ISO/IEC JTC1 SC22 WG14 WG14/N1388

Date: 2009-06-01

Reference number of document: **ISO/IEC WDTR 24731-2**

Committee identification: ISO/IEC JTC1 SC22 WG14

SC22 Secretariat: ANSI

Information Technology —

Programming languages, their environments and system software interfaces —

Extensions to the C Library, —

Part II: Dynamic Allocation Functions

Document type: Technical Report Type 2
Document subtype: n/a
Document stage: (5) Publication Stage
Document language: E

Contents

Foreword

1 ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are member of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

2 Technical Reports are drafted in accordance with the rules given in the ISO/IEC Directives, Part 3. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft Technical Reports adopted by the joint technical committee are circulated to national bodies for voting. Publication as a Technical Report requires approval by at least 75% of the member bodies casting a vote.

3 The main task of technical committees is to prepare International Standards, but in exceptional circumstances a technical committee may propose the publication of a Technical Report of one of the following types:

— type 1, when the required support cannot be obtained for the publication of an International Standard, despite repeated efforts;

— type 2, when the subject is still under technical development or where for any other reason there is the future but not immediate possibility of an agreement on an International Standard;

— type 3, when a technical committee has collected data of a different kind from that which is normally published as an International Standard ("state of the art", for example).

4 Technical Reports of types 1 and 2 are subject to review within three years of publication, to decide whether they can be transformed into International Standards. Technical Reports of type 3 do not necessarily have to be reviewed until the data they provide are considered to be no longer valid or useful.

5 ISO/IEC TR 24731, which is a Technical Report of type 2, was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

Introduction

1 Traditionally, the C Library has contained many functions that trust the programmer to provide output character arrays big enough to hold the result being produced. Not only do these functions not check that the arrays are big enough, they frequently lack the information needed to perform such checks. While it is possible to write safe, robust, and error-free code using the existing library, the library tends to promote programming styles that lead to mysterious failures if a result is too big for the provided array.

2 Perhaps the most common programming style is to declare character arrays large enough to handle most practical cases. However, if the program encounters strings too large for it to process, data is written past the end of arrays overwriting other variables in the program. The program never gets any indication that a problem exists, and so never has a chance to recover or to fail gracefully.

3 Worse, this style of programming has compromised the security of computers and networks. Daemons are given carefully prepared data that overflows buffers and tricks the daemons into granting access that should be denied.

4 If the programmer writes run time checks to verify lengths before calling library functions, then those run time checks frequently duplicate work done inside the library functions, which discover string lengths as a side effect of doing their job.

5 This technical report provides alternative functions for the C library that promote safer, more secure programming. Part one provides simple replacement functions for the library functions of ISO/IEC 9899:1999 that provide bounds checking. Those function can be used as simple replacements for the original library functions in legacy code. This part of this technical report presents replacements for many of these functions that use dynamically allocated memory to ensure that buffer overflow does not occur. Since the use of such functions requires adding additional calls to free the buffers later, these functions are better suited to new developments than to retrofitting old code.

6 In general, the functions described in this part of this Technical Report provide much greater assurance that buffer overflow problems will not occur, since buffers are always automatically sized to hold the data required. With the bounds checking functions, if an invalid size was passed to one of the functions, it could still sufffer from buffer overflow problems, while appearing to have addressed such issues. Applications that use dynamic memory allocation might, however, suffer from denial of service attacks where data is presented until memory is exhausted.

7 These functions are drawn from existing implementations that have widespread usage. Many of these functions are included in ISO/IEC 9945:2003 (POSIX) and as such are aligned with that standard.

1. Scope

1 This Technical Report specifies a series of extensions of the programming language C, specified by International Standard ISO/IEC 9899:1999.

2 International Standard ISO/IEC 9899:1999 provides important context and specification for this Technical Report. Clause 4 of this Technical Report should be read as if it were merged into Subclause 6.10.8 of ISO/IEC 9899:1999. Clause 5 of this Technical Report should be read as if it were merged into the parallel structure of named Subclauses of Clause 7 of ISO/IEC 9899:1999.

2. References

2.1 Normative references

1 The following normative documents contain provisions which, through reference in this text, constitute provisions of this Technical Report. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply. However, parties to agreements based on this Technical Report are encouraged to investigate the possibility of applying the most recent editions of the normative documents indicated below. For undated references, the latest edition of the normative document referred to applies. Members of ISO and IEC maintain registers of currently valid International Standards.

2 ISO/IEC 9899:1999, *Information technology — Programming languages, their environments and system software interfaces — Programming Language C*.

3 ISO/IEC 9899:1999/Cor 1:2001, *Information technology — Programming languages, their environments and system software interfaces — Programming Language C — Technical Corrigendum 1* .

4 ISO/IEC 9899:1999/Cor 2:2004, *Information technology — Programming languages, their environments and system software interfaces — Programming Language C — Technical Corrigendum 2* .

5 ISO/IEC 9945:2003 (including Technical Corrigendum 1), *Information technology — Programming languages, their environments and system software interfaces — Portable Operating System Interface (POSIX®)*.

6 ISO/IEC 23360:2006, *Information technology — Programming languages, their environments and system software interfaces — Linux Standard Base*.

7 ISO/IEC 646, *Information technology — ISO 7-bit coded character set for information interchange*.

8 ISO/IEC 2382−1:1993, *Information technology — Vocabulary — Part 1: Fundamental terms*.

9 ISO/IEC 10646 (all parts), *Information technology — Universal Multiple-Octet Coded Character Set (UCS)*.

2.2 Relationship to other standards

1 Many of the interfaces in this specification are derived from interfaces specified in other ISO/IEC specifications, and in particular:

— ISO/IEC 9945:2003 (including Technical Corrigendum 1), *Information technology — Programming languages, their environments and system software interfaces — Portable Operating System Interface (POSIX®)*.

— ISO/IEC IS 23360:2006, *Information technology — Programming languages, their environments and system software interfaces — Linux Standard Base*.

2 Where an interface is described as being derived from either of these standards, the functionality described on this reference page is intended to be aligned with that standard. Any conflict between the requirements described here and the referenced standard is unintentional. This technical report defers to the underlying standard.

3. Terms, definitions, and symbols

1 Terms are defined where they appear in *italic* type. Terms explicitly defined in this Technical Report are not to be presumed to refer implicitly to similar terms defined elsewhere. Terms not defined in this Technical Report are to be interpreted according to ISO/IEC 9899:1999 and ISO/IEC 9945-1:2001.

4. Predefined macro names

1 The following macro name is conditionally defined by the implementation:

`__STDC_ALLOC_LIB__`	The integer constant `200nnnL`, intended to indicate conformance to this technical report.1)

1) The intention is that this will remain an integer constant of type `long int` that is increased with each revision of this technical report.

5. Library

5.1 Introduction

5.1.1 Standard headers

1 The functions, macros, and types defined in Clause 5 and its subclauses are not defined by their respective headers if `__STDC_WANT_LIB_EXT2__` is defined as a macro which expands to the integer constant `0` or is not defined as a macro at the point in the source file where the appropriate header is included.

2 The functions, macros, and types defined in Clause 5 and its subclauses are defined by their respective headers if `__STDC_WANT_LIB_EXT2__` is defined as a macro which expands to the integer constant `1` at the point in the source file where the appropriate header is included.2)

3 Within a preprocessing translation unit, `__STDC_WANT_LIB_EXT2__` shall be defined identically for all inclusions of any headers from Clause 5. If `__STDC_WANT_LIB_EXT2__` is defined differently for any such inclusion, the implementation shall issue a diagnostic as if a preprocessor error directive was used.

5.1.2 Reserved identifiers

1 Each macro name in any of the following subclauses is reserved for use as specified if it is defined by any of its associated headers when included; unless explicitly stated otherwise (see ISO/IEC 9899:1999 Subclause 7.1.4).

2 All identifiers with external linkage in any of the following subclauses are reserved for use as identifiers with external linkage if any of them are used by the program. None of them are reserved if none of them are used.

3 Each identifier with file scope listed in any of the following subclauses is reserved for use as a macro name and as an identifier with file scope in the same name space if it is defined by any of its associated headers when included.

2) Future revisions of this technical report may define meanings for other values of `__STDC_WANT_LIB_EXT2__`.

5.1.3 Use of errno

1 An implementation may set `errno` for the functions defined in this technical report, but is not required to.

5.2 Input/output `<stdio.h>`

5.2.1 Streams

1 In addition to the requirements of ISO/IEC 9899:1999, clause 7.19.2, streams may be associated with memory buffers.

2 A stream associated with a memory buffer has the same operations for text files that a stream associated with an external file would have. In addition, the stream orientation is determined in exactly the same fashion.

3 Input and output operations on a stream associated with a memory buffer by a call to `fmemopen`, `open_memstream` or `open_wmemstream`[3)] are constrained by the implementation to take place within the bounds of the memory buffer. In the case of a stream opened by `open_memstream` or `open_wmemstream`, the memory area grows dynamically to accommodate write operations as necessary. For output, data are moved from the buffer provided by `setvbuf` to the memory stream during a flush or close operation. If there is insufficient memory to grow the memory area, or the operation requires access outside of the associated memory area, the associated operation fails.

5.2.2 Operations on buffers

5.2.2.1 The `fmemopen` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT2__ 1
#include <stdio.h>
FILE * fmemopen(void * restrict buf,
         size_t size, const char * restrict mode);
```

Description

2 This interface is derived from POSIX. Any conflict between the requirements described here and POSIX is unintentional. This technical report defers to POSIX.

3 The `fmemopen` function associates the buffer given by the `buf` and `size` arguments with a stream. The `buf` argument is either a null pointer or points to a

3) The `open_wmemstream` function is defined in `<wchar.h>`.

buffer that is at least `size` bytes long.

4 The `mode` argument is a character string having one of the following values:

r	Open text stream for reading.
w	Open text stream for writing.
a	Append; open text stream for writing at the first null byte.
r+	Open text stream for update (reading and writing).
w+	Open text stream for update (reading and writing). Truncate the buffer contents.
a+	Append; open text stream for update (reading and writing); the initial position is at the first null byte.
rb	Open binary stream for reading.
wb	Open binary stream for writing.
ab	Append; open binary stream for writing at the first null byte.
rb+ or *r+b*	Open binary stream for update (reading and writing).
wb+ or *w+b*	Open binary stream for update (reading and writing). Truncate the buffer contents.
ab+ or *a+b*	Append; open binary stream for update (reading and writing); the initial position is at the first null byte.

5 If a null pointer is specified as the `buf` argument, `fmemopen` allocates `size` bytes of memory as if by a call to `malloc`. This buffer shall be automatically freed when the stream is closed. Because this feature is only useful when the stream is opened for updating (because there is no way to get a pointer to the buffer) the `fmemopen` call may fail if the `mode` argument does not include a `+` when `buf` is a null pointer.

6 The stream maintains a current position in the buffer. This position is initially set to either the beginning of the buffer (for *r* and *w* modes) or to the first null byte in the buffer (for *a* modes). If no null byte is found in append mode, the initial position is set to one byte after the end of the buffer.

7 If `buf` is a null pointer, the initial position shall always be set to the beginning of the buffer.

8 The stream also maintains the size of the current buffer contents. For modes *r* and *r+* the size is set to the value given by the `size` argument. For modes *w* and *w+* the initial size is zero and for modes *a* and *a+* the initial size is either the position of the first null byte in the buffer or the value of the size argument if no null byte is found.

9 A read operation on the stream cannot advance the current buffer position beyond the current buffer size. Reaching the buffer size in a read operation counts as "end of file". Null bytes in the buffer have no special meaning for reads. The read operation starts at the current buffer position of the stream.

10 A write operation starts either at the current position of the stream (if mode has not specified *a* as the first character) or at the current size of the stream (if mode had *a* as the first character). If the current position at the end of the write is larger than the current buffer size, the current buffer size is set to the current position. A write operation on the stream cannot advance the current buffer size beyond the size given in the size argument.

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

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

13 Note that when writing to a text stream, line endings may occupy more than one character in the buffer.

Returns

14 The `fmemopen` function returns a pointer to the object controlling the stream. If the open operation fails, `fmemopen` returns a null pointer.

Examples

```
#define __STDC_WANT_LIB_EXT2__ 1
#include <stdio.h>
#include <string.h>

static char buffer[] = "foobar";

int
main (void)
{
    int ch;
    FILE *stream;

    stream = fmemopen(buffer, strlen (buffer), "r");
    if (stream == NULL)
        /* handle error */;

    while ((ch = fgetc(stream)) != EOF)
        printf("Got %c\n", ch);

    fclose(stream);
    return (0);
}
```

15 This program produces the following output:

```
Got f
Got o
Got o
Got b
Got a
Got r
```

5.2.2.2 The `open_memstream` function

Synopsis

1

```
#define __STDC_WANT_LIB_EXT2__ 1
#include <stdio.h>

FILE * open_memstream(char ** restrict bufp,
     size_t * restrict sizep);
```

Description

2 This interface is derived from POSIX. Any conflict between the requirements described here and POSIX is unintentional. This technical report defers to POSIX.

3 The `open_memstream` function creates a byte-oriented stream that is associated with a dynamically allocated buffer. The buffer is obtained as if by calls to `malloc` and `realloc` and expanded as necessary. The buffer should be freed by the caller after successfully closing the stream, by means of a call to `free`. The stream is opened for writing and is seekable.

4 The stream maintains a current position in the allocated buffer and a current buffer length. The position is initially set to zero (the beginning of the buffer). Each write starts at the current position and moves this position by the number of successfully written bytes. The length is initially set to zero. If a write moves the position to a value larger than the current length, the current length is set to this position. In this case a null byte is appended to the current buffer, but not accounted for in the buffer length.

5 After a successful `fflush` the pointer referenced by `bufp` and the variable referenced by `sizep` remain valid only until the next write operation on the stream or a call to `fclose`.

Returns

6 The `open_memstream` function returns a pointer to the object controlling the stream. If the open operation fails, `open_memstream` returns a null pointer.

Examples

```
#include <stdio.h>
int main (void)
{
    FILE *stream;
    char *buf;
    size_t len;

    stream = open_memstream(&buf, &len);

    if (stream == NULL)
        /* handle error */;

    fprintf(stream, "hello my world");
    fflush(stream);
    printf("buf=%s, len=%zu\n", buf, len);
    fseek(stream, 0, SEEK_SET);
    fprintf(stream, "good-bye cruel world");
    fclose(stream);
    printf("buf=%s, len=%zu\n", buf, len);
    free(buf);
    return 0;
}
```

7 This program produces the following output:

```
buf=hello my world, len=14
buf=good-bye cruel world, len=20
```

5.2.3 Formatted input/output functions

5.2.3.1 The `asprintf` function

Synopsis

1

```
#define __STDC_WANT_LIB_EXT2__
#include <stdio.h>
int asprintf(char ** restrict ptr,
     const char * restrict format, ...);
```

Description

2 This interface is derived from LSB. Any conflict between the requirements described here and LSB is unintentional. This technical report defers to LSB.

3 The `asprintf` function behaves as `sprintf`, except that the output string is dynamically allocated space, allocated as if by a call to `malloc`, of sufficient length to hold the resulting string. The address of this dynamically allocated string is stored in the location referenced by `ptr`. This dynamically allocated string should be freed by the caller by means of a call to `free` when the contents are no longer required.

5.2.3.2 The `vasprintf` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT2__ 1
#include <stdarg.h>
#include <stdio.h>
int vasprintf(char ** restrict ptr,
     const char * restrict format, va_list arg);
```

Description

2 The `vasprintf` function is equivalent to `asprintf`, with the variable argument list replaced by `arg`, which shall have been initialized by the `va_start` macro (and possibly subsequent `va_arg` calls). The `vasprintf` function does not invoke the `va_end` macro.

5.2.3.3 The `fscanf` function

Description

1 This interface is derived from POSIX. Any conflict between the requirements described here and POSIX is unintentional. This technical report defers to POSIX.

2 In addition to the requirements in ISO/IEC 9899:1999 clause 7.19.6.2, the `fscanf` function shall support the following requirements for conversion specifications.

3 For the string conversion specifiers `c`, `s` and `[`, there may be an optional *assignment-allocation* character, `m`, that appears in the sequence of characters that follow the `%` after any field width and before any length modifier. In this case, the receiving argument should be of type `char **`, or `wchar_t **` if the length modifier is `l`, and shall receive a pointer to a dynamically allocated buffer, allocated as if by a call to `malloc`, that contains the converted string. The string is always null terminated. If there was insufficient memory to allocate a buffer, the receiving argument receives a pointer to a null value. The buffer should be freed by the caller by means of a call to `free` when the contents are no longer required.

4 If `fscanf` returns `EOF`, any memory successfully allocated for parameters using the assignment-allocation character `m` for this call shall be freed.

5.2.4 Character input/output functions

5.2.4.1 The `getdelim` function

Synopsis

1
```
    #define __STDC_WANT_LIB_EXT2__ 1
    #include <stdio.h>
    ssize_t getdelim(char ** restrict lineptr,
         size_t * restrict n,
         int delimiter, FILE * stream);
```

Description

2 This interface is derived from POSIX. Any conflict between the requirements described here and POSIX is unintentional. This technical report defers to POSIX.

3 The `getdelim` function reads from *stream* until it encounters a character matching the `delimiter` character. The argument *delimiter* (when converted to an `unsigned char`) specifies the character that terminates the input text.

4 The `delimiter` argument is an `int`, the value of which should be a character representable as an `unsigned char` or equal to the macro `EOF`. If the `delimiter` argument has any other value, the behavior is undefined.

5 The value of `*lineptr` should be a valid argument that could be passed to the `free` function. If `*n` is nonzero, `*lineptr` should point to an object containing at least `*n` characters.

6 The size of the object pointed to by `*lineptr` is increased to fit the incoming line, if it isn't already large enough. The characters read are stored in the string pointed to by the argument `lineptr`.[4)]

Returns

7 The `getdelim` function returns the number of characters written into the buffer, including the delimiter character if one was encountered before `EOF`. If a read error occurs, the error indicator for the stream is set and `getdelim` returns –1.

5.2.4.2 The `getline` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT2__ 1
#include <stdio.h>
ssize_t getline(char ** lineptr, size_t * n,
     FILE * stream);
```

Description

2 This interface is derived from POSIX. Any conflict between the requirements described here and POSIX is unintentional. This technical report defers to POSIX.

3 The `getline` function is equivalent to the `getdelim` function with the `delimiter` character equal to the newline character.

4) Setting `*lineptr` to a null pointer and `*n` to zero are allowed and are a recommended way to start parsing a file.

5.3 String handling `<string.h>`

5.3.1 Copying functions

5.3.1.1 The `strdup` function

Synopsis

1
```
    #define __STDC_WANT_LIB_EXT2__ 1
    #include <string.h>
    char * strdup(const char * str1);
```

Description

2 This interface is derived from POSIX. Any conflict between the requirements described here and POSIX is unintentional. This technical report defers to POSIX. The `strdup` function shall return a pointer to a new string, which is a duplicate of the string pointed to by `s1`. The returned pointer can be passed to `free`.

Returns

The `strdup` function returns a pointer to the newly allocated string. A null pointer is returned if the new string cannot be created.

5.3.1.2 The `strndup` function

Synopsis

1
```
    #define __STDC_WANT_LIB_EXT2__ 1
    #include <string.h>
    char * strndup(const char * string, size_t n);
```

Description

2 This interface is derived from POSIX. Any conflict between the requirements described here and POSIX is unintentional. This technical report defers to POSIX.

3 The `strndup` function is equivalent to the `strdup` function, duplicating the provided `string` in a new block of memory allocated as if by using `malloc`, with the exception being that `strndup` copies at most `n` plus one bytes into the

newly allocated memory, terminating the new string with a null byte. If the length of `string` is larger than `n`, only `n` bytes shall be duplicated. If `n` is larger than the length of `string`, all bytes in `string` shall be copied into the new memory buffer, including the terminating null byte. The newly created string shall always be properly terminated.

Returns

4 The `strndup` function returns a pointer to the allocated string, or a null pointer if there was insufficient space. The allocated space should be subsequently freed by a call to `free`.

5.4 Extended multibyte and wide character utilities `<wchar.h>`

5.4.1 Operations on buffers

5.4.1.1 The `open_wmemstream` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT2__ 1
#include <wchar.h>
FILE *open_wmemstream(wchar_t ** bufp, size_t * sizep);
```

Description

2 This interface is derived from POSIX. Any conflict between the requirements described here and POSIX is unintentional. This technical report defers to POSIX.

3 The `open_wmemstream` function creates a wide oriented stream that is associated with a dynamically allocated buffer[5]. The buffer is obtained as if by calls to `malloc` and `realloc` and expanded as necessary. The buffer should be freed by the caller after successfully closing the stream, by means of a call to `free`. The stream is opened for writing and is seekable.

4 The stream maintains a current position in the allocated buffer and a current buffer length. The position is initially set to zero (the beginning of the buffer). Each write starts at the current position and moves this position by the number of successfully written wide characters. The length is initially set to zero. If a write moves the position to a value larger than the current length, the current length is set to this position. In this case a null wide character is appended to the current buffer, but not accounted for in the buffer length.

5 After a successful `fflush` the pointer referenced by `bufp` and the variable referenced by `sizep` remain valid only until the next write operation on the stream or a call to `fclose`.

Returns

6 Upon successful completion, `open_wmemstream` returns a pointer to the object controlling the stream. If the open operation fails, `open_wmemstream` returns a

5) Memory buffer based streams are described in <stdio.h>.

null pointer.

5.4.2 Formatted wide character input/output functions

5.4.2.1 The `aswprintf` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT2__
#include <stdio.h>
#include <wchar.h>
int aswprintf(wchar_t ** restrict ptr,
     const wchar_t * restrict format, ...);
```

Description

2 The `aswprintf` function behaves as `swprintf`, except that the output string is dynamically allocated space, allocated as if by a call to `malloc`, of sufficient length to hold the resulting string. The address of this dynamically allocated string is stored in the location referenced by `ptr`. This dynamically allocated string should be freed by the caller by means of a call to `free` when the contents are no longer required.

5.4.2.2 The `vaswprintf` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT2__ 1
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>
int vaswprintf(wchar_t ** restrict ptr,
     const wchar_t * restrict format, va_list arg);
```

Description

2 The `vaswprintf` function is equivalent to `aswprintf`, with the variable argument list replaced by `arg`, which shall have been initialized by the `va_start` macro (and possibly subsequent `va_arg` calls). The `vaswprintf`

function does not invoke the `va_end` macro.

5.4.2.3 The `fwscanf` function

Description

1 In addition to the requirements

2 This interface is derived from POSIX. Any conflict between the requirements described here and POSIX is unintentional. This technical report defers to POSIX.

3 In addition to the requirements in ISO/IEC 9899:1999 clause 7.24.2.2, the `fwscanf` function shall support the following requirements for conversion specifications.

4 For the string conversion specifiers `c`, `s` and `[`, there may be an optional *assignment-allocation* character, `m`, that appears in the sequence of characters that follow the `%` after any field width and before any length modifier. In this case, the receiving argument should be of type `char **`, or `wchar_t **` if the length modifier is `l`, and shall receive a pointer to a dynamically allocated buffer, allocated as if by a call to `malloc`, that contains the converted string. If the `l` length modifier is not specified, the corresponding argument should be of type `char **`, and shall receive a pointer to a dynamically allocated buffer containing characters from the input field, converted as if by repeated calls to the `wcrtomb` function, with the conversion state described by an `mbstate_t` object initialized to zero before the first wide character is converted.

5 In either case, the string shall always be null terminated. If there was insufficient memory to allocate a buffer, the receiving argument shall receive a pointer to a null value.

6 If `fwscanf` returns `EOF`, any memory successfully allocated for parameters using the assignment-allocation character `m` for this call shall be freed.

5.4.3 Wide character input/output functions

5.4.3.1 The `getwdelim` function

Synopsis

```
#define __STDC_WANT_LIB_EXT2__ 1
#include <stdio.h>
ssize_t getwdelim(wchar_t ** restrict lineptr,
     size_t * restrict n,
     wint_t delimiter, FILE * stream);
```

Description

1 The `getwdelim` function shall read from *stream* until it encounters a wide character matching the `delimiter` character. The argument `delimiter` shall specify the character that terminates the read process.

2 The `delimiter` argument is a `wint_t`, the value of which should be a wide character representable as an `wchar_t` or equal value to the macro `WEOF`. If the `delimiter` argument has any other value, the behavior is undefined.

3 The value of `*lineptr` should be a valid argument that could be passed to the `free` function. If `*n` is nonzero, `*lineptr` should point to an object containing at least `*n` wide characters.

4 The size of the object pointed to by `*lineptr` shall be increased to fit the incoming line, if it isn't already large enough. The wide characters read shall be stored in the string pointed to by the argument `lineptr`.[6)]

Returns

5 Upon successful completion the `getwdelim` function shall return the number of wide characters written into the buffer, including the delimiter character if one was encountered before end of file. Otherwise it shall return –1.

5.4.3.2 The `getwline` function

Synopsis

1
```
#define __STDC_WANT_LIB_EXT2__ 1
#include <stdio.h>
ssize_t getwline(wchar_t ** lineptr, size_t * n,
     FILE * stream);
```

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

Description

2 The `getwline` function shall be equivalent to the `getwdelim` function with the `delimiter` character equal to the wide newline character.

Annex A
(informative)

Comparison Of Library Methods

A.5 Introduction

1 This annex presents a small program to illustrate the differences between the approaches of ISO/IEC 9899:1999, the bounds checking functions presented in Part 1 of this Technical Report, and the allocating functions of this part of this Technical Report.

A.5.1 Standard C

1 This example uses only interfaces present in ISO/IEC 9899:1999.

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

void
get_y_or_n(void)
{
     char response[8];

     printf("Continue? [y] n: ");
     gets(response);
     if (response[0] == ’n’)
          exit(0);
     return;
}
```

2 This program has undefined behavior if more than 8 characters are entered at the prompt.

A.5.2 Bounds Checking

1 This example uses the interfaces described in part 1 of this Technical Report.

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>
#include <stdlib.h>

void
get_y_or_n(void)
{
     char response[8];
     size_t len = sizeof(response);

     printf("Continue? [y] n: ");
     gets_s(response, len);
     if (response[0] == ’n’)
          exit(0);
     return;
}
```

2 This program is very similar to to the original ISO/IEC 9899:1999 example, except that the array bounds are checked. There is implementation defined behavior (typically `abort`) if more than 8 characters are input.

3 This program can be improved to remove the implementation defined behavior at the cost of additional complexity[7]:

7) This program has implementation defined behavior in the presence of multiple threads; however, ISO/IEC 9899:1999 does not specify any behavior for multi-threaded programs.

```
#define __STDC_WANT_LIB_EXT1__ 1

#include <stdio.h>
#include <stdlib.h>

void
get_y_or_n(void)
{
     char response[8];
     size_t len = sizeof(response);
     constraint_handler_t oconstraint;

     oconstraint = set_constraint_handler_s(ignore_handler_s);
     printf("Continue? [y] n: ");
     if((gets_s(response, len) == NULL) || (response[0] == ’n’))  {
          (void) set_constraint_handler_s(oconstraint);
          exit(0);
     }
     (void) set_constraint_handler_s(oconstraint);
     return;
}
```

A.5.3 Dynamic Memory

1 This program uses the interfaces decribed in this part of this Technical Report.

```
#define __STDC_WANT_LIB_EXT2__ 1
#include <stdio.h>
#include <stdlib.h>

void
get_y_or_n(void)
{
     char *response = NULL;
     size_t len;

     printf("Continue? [y] n: ");
     if((getline(&response, &len, stdin) < 0) ||
        (len && response[0] == ’n’)) {
          if(response)
               free(response);
          exit(0);
     }
     free(response);
     return;
}
```

2 This program has defined behavior for any input. This includes the assumption that an extremely long line that exhausts all available memory to hold it should be treated as if it were a "no" response. While it requires calls to `free` to release any allocated memory, it is almost as simple as the original ISO/IEC 9899:1999 example.

Index