-- --

IEEE P1003.2 Draft 11.2 – September 1991

A.3.7.2 Conflicts

The parser produced for an input grammar may contain states in which conflicts occur. The conflicts occur because the grammar is not LALR(1). An ambiguous grammar always contains at least one LALR(1) conflict. The `yacc` utility shall resolve all conflicts, using either default rules or user-specified precedence rules.

Conflicts are either ''shift/reduce conflicts'' or ''reduce/reduce conflicts.'' A shift/reduce conflict is where, for a given state and lookahead symbol, both a shift action and a reduce action are possible. A reduce/reduce conflict is where, for a given state and lookahead symbol, reductions by two different rules are possible.

The rules below describe how to specify what actions to take when a conflict occurs. Not all shift/reduce conflicts can be successfully resolved this way because the conflict may be due to something other than ambiguity, so incautious use of these facilities can cause the language accepted by the parser to be much different than was intended. The description file shall contain sufficient information to understand the cause of the conflict. Where ambiguity is the reason either the default or explicit rules should be adequate to produce a working parser.

The declared precedences and associativities (see A.3.7.1.2) are used to resolve parsing conflicts as follows:

(1) A precedence and associativity is associated with each grammar rule; it is the precedence and associativity of the last token or literal in the body of the rule. If the `%prec` keyword is used, it overrides this default. Some grammar rules might not have both precedence and associativity.

(2) If there is a shift/reduce conflict, and both the grammar rule and the input symbol have precedence and associativity associated with them, then the conflict is resolved in favor of the action (shift or reduce) associated with the higher precedence. If the precedences are the same, then the associativity is used; left associative implies reduce, right associative implies shift, and nonassociative implies an error in the string being parsed.

(3) When there is a shift/reduce conflict that cannot be resolved by rule (2), the shift is done. Conflicts resolved this way are counted in the diagnostic output described in A.3.7.3.

(4) When there is a reduce/reduce conflict, a reduction is done by the grammar rule that occurs earlier in the input sequence. Conflicts resolved this way are counted in the diagnostic output described in A.3.7.3.

Conflicts resolved by precedence or associativity shall not be counted in the shift/reduce and reduce/reduce conflicts reported by `yacc` on either standard error or in the description file.

A.3.7.3 Error Handling

The token `error` shall be reserved for error handling. The name `error` can be used in grammar rules. It indicates places where the parser can recover from a syntax error. The default value of `error` shall be 256. Its value can be changed using a `%token` declaration. The lexical analyzer should not return the value of `error`.

The parser shall detect a syntax error when it is in a state where the action associated with the lookahead symbol is `error`. A semantic action can cause the parser to initiate error handling by executing the macro YYERROR. When YYERROR is executed, the semantic action shall pass control back to the parser. YYERROR cannot be used outside of semantic actions.

When the parser detects a syntax error, it normally calls `yyerror` with the character string `"syntax error"` as its argument. The call shall not be made if the parser is still recovering from a previous error when the error is detected. The parser is considered to be recovering from a previous error until the parser has shifted over at least three normal input symbols since the last error was detected or a semantic action has executed the macro `yyerrok`. The parser shall not call `yyerror` when YYERROR is executed.

The macro function YYRECOVERING () shall return 1 if a syntax error has been detected and the parser has not yet fully recovered from it. Otherwise, zero shall be returned.

When a syntax error is detected by the parser, the parser shall check if a previous syntax error has been detected. If a previous error was detected, and if no normal input symbols have been shifted since the preceding error was detected, the parser checks if the lookahead symbol is an endmarker (see A.3.7.4). If it is, the parser shall return with a nonzero value. Otherwise, the lookahead symbol shall be discarded and normal parsing shall resume.

When YYERROR is executed or when the parser detects a syntax error and no previous error has been detected, or at least one normal input symbol has been shifted since the previous error was detected, the parser shall pop back one state at a time until the parse stack is empty or the current state allows a shift over `error`. If the parser empties the parse stack, it shall return with a nonzero value. Otherwise, it shall shift over `error` and then resume normal parsing. If the parser reads a lookahead symbol before the error was detected, that symbol shall still be the lookahead symbol when parsing is resumed.

The macro `yyerrok` in a semantic action shall cause the parser to act as if it has fully recovered from any previous errors. The macro `yyclearin` shall cause the parser to discard the current lookahead token. If the current lookahead token has not yet been read, `yyclearin` shall have no effect.

The macro YYACCEPT shall cause the parser to return with the value zero. The macro YYABORT shall cause the parser to return with a nonzero value.

A.3.7.4 Interface to the Lexical Analyzer

The *yylex* () function is an integer-valued function that returns a *token number* representing the kind of token read. If there is a value associated with the token returned by *yylex* () (see the discussion of *tag* above), it shall be assigned to the external variable *yylval* .

If the parser and *yylex* () do not agree on these token numbers, reliable communication between them cannot occur. For (one character) literals, the token is simply the numeric value of the character in the current character set. The numbers for other tokens can either be chosen by `yacc`, or chosen by the user. In either case, the `#define` construct of C is used to allow *yylex* () to return these numbers symbolically. The `#define` statements are put into the code file, and the header file if that file is requested. The set of characters permitted by `yacc` in an identifier is larger than that permitted by C. Token names found to contain such characters shall not be included in the `#define` declarations.

If the token numbers are chosen by `yacc`, the tokens other than literals shall be assigned numbers greater than 256, although no order is implied. A token can be 1
explicitly assigned a number by following its first appearance in the declarations section with a number. Names and literals not defined this way retain their default definition. All assigned token numbers shall be unique and distinct from the token numbers used for literals. If duplicate token numbers cause conflicts in parser generation, `yacc` shall report an error; otherwise, it is unspecified whether the token assignment is accepted or an error is reported.

The end of the input is marked by a special token called the *endmarker* , which has a token number that is zero or negative. (These values are invalid for any other token.) All lexical analyzers shall return zero or negative as a token number upon reaching the end of their input. If the tokens up to, but excluding, the endmarker form a structure that matches the start symbol, the parser shall accept the input. If the endmarker is seen in any other context, it shall be considered an error.

A.3.7.5 Completing the Program

In addition to *yyparse* () and *yylex* (), the functions *yyerror* () and *main* () are required to make a complete program. The application can supply *main* () and *yyerror* (), or those routines can be obtained from the `yacc` library.

A.3.7.6 `yacc` Library

The following functions appear only in the `yacc` library accessible through the `-l y` operand to `c89`; they can therefore be redefined by a portable application:

`int main(void)` 1

This function shall call *yyparse* () and exit with an unspecified value. Other actions within this function are unspecified.

`int yyerror(const char *`*s*`)` 1

This function shall write the NUL-terminated argument to standard error, followed by a `<newline>`.

The order of the `-l y` and `-l l` operands given to `c89` is significant; the application shall either provide its own *main* () function or ensure that `-l y` precedes `-l l`.

A.3.7.7 Debugging the Parser

The parser generated by `yacc` shall have diagnostic facilities in it that can be optionally enabled at either compile time or at run time (if enabled at compile time). The compilation of the runtime debugging code is under the control of YYDEBUG , a preprocessor symbol. If YYDEBUG has a nonzero value, the debugging code shall be included. If its value is zero, the code shall not be included.

In parsers where the debugging code has been included, the external `int yydebug` can be used to turn debugging on (with a nonzero value) and off (zero value) at run time. The initial value of *yydebug* shall be zero.

When `-t` is specified, the code file shall be built such that, if YYDEBUG is not already defined at compilation time (using the `c89 -D YYDEBUG` option, for example), YYDEBUG shall be set explicitly to 1. When `-t` is not specified, the code file shall be built such that, if YYDEBUG is not already defined, it shall be set explicitly to zero.

The format of the debugging output is unspecified but includes at least enough information to determine the shift and reduce actions, and the input symbols. It also provides information about error recovery.

A.3.7.8 Algorithms

The parser constructed by `yacc` implements an LALR (1) parsing algorithm as documented in the literature. It is unspecified whether the parser is table-driven or direct-coded.

A parser generated by `yacc` shall never request an input symbol from *yylex* () while in a state where the only actions other than the error action are reductions by a single rule.

The literature of parsing theory defines these concepts.

A.3.7.9 Limits

The `yacc` utility may have several internal tables. The minimum maximums for these tables are shown in Table A-4. The exact meaning of these values is implementation defined. The implementation shall define the relationship between these values and between them and any error messages that the implementation may generate should it run out of space for any internal structure. An implementation may combine groups of these resources into a single pool as long as the total available to the user does not fall below the sum of the sizes specified by this subclause.

Table A-4 – yacc Internal Limits

Limit	Minimum Maximum	Description
{NTERMS }	126	Number of tokens.
{NNONTERM }	200	Number of nonterminals.
{NPROD }	300	Number of rules.
{NSTATES }	600	Number of states.
{MEMSIZE }	5200	Length of rules. The total length, in names (tokens and nonterminals), of all the rules of the grammar. The left-hand side is counted for each rule, even if it is not explicitly repeated, as specified in A.3.7.1.3.
{ACTSIZE }	4000	Number of actions. ''Actions'' here (and in the description file) refer to parser actions (shift, reduce, etc.) not to semantic actions defined in A.3.7.1.3.

A.3.8 Exit Status

The `yacc` utility shall exit with one of the following values:

- 0 Successful completion.
- >0 An error occurred.

A.3.9 Consequences of Errors

If any errors are encountered, the run is aborted and `yacc` exits with a nonzero status. Partial code files and header files files may be produced. The summary information in the description file shall always be produced if the `−v` flag is present.

A.3.10 Rationale. *(This subclause is not a part of P1003.2)*

The references in the Bibliography may be helpful in constructing the parser generator. The Pennello-DeRemer {B26} paper (along with the works it references) 2
describe a technique to generate parsers that conform to this standard. Work in this area continues to be done, so implementors should consult current literature before doing any new implementations. The original paper by Knuth {B27} is the 1
theoretical basis for this kind of parser, but the tables it generates are impractically large for reasonable grammars, and should not be used. The ''equivalent to'' wording is intentional to assure that the best tables that are LALR(1) can be generated.

There has been confusion between the class of grammars, the algorithms needed to generate parsers, and the algorithms needed to parse the languages. They are all reasonably orthogonal. In particular, a parser generator that accepts the full range of LR(1) grammars need not generate a table any more complex than one that accepts SLR(1) (a relatively weak class of LR grammars) for a grammar that happens to be SLR(1). Such an implementation need not recognize the case, either; table compression can yield the SLR(1) table (or one even smaller than that) without recognizing that the grammar is SLR(1). The speed of a LR(1) parser for any class is dependent more upon the table representation and compression (or the code generation if a direct parser is generated) than upon the class of grammar that the table generator handles.

The speed of the parser generator is somewhat dependent upon the class of gram-
mar it handles. However, the original Knuth {B27} algorithms for constructing 2
LR parsers was judged by its author to be impractically slow at that time. 2
Although full LR is more complex than LALR(1), as computer speeds and algo-
rithms improve, the difference (in terms of acceptable wall-clock execution time) is 2
becoming less significant. 2

Potential authors are cautioned that the Penello-DeRemer paper previously cited 2
identifies a bug (an oversimplification of the computation of LALR(1) lookahead 2
sets) in some of the LALR(1) algorithm statements that preceded it to publication. 2
They should take the time to seek out that paper, as well as current relevant 2
work, particularly Aho's {B22}.

Examples, Usage

Access to the `yacc` library is obtained with library search operands to `c89`. To use the `yacc` library *main* (),

```
c89 y.tab.c -l y
```

Both the `lex` library and the `yacc` library contain *main* (). To access the `yacc` *main* (),

```
c89 y.tab.c lex.yy.c -l y -l l
```

This ensures that the `yacc` library is searched first, so that its *main* () is used.

The historical `yacc` libraries have contained two simple functions that are normally coded by the application programmer. These library functions are similar to the following code:

```
#include <locale.h>
int main(void)
{
        extern int yyparse();

        setlocale(LC_ALL, "");

        /* If the following parser is one created by lex, the
           application must be careful to ensure that LC_CTYPE
           and LC_COLLATE are set to the POSIX Locale.  */
        (void) yyparse();
        return (0);
}

#include <stdio.h>

int yyerror(const char *msg)
{
        (void) fprintf(stderr, "%s\n", msg);
        return (0);
}
```

Historical implementations experience name conflicts on the names yacc.tmp, yacc.acts, yacc.debug, y.tab.c, y.tab.h, and y.output if more than one copy of yacc is running in a single directory at one time. The −b option was added to overcome this problem. The related problem of allowing multiple yacc parsers to be placed in the same file was addressed by adding a −p option to override the previously hardcoded yy variable prefix. (The −p option name was selected from a historical implementation.) Implementations will also have to be cognizant of 2.11.6.3, which requires that any temporary files used by yacc also be named to avoid collisions.

The description of the −p option specifies the minimal set of function and variable names that cause conflict when multiple parsers are linked together. YYSTYPE does not need to be changed. Instead, the programmer can use −b to give the header files for different parsers different names, and then the file with the *yylex* () for a given parser can include the header for that parser. Names such as *yyclearerr* don't need to be changed because they are used only in the actions; they do not have linkage. It is possible that an implementation will have other names, either internal ones for implementing things such as *yyclearerr* , or providing non-standard features, that it wants to change with −p.

The −b option was added to provide a portable method for permitting yacc to work on multiple separate parsers in the same directory. If a directory contains more than one yacc grammar, and both grammars are constructed at the same time (by, say, a parallel make program), conflict results. While the solution is not historical practice, it corrects a known deficiency in historical implementations. Corresponding changes were made to all sections that referenced the filenames y.tab.c (now ''the code file''), y.tab.h (now ''the header file''), and y.output (now ''the description file'').

The grammar for yacc input is based on System V documentation. The textual description shows there that the ; is required at the end of the rule. The grammar and the implementation do not require this. (The use of C_IDENTIFIER causes a reduce to occur in the right place.)

Also, in that implementation, the constructs such as %token can be terminated by a semicolon, but this is not permitted by the grammar. The keywords such as %token can also appear in uppercase, which is again not discussed. In most places where % is used, \ can be substituted, and there are alternate spellings for some of the symbols (e.g. %LEFT can be %< or even \<).

Multibyte characters should be recognized by the lexical analyzer and returned as tokens. They should not be returned as multibyte character literals. The token error that is used for error recovery is normally assigned the value 256 in the historical implementation. Thus, the token value 256, which used in many multibyte character sets, is not available for use as the value of a user-defined token.

Historically, <*tag*> can contain any characters except >, including white space, in the implementation. However, since the *tag* must reference a Standard C union member, in practice conforming implementations need only support the set of characters for Standard C identifiers in this context.

Some historical implementations are known to accept actions that are terminated by a period. Historical implementations often allow $ in names. A conforming implementation need support neither of these behaviors.

Unary operators that are the same token as a binary operator in general need their precedence adjusted. This is handled by the %prec advisory symbol associated with the particular grammar rule defining that unary operator. See A. Applications are not required to use this operator for unary operators, but the grammars that do not require it are rare.

Deciding when to use %prec illustrates the difficulty in specifying the behavior of yacc. There may be situations in which the *grammar* is not strictly speaking in error, and yet yacc cannot interpret it unambiguously. The resolution of ambiguities in the grammar can in many instances be resolved by providing additional information, such as using %type or %union declarations. It is often easier and it usually yields a smaller parser to take this alternative when it is appropriate.

The size and execution time of a program produced without the runtime debugging code is usually smaller and slightly faster in historical implementations.

There is a fair amount of material in this that appears tutorial in nature; some of it has been moved to the Rationale in Draft 9 to simplify the specification. It is hard to avoid because of the need to define terms at least informally. The alternative is to bring in one of the parser generator texts and use its terminology directly, but since there is some variation in that terminology, it was felt that informal definitions of the terms so that someone who understood the concepts would be sure to understand the terms would make the standard stand alone from any specific text.

Statistics messages from several historical implementations include the following types of information:

n/512 terminals, *n*/300 nonterminals
n/600 grammar rules, *n*/1500 states
n shift/reduce, *n* reduce/reduce conflicts reported
n/350 working sets used
memory: states,etc. *n*/15000, parser *n*/15000
n/600 distinct lookahead sets
n extra closures
n shift entries, *n* exceptions
n goto entries
n entries saved by goto default
Optimizer space used: input *n*/15000, output *n*/15000
n table entries, *n* zero
maximum spread: *n*, maximum offset: *n*

The report of internal tables in the description file is left implementation defined 2
because all aspects of these limits are also implementation defined. Some imple- 2
mentations may use dynamic allocation techniques and have no specific limit 2
values to report. 2

History of Decisions Made

The format of the `y.output` file is not given because specification of the format was not seen to enhance application portability. The listing is primarily intended to help human users understand and debug the parser; use of `y.output` by a portable application script is far-fetched. Furthermore, implementations have not produced consistent output and no clear winner was apparent. The format selected by the implementation should be human-readable, in addition to the requirement that it be a text file.

Standard error reports are not specifically described because they are seldom of use to portable applications and there was no reason to restrict implementations.

Some implementations recognize `={` as equivalent to `{`, because it appears in historical documentation. This construction was recognized and documented as obsolete as long ago as 1978, in the original paper *Yacc: Yet Another Compiler-Compiler* by Stephen C. Johnson. POSIX.2 chose to leave it as obsolete and omit it.

Annex B
(normative)
C Language Bindings Option

This annex describes the C language bindings to the language-independent services described in Section 7.

The interfaces described in this annex may be provided by the conforming system; however, any system claiming conformance to the Language-Independent System Services C Language Bindings Option shall provide all of the interfaces described here.

B.0.1 C Language Bindings Option Rationale. *(This subclause is not a part of P1003.2)*

In this version of POSIX.2, the language-independent descriptions in Section 7 have not been developed. The language-independent syntax is being created in parallel by the POSIX.1 working group. Therefore, the C language bindings described in this annex are actually the full functional specifications. It is the intention of the POSIX.2 working group to rectify this situation in a revision to this standard, by moving the majority of the functional specifications back into Section 7, leaving Annex B with only brief descriptions of the C bindings to those services.

B.1 C Language Definitions

B.1.1 POSIX Symbols

Certain symbols in this annex are defined in headers. Some of those headers could also define symbols other than those defined by this standard, potentially conflicting with symbols used by the application. Also, this standard defines symbols that other standards do not permit to appear in those headers without some control on the visibility of those symbols.

Symbols called *feature test macros* are used to control the visibility of symbols that might be included in a header. Implementations, future versions of this standard, and other standards may define additional feature test macros. The `#define`s for feature test macros shall appear in the application source code before any `#include` of a header where a symbol should be visible to some, but not all, applications. If the definition of the macro does not precede the `#include`, the result is undefined.

Feature test macros shall begin with the underscore character (_) and an upper- 1
case letter, or with two underscore characters. 1

Implementations may add symbols to the headers shown in Table B-1, provided 1
the identifiers for those symbols begin with the corresponding reserved prefixes in 1
Table B-1. Similarly, implementations may add symbols to the headers in 1
Table B-1 that end in the string indicated as a reserved suffix as long as the 1
reserved suffix is in that part of the name considered significant by the implemen- 1
tation. This shall be in addition to any reservations made in the C Standard {7}. 1

After the last inclusion of a given header, an application may use any of the sym- 1
bol classes reserved in Table B-1 for its own purposes, as long as the requirements 1
in the note to Table B-1 are satisfied, noting that the symbol declared in the 1
header may become inaccessible. 1

Future revisions of this standard, and other POSIX standards, are likely to use 1
symbols in these same reserved spaces. 1

In addition, implementations may add members to a structure or union without 1
controlling the visibility of those members with a feature test macro, as long as a 1
user-defined macro with the same name cannot interfere with the correct 1
interpretation of the program. 1

A conforming POSIX.2 application shall define the feature test macro in Table B-2. When an application includes a header and the _POSIX_C_SOURCE feature test macro is defined to be the value 1 or 2, the effect shall be the same as if _POSIX_SOURCE was defined as described in POSIX.1 {8}.

In addition, when the application includes any of the headers defined in this stan- 1
dard, and _POSIX_C_SOURCE is defined to be the value 2: 1

(1) All symbols defined in POSIX.2 to appear when the header is included
shall be made visible. 1

Table B-1 – POSIX.2 Reserved Header Symbols

Header	Key	Reserved Prefix	Reserved Suffix
`<fnmatch.h>`	2	`FNM_`	
`<glob.h>`	1	`gl_`	
	2	`GLOB_`	
`<limits.h>`	1		`_MAX`
`<regex.h>`	1	`re_`	
	1	`rm_`	
	2	`REG_`	
`<wordexp.h>`	1	`we_`	
	2	`WRDE_`	

NOTE: The Key values are:

(1) Prefixes and suffixes of symbols that shall not be declared or `#define`d by the application.

(2) Prefixes and suffixes of symbols that shall be preceded in the application with a `#undef` of that symbol before any other use.

Table B-2 – _POSIX_C_SOURCE

Name	Description
_POSIX_C_SOURCE	Enable POSIX.1 {8} and POSIX.2 symbols; see text.

(2) Symbols that are explicitly permitted, but not required, by POSIX.2 to appear in the header (including those in reserved name spaces) may be made visible.

(3) Additional symbols shall not be made visible, unless controlled by another feature test macro.

The effect of defining the _POSIX_C_SOURCE macro to any other value is unspecified.

If there are no feature test macros present in a program, only the set of symbols defined by the C Standard {7} shall be present. For each feature test macro present, only the symbols specified by that feature test macro plus those of the C Standard {7} shall be defined when the header is included.

B.1.1.1 POSIX Symbols Rationale. *(This subclause is not a part of P1003.2)*

When the application defines the _POSIX_C_SOURCE feature test macro with 1
value 2, it must be aware that all of the name space from POSIX.1 {8} and POSIX.2 1
has been reserved. This does not imply that a POSIX.2 implementation must support POSIX.1 {8}, just that the application must not conflict with an implementation that does. The application can check _POSIX_VERSION and 1
_POSIX2_C_VERSION at compile time to see which standards are supported, if 1
that is necessary. This is primarily an issue for the headers <stdio.h>, <limits.h>, <locale.h>, and <unistd.h>, since other POSIX.1 {8} names appear in other headers not mentioned in POSIX.2.

It is expected that C bindings to future POSIX standards and revisions will define
new values for _POSIX_C_SOURCE, with each new value reserving the name 1
space for that new standard or revision, plus all earlier POSIX standards. Using a single feature test macro for all standards rather than a separate macro for each standard furthers the goal of eventually combining all of the C bindings into one standard, which will be included in an international standard that refers to a language-independent ISO/IEC 9945-1 {8}.

B.1.2 Headers and Function Prototypes

Implementations shall declare function prototypes for all functions. Each function prototype shall appear in the header included in the synopsis of the function.

B.1.3 Error Numbers

Some of the functions in this annex use the variable *errno* to report errors. Such usage is documented in Errors in each specification. The usage of *errno* and the meanings of the symbolic names shall be as defined in POSIX.1 {8} B.1.3.

B.1.4 C Language Definitions Rationale. *(This subclause is not a part of P1003.2)*

This clause clarifies the interface to the C Standard {7}. The description was
taken from POSIX.1, with one important modification. Since POSIX.1 {8} and the 1
C Standard {7} were being developed and approved at about the same time, 1
POSIX.1 {8} allowed ‘‘Common Usage C’’ implementations to give system vendors 1
time to develop Standard C interfaces. Since Standard C compilers are now com- 1
monly available, POSIX.2 does not explicitly describe the binding to Common 1
Usage C. However, such a binding would be straightforward, as long as the rules for Common Usage C in POSIX.1 are followed.

B.2 C Numerical Limits

The following subclauses list the names of macros that C language applications can use to obtain minimum and current values for limits defined in 2.13.1.

B.2.0.1 C Numerical Limits Rationale. *(This subclause is not a part of P1003.2)*

This subclause was added in Draft 9 to give C applications access to limits at compile time. Applications can use the values from the macros without resorting to *sysconf* (). The descriptions very closely follow the descriptions of macros and limits in POSIX.1 {8}.

This definition of the limits is specific to the C language. Other language bindings might use different interfaces or names to provide equivalent information to the application.

Note that there are no C bindings or interfaces that change based on the macros in Table B-5. These macro only advertise the availability of the associated utilities.

B.2.1 C Macros for Symbolic Limits

The macros in Table B-3 shall be defined in the header `<limits.h>`. They specify values for the symbolic limits defined in 2.13.1.

Table B-3 – C Macros for Symbolic Limits

Symbolic Limit	Minimum Allowed by POSIX.2	Minimum for this Implementation
{BC_BASE_MAX}	_POSIX2_BC_BASE_MAX	BC_BASE_MAX
{BC_DIM_MAX}	_POSIX2_BC_DIM_MAX	BC_DIM_MAX
{BC_SCALE_MAX}	_POSIX2_BC_SCALE_MAX	BC_SCALE_MAX
{BC_STRING_MAX}	_POSIX2_BC_STRING_MAX	BC_STRING_MAX
{COLL_WEIGHTS_MAX}	_POSIX2_COLL_WEIGHTS_MAX	COLL_WEIGHTS_MAX
{EXPR_NEST_MAX}	_POSIX2_EXPR_NEST_MAX	EXPR_NEST_MAX
{LINE_MAX}	_POSIX2_LINE_MAX	LINE_MAX
{RE_DUP_MAX}	_POSIX2_RE_DUP_MAX	RE_DUP_MAX

The names in the first column of Table B-3 are symbolic limits as defined in 2.13.1. The names in the second column are C macros that define the smallest values permitted for the symbolic limits on any POSIX.2 implementation; they shall be defined as constant expressions with the most restrictive values specified in 2.13.1. The names in the third column are C macros that define less restrictive values provided by the implementation; each shall be defined as a constant that

— is not smaller than the associated macro in column 2, and

— is not larger than the smallest value that will be returned by *sysconf* () when the application is executed.

B.2.1.1 C Macros for Symbolic Limits Rationale. *(This subclause is not a part of P1003.2)*

The macros in column 3 of Table B-3 are required to be constant expressions.

If the C binding is to be used with POSIX.2 implementations over which the implementor of the binding has no control, the column-3 values must be the same as column-2. If the implementation of the C binding is intended to be used with a POSIX.2 implementation that always supports a larger value than one in column 2, that implementation of the binding may use the larger value for the column-3 macro. If an application compiled with that binding is then used with a different POSIX.2 implementation, it is the user's fault that the application is being run in an environment in which it was not intended.

The application can assume, for example, that the stream created by `popen("mailx user","w")` will accept lines of length {LINE_MAX}, even if this is larger than {_POSIX2_LINE_MAX}. However, if the application is creating a data file that might be processed on another implementation, it should use the values in column 2.

B.2.2 Compile-Time Symbolic Constants for Portability Specifications

The macros in Table B-4 shall be defined in the header `<unistd.h>`. These macros can be used by the application, at compile time, to determine which optional facilities are present and what actions shall be taken by the implementation.

Table B-4 – C Compile-Time Symbolic Constants

Macro Name	Description	
_POSIX2_C_VERSION	The integer value 199???L. This value indicates the version of the inter- faces in this annex that are provided by the implementation. It will change with each published version of this standard to indicate the 4-digit year and 2-digit month that the standard was approved by the IEEE Stan- dards Board.	1 1 1 1 1

B.2.2.1 Compile-Time Symbolic Constants for Portability Specifications Rationale. *(This subclause is not a part of P1003.2)*

The symbolic constant _POSIX2_C_VERSION is analogous to _POSIX_VERSION, defined in POSIX.1 {8}. It indicates the version of the C interfaces that are supplied by the compiler and runtime library. 1

The version of the utilities is given by the {POSIX2_VERSION} limit (see 2.13.1), whose value can be obtained at runtime using *sysconf*() (see B.10.2). 1

B.2.3 Execution-Time Symbolic Constants for Portability Specifications

The macros in Table B-5 can be used by the application at execution time to determine which optional facilities are present. If a macro is defined to have the value −1 in the header `<unistd.h>`, the implementation shall never provide that feature when the application runs under that implementation. If a macro is defined to have a value other than −1, the implementation shall always provide that feature. If the macro is undefined, then the *sysconf* () function (see B.10.2) can be used to determine if the feature is provided for a particular invocation of the application.

Table B-5 − C Execution-Time Symbolic Constants

Macro Name	Description
_POSIX 2_C_DEV	The system supports the C Language Development Utilities Option (see Annex A)
_POSIX 2_FORT _DEV	The system supports the FORTRAN Development Utilities Option (see Annex C)
_POSIX 2_FORT _RUN	The system supports the FORTRAN Runtime Utilities Option (see Annex C)
_POSIX 2_LOCALEDEF	The system supports the creation of locales as described in 4.35.
_POSIX 2_SW _DEV	The system supports the Software Development Utilities Option (see Section 6)

B.2.4 POSIX. 1 C Numerical Limits

The macros specified in POSIX .1 {8} to provide compile-time values for the configurable variables in Table 7-1 (see 7.8.2) shall also be visible in a POSIX. 2 system. Other macros required by POSIX .1 {8} 2.9 (Numerical Limits) and 2.10 (Symbolic Constants) may also be visible in a POSIX. 2 system.

B.2.4.1 POSIX. 1 C Numerical Limits Rationale. *(This subclause is not a part of P1003.2)*

Subclause 7.8.2 requires that certain POSIX .1 {8} configurable variables be visible in POSIX. 2. Subclause B.2.4 ensures that POSIX. 2 C applications can obtain these variables using the same macros as POSIX .1 {8} C applications. It also allows an implementation to make all of the POSIX .1 {8} macros available even if _POSIX _SOURCE is not set. It also allows an implementation to make all of the POSIX .1 {8} symbols available even if it does not support all of POSIX .1 {8}.

B.3 C Binding for Shell Command Interface

B.3.0.1 C Binding for Shell Command Interface Rationale. *(This subclause is not a part of P1003.2)*

The *system* () and *popen* () functions should not be used by programs that have set user (or group) ID privileges, as defined in POSIX.1 {8}. The *fork* () and *exec* family of functions [except *execlp* () and *execvp* ()], also defined in POSIX.1 {8}, should be used instead. This prevents any unforeseen manipulation of the user's environment that could cause execution of commands not anticipated by the calling program.

If the original and '*popen* ()ed'' processes both intend to read or write or read and write a common file, and either will be using FILE-type C functions [*fread* (), *fwrite* (), etc.], the rules in POSIX.1 {8}8.2.3 must be observed.

B.3.1 C Binding for Execute Command

Function: *system* ()

B.3.1.1 Synopsis

```
#include <stdlib.h>
int system(const char *command );
```

B.3.1.2 Description

This standard requires the *system* () function as described in the C Standard {7}.

The *system* () function shall execute the command specified by the string pointed to by *command*. The environment of the executed command shall be as if a child process were created using the POSIX.1 {8} *fork* () function, and the child process invoked the sh utility (see 4.56) using the POSIX.1 {8} *execl* () function as follows:

```
execl(<shell path>, "sh", "-c", command , (char *)0);
```

where *<shell path>* is an unspecified pathname for the sh utility.

The *system* () function shall ignore the SIGINT and SIGQUIT signals, and block the SIGCHLD signal, while waiting for the command to terminate. If this might cause the application to miss a signal that would have killed it, then the application should examine the return value from *system* () and take whatever action is appropriate to the application if the command terminated due to receipt of a signal.

The *system* () function shall not affect the termination status of any child of the calling processes other than the process(es) it itself creates.

The *system* () function shall not return until the child process has terminated.

B.3.1.3 Returns

If *command* is **NULL**, the *system* () function shall return nonzero.

If *command* is not **NULL**, the *system* () function shall return the termination status of the command language interpreter in the format specified by the *waitpid* () function in POSIX.1 {8}. The termination status of the command language interpreter is as specified for the `sh` utility, except that if some error prevents the command language interpreter from executing after the child process is created, the return value from *system* () shall be as if the command language interpreter had terminated using *exit* (127) or *_exit* (127). If a child process cannot be created, or if the termination status for the command language interpreter cannot be obtained, *system* () shall return −1 and set *errno* to indicate the error.

B.3.1.4 Errors

The *system* () function may set *errno* values as described by *fork* () in POSIX.1 {8}.

B.3.1.5 Rationale. *(This subclause is not a part of P1003.2)*

The C Standard {7} specifies that when *command* is **NULL**, *system* () returns nonzero if there is a command interpreter available and zero if one is not available. At first reading, it might appear that POSIX.2 conflicts with this, since it requires *system* (**NULL**) to always return nonzero. There is no conflict, however. A POSIX.2 implementation must always have a command interpreter available, and is nonconforming if none is present. It is therefore permissible for the *system* () function on a POSIX.2 system to implement the behavior specified by the C Standard {7} as long as it is understood that the implementation is not POSIX.2 conforming if *system* (**NULL**) returns zero. 1

Note that, while *system* () must ignore SIGINT and SIGQUIT and block SIGCHLD while waiting for the child to terminate, the handling of signals in the executed command is as specified by *fork* () and *exec*. For example, if SIGINT is being caught or is set to SIG_DFL when *system* () is called, then the child will be started with SIGINT handling set to SIG_DFL.

Ignoring SIGINT and SIGQUIT in the parent process prevents coordination problems (two processes reading from the same terminal, for example) when the executed command ignores or catches one of the signals. It is also usually the correct action when the user has given a command to the application to be executed synchronously (as in the ‘‘!’’ command in many interactive applications). In either case, the signal should be delivered only to the child process, not to the application itself. There is one situation where ignoring the signals might have less than the desired effect. This is when the application uses *system* () to perform some task invisible to the user. If the user typed the interrupt character (`^C` for example) while *system* () is being used in this way, one would expect the application to be killed, but only the executed command will be killed. Applications that use *system* () in this way should carefully check the return status from *system* () to see if the executed command was successful, and should take appropriate action when the command fails.

Blocking SIGCHLD while waiting for the child to terminate prevents the application from catching the signal and obtaining status from *system* ()'s child process before *system* () can get the status itself.

Examples, Usage

The context in which the utility is ultimately executed may differ from that in which the *system* () function was called. For example, file descriptors that have the FD_CLOEXEC flag set will be closed, and the process ID and parent process ID will be different. Also, if the executed utility changes its environment variables or its current working directory, that change will not be reflected in the caller's context.

Earlier drafts of this standard required, or allowed, *system* () to return with *errno* [EINTR] if it was interrupted with a signal. This error return was removed, and a requirement that *system* () not return until the child has terminated was added. This means that if a *waitpid* () call in *system* () exits with *errno* [EINTR], *system* () must re-issue the *waitpid* (). This change was made for two reasons:

(1) There is no way for an application to clean up if *system* () returns [EINTR], short of calling *wait* (), and that could have the undesirable effect of returning status of children other than the one started by *system* ().

(2) While it might require a change in some historical implementations, those implementations already have to be changed because they use *wait* () instead of *waitpid* ().

Note that if the application is catching SIGCHLD signals, it will receive such a sig- 1
nal before a successful *system* () call returns. 1

History of Decisions Made

The C Standard {7} requires that a call to *system* () with a **NULL** will return a nonzero value, indicating the presence of a command language interpreter available to the system. It was explicitly decided that when *command* is **NULL** , *system* () should not be required to check to make sure that the command language interpreter actually exists with the correct mode, that there are enough processes to execute it, etc. The call *system* (**NULL**) could, theoretically, check for such problems as too many existing child processes, and return zero. However, it would be inappropriate to return zero due to such a (presumably) transient condition. If some condition exists that is not under the control of this application and that would cause *any system* () call to fail, that system has been rendered nonconformant.

Modified in Draft 6 to reflect the availability of the *waitpid* () function in POSIX .1 {8}. To conform to this standard, *system* () must use *waitpid* (), or some similar function, instead of *wait* ().

Figure B-1 illustrates how *system* () might be implemented on a POSIX .1 {8} implementation.