 **DESCRIPTION**

544 The *mkdtemp()* function uses the contents of *template* to construct a unique directory name. The
 545 string provided in *template* shall be a filename ending with six trailing 'X's. The *mkdtemp()*
 546 function shall replace each 'X' with a character from the portable filename character set. The
 547 characters are chosen such that the resulting name does not duplicate the name of an existing
 548 file at the time of a call to *mkdtemp()*. The unique directory name is used to attempt to create the
 549 directory using mode 0700 as modified by the file creation mask.

550 **RETURN VALUE**

551 Upon successful completion, the *mkdtemp()* function shall return a pointer to the string
 552 containing the directory name if it was created. Otherwise, it shall return a null pointer and shall
 553 set *errno* to indicate the error.

554 **ERRORS**555 The *mkdtemp()* function shall fail if:

556 [EACCES] Search permission is denied on a component of the path prefix, or write
 557 permission is denied on the parent directory of the directory to be created.

558 [EINVAL] The string pointed to by *template* does not end in "XXXXXXX".

559 [ELOOP] A loop exists in symbolic links encountered during resolution of the path of
 560 the directory to be created.

561 [EMLINK] The link count of the parent directory would exceed {LINK_MAX}.

562 [ENAMETOOLONG]

563 The length of the *template* argument exceeds {PATH_MAX} or a pathname
 564 component is longer than {NAME_MAX}.

565 [ENOENT] A component of the path prefix specified by the *template* argument does not
 566 name an existing directory or path is an empty string.

567 [ENOSPC] The file system does not contain enough space to hold the contents of the new
 568 directory or to extend the parent directory of the new directory.

569 [ENOTDIR] A component of the path prefix is not a directory.

570 [EROFS] The parent directory resides on a read-only file system.

571 The *mkdtemp()* function may fail if:

572 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
 573 resolution of the path of the directory to be created.

574 [ENAMETOOLONG]

575 As a result of encountering a symbolic link in resolution of the path of the
 576 directory to be created, the length of the substituted pathname string
 577 exceeded {PATH_MAX}.

578 **EXAMPLES**

579 None.

580 **APPLICATION USAGE**

581 The *mkdtemp()* function is part of the Extended Interfaces Option Group and need not be
582 available on all implementations.

583 **RATIONALE**

584 None.

585 **FUTURE DIRECTIONS**

586 None.

587 **SEE ALSO**

588 *mkdir()*, the Base Definitions volume of IEEE Std 1003.1-2001, <stdlib.h>

589 **CHANGE HISTORY**

590 First released in Issue X.

591 **NAME**

592 open_memstream, open_wmemstream — open a dynamic memory buffer stream

593 **SYNOPSIS**

594 ux #include <stdio.h>

595 FILE *open_memstream(char **bufp, size_t *sizep);

596 #include <wchar.h>

597 FILE *open_wmemstream(wchar_t **bufp, size_t *sizep);

598

599 **DESCRIPTION**600 The *open_memstream()* and *open_wmemstream()* functions shall create an I/O stream associated
601 with a dynamically allocated memory buffer. The stream shall be opened for writing and shall
602 be seekable.603 The stream associated with a call to *open_memstream()* shall be byte-oriented.604 The stream associated with a call to *open_wmemstream()* shall be wide-oriented.605 The stream shall maintain a current position in the allocated buffer and a current buffer length.
606 The position shall be initially set to zero (the start of the buffer). Each write to the stream shall
607 start at the current position and move this position by the number of successfully written bytes
608 for *open_memstream()* or the number of successfully written wide characters for
609 *open_wmemstream()*. The length shall be initially set to zero. If a write moves the position to a
610 value larger than the current length, the current length shall be set to this position. In this case a
611 null character for *open_memstream()* or a null wide character for *open_wmemstream()* shall be
612 appended to the current buffer. For both functions the terminating null is not included in the
613 calculation of the buffer length.614 After a successful *fflush()* or *fclose()*, the pointer referenced by *bufp* shall contain the address of
615 the buffer, and the variable pointed to by *sizep* shall contain the number of successfully written
616 bytes for *open_memstream()* or the number of successfully written wide characters for
617 *open_wmemstream()*. The buffer shall be terminated by a null character for *open_memstream()* or a
618 null wide character for *open_wmemstream()*.619 After a successful *fflush()* the pointer referenced by *bufp* and the variable referenced by *sizep*
620 remain valid only until the next write operation on the stream or a call to *fclose()*.621 **RETURN VALUE**622 Upon successful completion, these functions shall return a pointer to the object controlling the
623 stream. Otherwise, a null pointer shall be returned, and *errno* shall be set to indicate the error.624 **ERRORS**

625 These functions may fail if:

626 [EINVAL] *bufp* or *sizep* are NULL.

627 [EMFILE] {FOPEN_MAX} streams are currently open in the calling process.

628 [ENOMEM] Memory for the stream or the buffer could not be allocated.

629 **EXAMPLES**

```
630     #include <stdio.h>
631     int main (void)
632     {
633         FILE *stream;
634         char *buf;
635         size_t len;
636
637         stream = open_memstream(&buf, &len);
638         if (stream == NULL)
639             /* handle error */;
640
641         fprintf(stream, "hello my world");
642         fflush(stream);
643         printf("buf=%s, len=%zu\n", buf, len);
644         fseeko(stream, 0, SEEK_SET);
645         fprintf(stream, "good-bye");
646         fclose(stream);
647         printf("buf=%s, len=%zu\n", buf, len);
648         free(buf);
649         return 0;
650     }
```

649 This program produces the following output:

```
650     buf=hello my world, len=14
651     buf=good-bye world, len=14
```

652 **APPLICATION USAGE**

653 These functions are part of the Extended Interfaces Option Group and need not be available on
654 all implementations.

655 The buffer created by these functions should be freed by the application after closing the stream,
656 by means of a call to *free()*.

657 **RATIONALE**

658 These functions are similar to *fmemopen()* except that the memory is always allocated
659 dynamically by the function, and the stream is opened only for output.

660 **FUTURE DIRECTIONS**

661 None.

662 **SEE ALSO**

663 *fclose()*, *fdopen()*, *fflush()*, *fopen()*, *fmemopen()*, *free()*, *freopen()*, the Base Definitions volume of
664 IEEE Std 1003.1-2001, *<stdio.h>*

665 **CHANGE HISTORY**

666 First released in Issue X.

667 **NAME**

668 psiginfo, psignal — print signal information to standard error

669 **SYNOPSIS**670 UX `#include <signal.h>`671 `void psiginfo(siginfo_t *pinfo, const char *message);`672 `void psignal(int signum, const char *message);`

673

674 **DESCRIPTION**

675 The *psiginfo()* and *psignal()* functions shall print a message out on *stderr* associated with a signal
676 number. If *message* is not null and is not the empty string, then the string pointed to by the
677 *message* argument shall be printed first, followed by a colon, a space, and the signal description
678 string indicated by *signum*, or by the signal associated with *pinfo*. If the *message* argument is null
679 or points to an empty string, then only the signal description shall be printed. For *psiginfo()*, the
680 argument *pinfo* references a valid **siginfo_t** structure. For *psignal()*, if *signum* is not a valid signal
681 number, the behavior is implementation-defined.

682 **RETURN VALUE**

683 These functions shall not return a value.

684 **ERRORS**

685 No errors are defined.

686 **EXAMPLES**

687 None.

688 **APPLICATION USAGE**

689 These functions are part of the Extended Interfaces Option Group and need not be available on
690 all implementations.

691 **RATIONALE**

692 System V historically has *psignal()* and *psiginfo()* in **<siginfo.h>**. However, the **<siginfo.h>**
693 header is not specified in the Base Definitions volume of IEEE Std 1003.1-2001, and the type
694 **siginfo_t** is defined in **<signal.h>**.

695 **FUTURE DIRECTIONS**

696 None.

697 **SEE ALSO**698 *perror()*, *strsignal()*, the Base Definitions volume of IEEE Std 1003.1-2001, **<signal.h>**699 **CHANGE HISTORY**

700 First released in Issue X.

701 **NAME**

702 stpcpy — copy a string and return a pointer to the end of the result

703 **SYNOPSIS**704 ux `#include <string.h>`705 `char *stpcpy(char *restrict dst, const char *restrict src);`

706

707 **DESCRIPTION**

708 The *stpcpy()* function shall be equivalent to *strcpy()*, copying the string pointed to by *src* into the
709 array pointed to by *dst*, with the exception that *stpcpy()* shall return a pointer to the terminating
710 null byte in *dst*, rather than the beginning of this array, allowing succeeding calls to add
711 additional text to the *dst* array.

712 If copying takes place between objects that overlap, the behavior is undefined.

713 **RETURN VALUE**

714 The *stpcpy()* function shall return a pointer to the terminating null byte at the end of the *dst*
715 buffer. No return values are reserved to indicate an error.

716 **ERRORS**

717 No errors are defined.

718 **EXAMPLES**

719 The following example demonstrates the construction of a multi-part message in a single buffer.

720 `#include <string.h>`721 `#include <stdio.h>`722 `int`723 `main (void)`724 `{`725 `char buffer [10];`726 `char *name = buffer;`727 `name = stpcpy (stpcpy (stpcpy (name, "ice"), "-"), "cream");`728 `puts (buffer);`729 `return 0;`730 `}`731 **APPLICATION USAGE**

732 The *stpcpy()* function is part of the Extended Interfaces Option Group and need not be available
733 on all implementations.

734 **RATIONALE**

735 None.

736 **FUTURE DIRECTIONS**

737 None.

738 **SEE ALSO**739 *strcpy()*, the Base Definitions volume of IEEE Std 1003.1-2001, `<string.h>`740 **CHANGE HISTORY**

741 First released in Issue X.

742 **NAME**

743 stpncpy — copy fixed length string, returning a pointer to the array end

744 **SYNOPSIS**

745 ux #include <string.h>

746 char *stpncpy(char *restrict *dst*, const char *restrict *src*, size_t *size*);

747

748 **DESCRIPTION**749 The *stpncpy()* function shall be equivalent to the *strcpy()* function, with the added restriction
750 that it shall copy at most *size* bytes from *src* into *dst*.751 If *size* is less than or equal to the length of the string pointed to by *src* then no termination null
752 byte shall be inserted into the *dst* array after the *size* bytes have been copied.753 If *size* is greater than the length of the string pointed to by *src* then all of the bytes in *src* are
754 copied into the *dst* array. As many terminating null bytes are inserted as are needed to bring the
755 total bytes transferred equal to *size*.

756 If copying takes place between objects that overlap, the behavior is undefined.

757 **RETURN VALUE**758 If a null byte is written to the destination, the *stpncpy()* function shall return the address of the
759 first such null byte. Otherwise, it shall return *&src[size]*. No return values are reserved to
760 indicate an error.761 **ERRORS**

762 No errors are defined.

763 **EXAMPLES**764 **APPLICATION USAGE**765 The *stpncpy()* function is part of the Extended Interfaces Option Group and need not be
766 available on all implementations.767 Applications must provide the space in *dst* for the *size* bytes to be transferred, as well as ensure
768 that the *src* and *dst* arrays do not overlap.769 **RATIONALE**

770 None.

771 **FUTURE DIRECTIONS**

772 None.

773 **SEE ALSO**774 *strcpy()*, the Base Definitions volume of IEEE Std 1003.1-2001, <string.h>775 **CHANGE HISTORY**

776 First released in Issue X.

777 **NAME**

778 strndup — duplicate a specific number of bytes from a string

779 **SYNOPSIS**780 ux `#include <string.h>`781 `char *strndup(const char *string, size_t size);`

782

783 **DESCRIPTION**

784 The *strndup()* function shall be equivalent to the *strdup()* function, duplicating the provided
785 *string* in a new block of memory allocated as if by using *malloc()*, with the exception being that
786 *strndup()* copies at most *size* plus one bytes into the newly allocated memory, terminating the
787 new string with a null byte.

788 If the length of *string* is larger than *size*, only *size* bytes shall be duplicated. If *size* is larger than
789 the length of *string*, all bytes in *string* shall be copied into the new memory buffer, including the
790 terminating null byte. The newly created string shall always be properly terminated.

791 **RETURN VALUE**

792 Upon successful completion, the *strndup()* function shall return a pointer to the newly allocated
793 memory containing the duplicated string. Otherwise, it shall return a null pointer and set *errno*
794 to indicate the error.

795 **ERRORS**796 The *strndup()* function shall fail if:

797 [ENOMEM] Insufficient memory available for the target string.

798 **EXAMPLES**

799 None.

800 **APPLICATION USAGE**

801 The *strndup()* function is part of the Extended Interfaces Option Group and need not be
802 available on all implementations.

803 **RATIONALE**

804 None.

805 **FUTURE DIRECTIONS**

806 None.

807 **SEE ALSO**808 *malloc()*, *strdup()*, the Base Definitions volume of IEEE Std 1003.1-2001, <string.h>809 **CHANGE HISTORY**

810 First released in Issue X.

811 **NAME**

812 strnlen — determine length of fixed size string

813 **SYNOPSIS**814 UX `#include <string.h>`815 `size_t strnlen(const char *s, size_t maxlen);`

816

817 **DESCRIPTION**

818 The *strnlen()* function shall compute the smaller of the number of bytes in the string to which *s*
819 points, not including the terminating null byte, or the value of the *maxlen* argument. The
820 *strnlen()* function shall never examine more than *maxlen* bytes of the string pointed to by *s*.

821 **RETURN VALUE**

822 The *strnlen()* function shall return an integer containing the smaller of either the length of the
823 string pointed to by *s* or *maxlen*.

824 **ERRORS**

825 No errors are defined.

826 **EXAMPLES**

827 None.

828 **APPLICATION USAGE**

829 The *strnlen()* function is part of the Extended Interfaces Option Group and need not be available
830 on all implementations.

831 **RATIONALE**

832 None.

833 **FUTURE DIRECTIONS**

834 None.

835 **SEE ALSO**836 *strlen()*, the Base Definitions volume of IEEE Std 1003.1-2001, `<string.h>`837 **CHANGE HISTORY**

838 First released in Issue X.

839 **NAME**840 `strsignal` — get name of signal841 **SYNOPSIS**842 `UX` `#include <string.h>`843 `char *strsignal(int signum);`

844

845 **DESCRIPTION**

846 The `strsignal()` function shall map the signal number in `signum` to an implementation-defined
847 string and shall return a pointer to it. It shall use the same set of messages as the `psignal()`
848 function.

849 The string pointed to shall not be modified by the application, but may be overwritten by a
850 subsequent call to `strsignal()` or `setlocale()`.

851 The contents of the message strings returned by `strsignal()` should be determined by the setting
852 of the `LC_MESSAGES` category in the current locale.

853 The implementation shall behave as if no function defined in this standard calls `strsignal()`.

854 Since no return value is reserved to indicate an error, an application wishing to check for error
855 situations should set `errno` to 0, then call `strsignal()`, then check `errno`.

856 The `strsignal()` function need not be reentrant. A function that is not required to be reentrant is
857 not required to be thread-safe.

858 **RETURN VALUE**

859 Upon successful completion, `strsignal()` shall return a pointer to a string. Otherwise, if `signum` is
860 not a valid signal number, the return value is unspecified.

861 **ERRORS**

862 No errors are defined.

863 **EXAMPLES**

864 None.

865 **APPLICATION USAGE**

866 The `strsignal()` function is part of the Extended Interfaces Option Group and need not be
867 available on all implementations.

868 **RATIONALE**

869 If `signum` is not a valid signal number, some implementations return NULL, while for others the
870 `strsignal()` function returns a pointer to a string containing an unspecified message denoting an
871 unknown signal. This standard leaves this return value unspecified.

872 **FUTURE DIRECTIONS**

873 None.

874 **SEE ALSO**875 `perror()`, `psignal()`, `setlocale()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<string.h>`876 **CHANGE HISTORY**

877 First released in Issue X.

878 **NAME**879 `wcpcpy` — copy a wide-character string, returning a pointer to its end880 **SYNOPSIS**881 `UX` `#include <wchar.h>`882 `wchar_t *wcpcpy(wchar_t *restrict dst, const wchar_t *restrict src);`

883

884 **DESCRIPTION**

885 The `wcpcpy()` function is the wide-character equivalent of the `strcpy()` function. It shall copy the
886 wide-character string pointed to by `src`, including the terminating null wide-character code, into
887 the array pointed to by `dst`.

888 The application shall ensure that there is room for at least `wcslen(src)+1` wide characters in the
889 `dst` array, and that the `src` and `dst` arrays do not overlap.

890 **RETURN VALUE**

891 The `wcpcpy()` function shall return a pointer to the last wide character written into the `dst` array
892 that is a pointer to the terminating null wide-character code. No return value is reserved to
893 indicate an error.

894 **ERRORS**

895 No errors are defined.

896 **EXAMPLES**

897 None.

898 **APPLICATION USAGE**

899 The `wcpcpy()` function is part of the Extended Interfaces Option Group and need not be available
900 on all implementations.

901 **RATIONALE**

902 None.

903 **FUTURE DIRECTIONS**

904 None.

905 **SEE ALSO**906 `strcpy()`, `strcpy()`, `wscpy()`, the Base Definitions volume of IEEE Std 1003.1-2001, `<wchar.h>`907 **CHANGE HISTORY**

908 First released in Issue X.

909 **NAME**

910 wcpncpy — copy a fixed-size wide-character string, returning a pointer to its end

911 **SYNOPSIS**912 ux

```
#include <wchar.h>
```

913

```
wchar_t *wcpncpy(wchar_t restrict *dst, const wchar_t *restrict src,
```


914

```
size_t n);
```

915

916 **DESCRIPTION**917 The *wcpncpy()* function is the wide-character equivalent of the *stpncpy()* function. It shall copy
918 at most *n* wide characters from the string pointed to by *src*, including the terminating null wide-
919 character code, into the array pointed to by *dst*. Exactly *n* wide characters shall be written into
920 *dst*. If the length of *src* is smaller than *n*, remaining characters for *dst* are filled in using the
921 terminating null wide-character code. If the *src* array length is greater than or equal to *n*, then *n*
922 characters from *src* shall be copied to *dst* with no terminating null wide-character code in the *dst*
923 array.924 The application shall ensure that there is room for at least *n* wide characters in the *dst* array, and
925 that the *src* and *dst* arrays do not overlap.926 **RETURN VALUE**927 If any null wide-character codes were written into the *dst* array, the *wcpncpy()* function shall
928 return the address of the first such null wide-character code. Otherwise, it shall return *&dst[n]*.
929 No return values are reserved to indicate an error.930 **ERRORS**

931 No errors are defined.

932 **EXAMPLES**

933 None.

934 **APPLICATION USAGE**935 The *wcpncpy()* function is part of the Extended Interfaces Option Group and need not be
936 available on all implementations.937 **RATIONALE**

938 None.

939 **FUTURE DIRECTIONS**

940 None.

941 **SEE ALSO**942 *stpncpy()*, *wpcpy()*, *wscncpy()*, the Base Definitions volume of IEEE Std 1003.1-2001, *<wchar.h>*943 **CHANGE HISTORY**

944 First released in Issue X.

945 **NAME**946 `wcscasecmp` — compare two wide-character strings, ignoring case947 **SYNOPSIS**948 `UX` `#include <wchar.h>`949 `int wcscasecmp(const wchar_t *st1, const wchar_t *st2);`

950

951 **DESCRIPTION**952 The `wcscasecmp()` function is the wide-character equivalent of the `strcasecmp()` function.953 The `wcscasecmp()` function shall compare, while ignoring differences in case, the string pointed
954 to by `st1` to the string pointed to by `st2`.955 In the POSIX locale, `wcscasecmp()` shall behave as if the strings had been converted to lowercase
956 and then a character comparison performed. The results are unspecified in other locales.957 **RETURN VALUE**958 Upon completion, the `wcscasecmp()` function shall return an integer greater than, equal to, or less
959 than 0 if the wide-character string pointed to by `st1` is, ignoring case, greater than, equal to, or
960 less than the wide-character string pointed to by `st2`, respectively. No return value is reserved to
961 indicate an error.962 **ERRORS**

963 No errors are defined.

964 **EXAMPLES**

965 None.

966 **APPLICATION USAGE**967 The `wcscasecmp()` function is part of the Extended Interfaces Option Group and need not be
968 available on all implementations.969 **RATIONALE**

970 None.

971 **FUTURE DIRECTIONS**

972 None.

973 **SEE ALSO**974 `strcasecmp()`, `wscmp()`, `wcsncasecmp()`, the Base Definitions volume of IEEE Std 1003.1-2001,
975 `<wchar.h>`976 **CHANGE HISTORY**

977 First released in Issue X.

978 **NAME**

979 wcsdup — duplicate a wide-character string

980 **SYNOPSIS**

981 ux #include <wchar.h>

982 wchar_t *wcsdup(const wchar_t *string);

983

984 **DESCRIPTION**985 The *wcsdup()* function is the wide-character equivalent of the *strdup()* function.986 The *wcsdup()* function shall return a pointer to a new wide-character string, which is the
987 duplicate of the wide-character string *string*. The returned pointer can be passed to *free()*. A null
988 pointer is returned if the new wide-character string cannot be created.989 **RETURN VALUE**990 Upon successful completion, the *wcsdup()* function shall return a pointer to the newly allocated
991 wide-character string. Otherwise, it shall return a null pointer and set *errno* to indicate the error.992 **ERRORS**993 The *wcsdup()* function shall fail if:

994 [ENOMEM] Memory large enough for the duplicate string could not be allocated.

995 **EXAMPLES**

996 None.

997 **APPLICATION USAGE**998 The *wcsdup()* function is part of the Extended Interfaces Option Group and need not be available
999 on all implementations.1000 **RATIONALE**

1001 None.

1002 **FUTURE DIRECTIONS**

1003 None.

1004 **SEE ALSO**1005 *free()*, *strdup()*, *wscpy()*, the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>1006 **CHANGE HISTORY**

1007 First released in Issue X.

1008 **NAME**

1009 wcsncasecmp — compare two fixed-size wide-character strings, ignoring case

1010 **SYNOPSIS**

1011 ux #include <wchar.h>

1012 int wcsncasecmp(const wchar_t *st1, const wchar_t *st2, size_t n);

1013

1014 **DESCRIPTION**1015 The *wcsncasecmp()* function is the wide-character equivalent of the *strncasecmp()* function.1016 The *wcsncasecmp()* function shall compare, while ignoring differences in case, not more than *n*
1017 characters from the wide-character string pointed to by *st1* to the wide-character string pointed
1018 to by *st2*.1019 In the POSIX locale, *wcsncasecmp()* shall behave as if the strings had been converted to lowercase
1020 and then a character comparison performed. The results are unspecified in other locales.1021 **RETURN VALUE**1022 Upon completion, the *wcsncasecmp()* function shall return an integer greater than, equal to, or
1023 less than 0 if the possibly null wide-character terminated string pointed to by *st1* is, ignoring
1024 case, greater than, equal to, or less than the possibly null wide-character terminated string
1025 pointed to by *st2*, respectively. No return value is reserved to indicate an error.1026 **ERRORS**

1027 No errors are defined.

1028 **EXAMPLES**

1029 None.

1030 **APPLICATION USAGE**1031 The *wcsncasecmp()* function is part of the Extended Interfaces Option Group and need not be
1032 available on all implementations.1033 **RATIONALE**

1034 None.

1035 **FUTURE DIRECTIONS**

1036 None.

1037 **SEE ALSO**1038 *strncasecmp()*, *wscasecmp()*, *wcsncmp()*, the Base Definitions volume of IEEE Std 1003.1-2001,
1039 <wchar.h>1040 **CHANGE HISTORY**

1041 First released in Issue X.

1042 **NAME**

1043 wcsnlen — determine the length of a fixed-sized wide-character string

1044 **SYNOPSIS**

1045 ux #include <wchar.h>

1046 size_t wcsnlen(const wchar_t *wcs, size_t maxlen);

1047

1048 **DESCRIPTION**1049 The *wcsnlen()* function is the wide-character equivalent of the *strlen()* function.

1050 The *wcsnlen()* function shall compute the smaller of the number of wide characters in the string
1051 to which *wcs* points, not including the terminating null wide-character code, and the value of
1052 *maxlen*. The *wcsnlen()* function shall never examine more than the first *maxlen* characters of the
1053 wide-character string pointed to by *wcs*.

1054 **RETURN VALUE**

1055 The *wcsnlen()* function shall return an integer containing the smaller of either the length of the
1056 wide-character string pointed to by *wcs* or *maxlen*. No return value is reserved to indicate an
1057 error.

1058 **ERRORS**

1059 No errors are defined.

1060 **EXAMPLES**

1061 None.

1062 **APPLICATION USAGE**

1063 The *wcsnlen()* function is part of the Extended Interfaces Option Group and need not be
1064 available on all implementations.

1065 **RATIONALE**

1066 None.

1067 **FUTURE DIRECTIONS**

1068 None.

1069 **SEE ALSO**1070 *strlen()*, *wcslen()*, the Base Definitions volume of IEEE Std 1003.1-2001, <wchar.h>1071 **CHANGE HISTORY**

1072 First released in Issue X.

1073 **NAME**

1074 wcsnrtoombs — convert wide-character string to multi-byte string

1075 **SYNOPSIS**1076 **UX** #include <wchar.h>

```
1077       size_t wcsnrtoombs(char *dst, const wchar_t **src, size_t nwc,
1078                        size_t len, mbstate_t *ps);
```

1079

1080 **DESCRIPTION**

1081 The *wcsnrtoombs()* function shall be equivalent to the *wcsrtombs()* function, except that the
 1082 conversion is limited to the first *nwc* wide characters.

1083 The *wcsnrtoombs()* function shall convert a sequence of at most *nwc* wide characters from the
 1084 array indirectly pointed to by *src* into a sequence of corresponding characters, beginning in the
 1085 conversion state described by the object pointed to by *ps*. If *dst* is not a null pointer, the
 1086 converted characters shall then be stored into the array pointed to by *dst*. Conversion continues
 1087 up to and including a terminating null wide character, which shall also be stored. Conversion
 1088 shall stop earlier in the following cases:

- 1089 • When a code is reached that does not correspond to a valid character
- 1090 • When the next character would exceed the limit of *len* total bytes to be stored in the array
 1091 pointed to by *dst* (and *dst* is not a null pointer)
- 1092 • When *nwc* wide characters from *src* have been converted

1093 Each conversion shall take place as if by a call to the *wcrtomb()* function.

1094 If *dst* is not a null pointer, the pointer object pointed to by *src* shall be assigned either a null
 1095 pointer (if conversion stopped due to reaching a terminating null wide character) or the address
 1096 just past the last wide character converted (if any). If conversion stopped due to reaching a
 1097 terminating null wide character, the resulting state described shall be the initial conversion state.

1098 If *ps* is a null pointer, the *wcsnrtoombs()* function shall use its own internal **mbstate_t** object,
 1099 which is initialized at program start-up to the initial conversion state. Otherwise, the **mbstate_t**
 1100 object pointed to by *ps* shall be used to completely describe the current conversion state of the
 1101 associated character sequence. The implementation shall behave as if no function defined in
 1102 System Interfaces volume of IEEE Std 1003.1-2001 calls *wcsnrtoombs()*.

1103 **UX CX** If the application uses any of the `_POSIX_THREAD_SAFE_FUNCTIONS` or `_POSIX_THREADS`
 1104 functions, the application shall ensure that the *wcsnrtoombs()* function is called with a non-NULL
 1105 *ps* argument.

1106 The behavior of this function shall be affected by the *LC_CTYPE* category of the current locale.

1107 **RETURN VALUE**

1108 Refer to *wcsrtombs()*.

1109 **ERRORS**

1110 Refer to *wcsrtombs()*.

1111 **EXAMPLES**

1112 None.

1113 **APPLICATION USAGE**1114 The *wcsnrombs()* function is part of the Extended Interfaces Option Group and need not be
1115 available on all implementations.1116 **RATIONALE**

1117 None.

1118 **FUTURE DIRECTIONS**

1119 None.

1120 **SEE ALSO**1121 *wrtomb()*, *wcrtombs()*, the Base Definitions volume of IEEE Std 1003.1-2001, <**wchar.h**>1122 **CHANGE HISTORY**

1123 First released in Issue X.

Index

2	alphasort()	9
3	dirent.h.....	3
4	dirfd().....	11
5	dprintf().....	13
6	fmemopen().....	14
7	FOPEN_MAX.....	15
8	getdelim()	17
9	mbsnrtowcs()	19
10	mkdtemp().....	21
11	NAME_MAX	9
12	OPEN_MAX.....	9
13	open_memstream()	23
14	PATH_MAX.....	9
15	psiginfo().....	25
16	signal.h.....	4
17	stdio.h.....	4
18	stdlib.h.....	4
19	stpcpy()	26
20	stpncpy()	27
21	string.h	4
22	strndup()	28
23	strnlen().....	29
24	strsignal().....	30
25	wchar.h	4
26	wcpcpy()	31
27	wcpncpy().....	32
28	wscasecmp().....	33
29	wcsdup()	34
30	wcsncasecmp()	35
31	wcsnlen().....	36
32	wcsnrtombs()	37

