Section 1: Overview

[Jon expressed a concern about all the references to the C standard. I really don't want to copy in large chunks of the C standard, especially regarding expression evaluation. Anybody have any ideas?]

[I used the ANSI C standard in drafting this paper. The ISO C standard added some sections which causes a mismatch of ANSI C and ISO C section numbers. During later processing of this paper, I will track down all section references to the C standard to make sure they correspond to the ISO section numbering.]

1.1 Scope

ISO/IEC 1539 is a multi-part International Standard; the parts are published separately. The first part of the standard, 1539-1, specifies the form and establishes the interpretation of programs expressed in the Fortran language. The second part, 1539-2, defines additional facilities for the manipulation of character strings of variable length. This publication, 1539-3, which is the third part, specifies the form and establishes the interpretation of source processing directives that may be interspersed in Fortran source program text. The purpose of this part is to promote portability, reliability, and maintainability of Fortran programs for use on a variety of computing systems. A processor conforming to 1539-1 need not conform to 1539-3.

Throughout this publication, the term "this standard" refers to 1539-3 and the term "the C standard" refers to ISO/IEC 9899.

1.2 Processor

The combination of a computing system and the mechanism by which source text is transformed for use on that computing system is called a **processor** in this standard. In particular, the processor described by this standard is called a **source processor**.

1.3 Inclusions

This standard specifies

(1) The forms that source processing directives may take,
(2) The rules for interpreting the meaning of a source processing directive,
(3) The form of the input to the source processor, and
(4) The form of the output from the source processor.

1.4 Exclusions

This standard does not specify

(1) The mechanism by which source text is transformed,
(2) The operations required for setup and control of the source processor on computing systems,
(3) The method of transfer of source text to or from a storage medium,
(4) The processor behavior and resultant output (if any) when this standard fails to establish an interpretation,

(5) The size or complexity of Fortran source lines and/or source processing directives that will exceed the capacity of any specific computing system or the capability of a particular source processor,

(6) The physical properties of the representation of quantities and the method of rounding, approximating, or computing numeric values on a particular processor,

(7) The physical properties of the source text input file, or

(8) The physical properties and implementation of storage.

1.5 Conformance

The source text input to the source processor is **standard-conforming Fortran source text** if the Fortran source lines (2.1), source processing directive lines, and macro invocations have an interpretation according to this standard.

A processor conforms to this standard if

(1) It processes the source processing directives and macro invocations contained in any standard-conforming Fortran source text file in a manner that fulfills the interpretations herein, subject to any limits that the processor may impose on the size and complexity of the source text file, the directives, or the macro invocations;

(2) It contains the capability to detect and report the use within a submitted source processing directive or macro invocation of an additional form or relationship that is not permitted by the numbered syntax rules or their associated constraints;

(3) It contains the capability to detect and report the use within a source processing directive of an intrinsic procedure whose name is not defined in this standard or in Section 13 of 1539-1 (conformance to Section 13 of 1539-1 is only required when the processor is interpreting expressions according to 1539-1); and

(4) It contains the capability to detect and report the reason for rejecting a submitted Fortran source text file.

A standard-conforming processor may allow additional forms and relationships provided that such additions do not conflict with the standard forms and relationships. However, a standard-conforming processor may allow additional intrinsic procedures within source processing directives. A standard-conforming source processing directive shall not use nonstandard intrinsic procedures that have been added by the processor.

Because a standard-conforming source processing directive (particularly a macro expansion) may place demands on a processor that are not within the scope of this standard, conformance to this standard does not ensure that the source text input file will be processed consistently on all or any standard-conforming source processors.

In some cases, this standard allows the provision of facilities that are not completely specified in the standard. These facilities are identified as **processor dependent**, and they shall be provided, with methods or semantics determined by the processor.

NOTE 1.1

The processor should be accompanied by documentation that specifies the limits it imposes on the size and complexity of a directive, macro expansion, or source text input file and the means of reporting when these limits are exceeded, that defines the additional forms and relationships it allows, and that defines the means of reporting the use of additional forms and relationships.

The processor should be accompanied by documentation that specifies the methods or semantics of processor-dependent facilities.

1.5.1 Standard C source preprocessing compatibility

[This is where differences between Fortran standards are described. I will either describe the differences between the C source preprocessor and the one described in this document in an Annex or in each place in the document where a particular feature is described if there is sufficient call for noting the differences. For example, the C standard allows a name to be simply an underscore. I contend this hardly promotes maintainability and thus require a name beginning with an underscore to contain at least one other character. }

1.6 Notation used in this standard

In this standard, "shall" is to be interpreted as a requirement; conversely, "shall not" is to be interpreted as a prohibition. Such requirements and prohibitions apply to the source processing directives or to the processor as noted in this standard.

1.6.1 Informative notes

Informative notes of explanation, rationale, examples, and other material are interspersed with the normative body of this publication. The informative material is identified by shading and is non-normative.

1.6.2 Syntax rules

Syntax rules are used to help describe the forms that the input source text file, and source processing lexical tokens, directives, and constructs may take. These syntax rules are expressed in a variation of Backus-Naur form (BNF) in which:

(1) Characters from the Fortran character set (1539-1, 3.1, and 3.1 of this standard) are interpreted literally as shown, except where otherwise noted.

(2) Lower-case italicized letters and words (often hyphenated and abbreviated) represent general syntactic classes for which specific syntactic entities shall be substituted in actual statements.

An example of such an abbreviation used in a syntactic terms is:

arg	for	argument

(3) The syntactic metasymbols used are:

is	introduces a syntactic class definition
or	introduces a syntactic class alternative
[ ]	encloses an optional item
[ ] ...	encloses an optionally repeated item which may occur zero or more times
■	continues a syntax rule

(4) Each syntax rule is given a unique identifying number of the form R*snn*, where *s* is a one- or two-digit section number and *nn* is a two-digit sequence number within that section. The syntax rules are distributed as appropriate throughout the text, and are referenced by number as needed. In some cases, a rule may be used in a given section but is more fully described in a later section; in such cases, the section number *s* is the number of the later section where the rule is defined.

NOTE 1.2

An example of the use of the syntax rules is:

digit-string	**is**	*digit* [*digit*] ...

The following are examples of forms for a digit string allowed by the above rule:

digit
digit digit
digit digit digit digit
digit digit digit digit digit digit digit digit

When a specific number character is substituted for *digit*, actual digit strings might be:

```
4
67
1999
10243852
```

1.6.3 Assumed syntax rules

In order to minimize the number of additional syntax rules and convey appropriate constraint information, the following rules are assumed. The letters "*xyz*" stand for any legal syntactic class phrase:

xyz-list	**is**	*xyz* [, *xyz*] ...
xyz-name	**is**	*name*

1.6.4 Syntax conventions and characteristics

(1) Any syntactic class name ending in "*-directive*" follows the source processing directive form rules. Conversely, everything considered to be a source processing directive is given a "*-directive*" ending in the syntax rules.

(2) Expression hierarchy is described rigorously in the definition of *constant-expression* (C standard, 3.4) and in section 7 of 1539-1 (applicable only if expressions are being evaluated according to 1539-1).

1.6.5 Text conventions

In the descriptive text, an English word equivalent of a BNF syntactic term is usually used. Specific directive keywords are identified in the text by the lower-case keyword in a distinctive font, e.g., "`#define` directive". Boldface words are used in the text where they are first defined with a specialized meaning. Any feature of the source processor that references an obsolescent feature of 1539-1 uses the same distinguishing type size as used in 1539-1.

NOTE 1.3

This sentence is an example of the size used for obsolescent features.

1.7 Normative references

The following standards contain provisions which, through reference in this standard, constitute provisions of this standard. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based upon this standard are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below. Members of IEC and ISO maintain registers of currently valid International Standards.

ISO/IEC 9899-1990 (E) Programming Language - C

ISO/IEC 9899:1990/Amendment 1:1994, Amendment 1: C Integrity

ISO/IEC 9899:1990/Technical Corrigendum 1

ISO/IEC 9899:1990/Technical Corrigendum 2

ISO/IEC 646:1991, Information processing—ISO 7-bit coded character set for information interchange.

Section 2: Fortran source processing terms and concepts

The source processor can conditionally process and skip sections of the submitted source text, include other source text, and replace macros. These capabilities are sometimes called **source preprocessing**, because conceptually they often occur before the source text is submitted to the Fortran language processor.

2.1 Conceptual model

A **Fortran source line** is defined to be a Fortran line as defined by 1539-1, 3.3. In particular, this means a Fortran source line is a sequence of zero or more characters which may constitute Fortran statements, a comment, or an INCLUDE line. In addition, this standard allows a Fortran source line to contain source processing macro references. However, during processing of the source text input file, all such macros shall be transformed into Fortran source text such that at the end of source processing no macro references remain in any Fortran source lines.

The source processor permits the Fortran source lines to appear in free source form or fixed source form as defined in 1539-1, 3.3. As in 1539-1, free form and fixed form source lines shall not be mixed in the same source text input file. The means for specifying the source form of a source text input file are processor dependent.

A **directive line** is a line that begins with the # character as defined by the numbered syntax rules and associated constraints in this standard.

A **line** is defined to be either a Fortran source line or a directive line. The two types of lines may be distinguished by the presence of the leading # character. A Fortran source line does not start with this character.

A **file** is defined to be a collection of Fortran source lines and source processing directive lines where the number of either or both of these lines may be zero. That is, a file may be empty. The Fortran source lines are assumed to conform to the numbered syntax rules and associated constraints of 1539-1 (with the possible addition of macro references) but there is no requirement that they conform to 1539-1.

The model used in this standard to conceptually describe the actions of source processing (and thus the conceptual actions of a source processor) is that the source text to be operated upon is a collection of Fortran source lines and directive lines contained in a file. This file is serves as the input to the source processor. The source processor then interprets any directives and macro references contained in the file and outputs another file containing zero or more Fortran source lines, and possibly `#line` directives (**[#line directive]**). There is no requirement, however, that the input to the source processor be physically contained in a file and there is likewise no requirement that the output of the source processor be physically contained in a file. There is no requirement that the Fortran source lines output by the source processor conform to the numbered syntax rules and associated constraints of 1539-1 or that the collection of Fortran source lines output by the source processor constitute a standard-conforming Fortran program as defined by 1539-1. Furthermore, there is no requirement that the output of the source processor serves as input, directly or indirectly, to a Fortran language processor. And finally, there is no requirement that a Fortran source processor be physically part of a Fortran language processor (but there are likewise no provisions in this standard which would prohibit packaging a source processor physically with a language processor).

2.2 Input to the source processor

The input to the source processor is a file containing Fortran source lines with interspersed directives. Each source processor directive shall begin with the number sign character (#). This character shall appear as the first significant character in a line; that is, only whitespace (x,y) may precede this character. If the # character is followed by a directive keyword, only whitespace may appear between the # character and the keyword.

Blanks within a source processing directive are significant. A source processing directive is unaffected by the source form of the Fortran source lines in the source text input file. The maximum allowed length of a source processing directive is processor dependent.

2.3 Output from the source processor

The output of the source processor is a file containing zero or more Fortran source lines and zero or more `#line` directives. The lines output are dependent on any source processing directives, in particular `#if` directives, contained in the input file. Source processing directive lines other than `#line` directive lines shall not be contained in the output file.

The source processor shall not modify the text of a Fortran comment. In particular, this means that if a Fortran comment contains a set of characters which would otherwise constitute a macro reference, that set of characters is not processed as a macro reference. The set of characters are regarded as comment characters.

The expansion of a macro or a string replacement may cause the column width of a Fortran source line to exceed column 132 (for free form) or 72 (for fixed form). By default, the source processor shall output the line as generated. It may optionally generate continuation lines as needed and an appropriate continuation symbol for each continuation line (using the appropriate form as defined for continuation lines in 1539-1). The method of directing the source processor to generate continuation lines is processor dependent.

NOTE 2.1

The intent is that by default the source processor just outputs the line at whatever length is produced by the macro expansion. If the output from the source processor is later provided as input to a Fortran language processor and if the resultant line length exceeds the maximum specified by 1539-1 for the source form of the file input to the Fortran language processor, it is assumed that the Fortran language processor will issue a diagnostic message. It is also the intent that the source processor provide a command line option that specifies that continuation lines are to be generated when a macro expansion causes the output line to otherwise exceed the maximum length for the Fortran source form.

[QUESTION: One reviewer suggested that the command line optionality in this case not be provided. He prefers that the source processor always generate the continuation lines or always issue a diagnostic if the line exceeds the max for the source form. I think that users are going to want to be able to control whether or not they want the continuation lines generated. There could, of course, be a third way to handle this: the source processor issue a diagnostic if the line exceeds the max length for the source form. This would be another command line option. So then we'd have: (default) just output the line as generated, (command line opt 1) if the line exceeds the max length for the source form, break it into continuation lines, (command line opt 2) if the line exceeds the max length for the source form, issue a diagnostic. Opinions?]

Section 3: Characters, lexical tokens, and source form

3.1 Processor character set

The processor character set is as described in 1539-1, 3.1, with the addition of the characters in the basic source and execution character sets, escape sequences, and null character as defined in 2.2.1 of the C standard.

Rxxx *character* **is** *alphanumeric-character*
or *special-character*

Rxxx *alphanumeric-character* **is** *letter*
or *digit*
or *underscore*

3.1.1 Letters

The letters are the 26 upper-case letters as described by 1539-1, with the addition that the source processor allows lower-case letters. Within a Fortran source line, the lower-case letters are equivalent to the corresponding upper-case letters except in a character context (1539-1, 3.3) or in a macro reference. Within source processing directives and macros (including macro references contained in Fortran source lines), lower-case letters are distinguished from the corresponding upper-case letters.

The set of upper-case and lower-case letters defines the syntactic class *letter.*

3.1.2 Digits

The 10 digits are as described in 1539-1, 3.1.2 and define the syntactic class *digit*.

3.1.3 Underscore

Rxxx *underscore* **is** _

The underscore may be used as a significant character in a source processing name, including the first character of a source processing name.

3.1.4 Special characters

The special characters are as defined in 1539-1, 3.1.4, with the addition of the characters called "graphic characters" in 2.2.1 of the C standard except for the vertical tab and form feed characters. Together, these two sets of characters define the syntactic class *special-character.*

3.1.5 Additional characters

Additional characters may be representable in the processor, but may appear only in comments (both Fortran and source processing), character constants (both Fortran and source processing), and character string edit descriptors.

The source processor does not support nondefault character types in source processing directives and macros.

3.2 Low-level syntax

The **low-level syntax** describes the fundamental lexical tokens of the source text input file. The term **lexical tokens** may refer to both Fortran language lexical tokens and the tokens making up a source processing directive or macro reference. When necessary, the tokens may be identified as being Fortran language tokens or source processing tokens. Lexical tokens in general are defined by 1539-1, 3.2.

3.2.1 Names

Names are used for various entities such as source processing variables and macros. The rule for the formation of a name differs from 1539-1 in that a source processing name may start with an underscore character.

Rxxx *name* **is** *letter* [*alphanumeric-character*] ...
or *underscore alphanumeric-character* ■
■ [*alphanumeric-char*] ...

NOTE 3.1

Although a name may begin with an underscore, generally the use of an underscore as the first character is discouraged because it may conflict with a name predefined by the implementation. In fact, the C standard has the concept of reserved identifiers (4.1.2.1 - need ISO section ref). It states, in particular, that all identifiers that begin with an underscore and either an uppercase letter or another underscore are always reserved for any use. Thus, if the Fortran program is communicating with any C routines, the use of identifiers beginning with underscores in the Fortran program could cause conflicts with names reserved by the vendor of the C product.

3.2.2 Constants

If the source processor is evaluating expressions in Fortran expression evaluation mode (x,y), all literal constants in directives shall have the forms of Fortran literal constants as described in 1539-1, 3.2.2 and shall be evaluated as specified by 1539-1. If the source processor is evaluating expressions in C expression evaluation mode, the forms of constants allowed in directives are shall have the forms of C literal constants as described in 6.1.3 of the C standard and shall be evaluated as specified by the C standard.

3.2.3 Operators

If the source processor is in Fortran expression evaluation mode, the operators supported are as described in 1539-1, 3.2.3, syntactic class *intrinsic-operator*. No defined operator shall appear in any expression in any directive.

NOTE 3.2

"Fortran evaluation mode" means that the conditional expressions of source processing *if-construct*s are logical expressions, not integer expressions. It also allows the use of Fortran intrinsic procedures in the expressions (note, however, that the expressions are initialization expressions).

If the source processor is in C expression evaluation mode, the operators that are supported is the following subset of the operators described in 6.1.5 of the C standard:

`<, >, ==, !=, >=, <=, +, -, /, *, %, <<, >>, &, ~, |, &&, ||`

3.2.4 Whitespace

Whitespace within a source processing directive is defined to be a sequence of one or more blanks or horizontal tabs (also called simply "tabs") between source processing tokens. For the purposes of this definition, the # character beginning a directive and the directive keyword are also both considered to be source processing tokens. Additionally, a source processing comment may appear anywhere the white space may appear.

3.3 Source form

3.3.1 Source processing commentary

The delimiter "/*" initiates a **source processing comment** except when the character sequence appears in a character context. The comment extends to the first subsequent "*/" delimiter. The comment start character sequence "/*" within a comment has no effect. The ending delimiter of the source processing comment is not required to be on the same line as the beginning delimiter; that is, a source processing comment may continue across multiple lines.

Lines containing only blanks or containing no characters are also comment lines.

A source processing comment may appear within a source processing directive anywhere whitespace (x.y) may appear. A Fortran source line shall not contain a source processing comment. Source processing comments may precede the first Fortran source line or directive line in the source text input file and may follow the last Fortran source line or directive line.

Comments have no effect on the interpretation of any source processing directives or the Fortran source lines in the source text input file.

Source processing comments shall not be output by the source processor. For purposes of evaluating directive lines, the source processor shall replace each source processing comment with a single blank character.

NOTE 3.3

In other words, a source processing comment can appear before the # that begins the directive because the comment is first replaced by a single blank. Thus,

```
/* This is a comment. */  #define  SMALL  0
```

becomes

```
   #define SMALL 0
```

and

```
# /* SMALL */ define  SMALL  0
```

becomes

```
#   define  SMALL  0
```

3.3.2 Source processing directive continuation

The character "\" with no following characters is used to indicate that the current source processing directive is continued on the next line. The continuation takes effect even when the "\" (with no characters following it on the line) appears in a character context. When used for continuation, the "\" is not part of the directive. Source processing comment lines can be continued across multiple lines without the use of the continuation character; that is, a "\" within a comment has no effect.

As described in 2.2.1 of the C standard, the "\" may also be used to initiate an escape sequence inside a character string literal. This use of the "\" does not constitute a continuation (because it is not the last character on the line).

Section 4: Input and directive forms

4.1 High level syntax

Rxxx	*source-text*	**is**	[*source-text-part*] ...

Rxxx	*source-text-part*	**is**	*Fortran-source-line*
		or	*directive-line*
		or	*if-construct*

The rule for *source-text* describes the contents of the input file to the source processor. Section 2.1 of this standard defines the syntactic class *Fortran-source-line*.

4.2 Directive overview

The general term "source processing directive" or simply "directive" includes both *directive-line* and *if-construct*.

A source processing directive consists of a sequence of source processing tokens that begins with a `#` character. The `#` character signals that this line of the source text contains a source processing directive. Since lower-case letters are distinguished from upper-case letters in directive lines, the letters making up a source processing keyword shall all be lower case.

Rxxx	*directive-line*	**is**	*expr-eval-mode-directive*
		or	*macro-def-directive*
		or	*macro-undef-directive*
		or	*include-directive*
		or	*error-directive*
		or	*stop-directive*
		or	*null-directive*
		or	*line-directive*

4.3 Expression evaluation mode

Rxxx	*expr-eval-directive*	**is**	`#fortran_mode`
		or	`#c_mode`

The source processor shall be capable of evaluating expressions both according to 1539-1 (the default expression evaluation mode) and according to the C standard. All expressions shall be evaluated in a single evaluation mode for the entire source text file input to the source processor as well as for all source text included by any `#include` directive (at any level of nesting) encountered while processing the source input file.

The default expression evaluation mode may be confirmed via use of the `#fortran_mode` directive. Expression evaluation mode may be set to the C mode via the `#c_mode` directive. If either directive appears in any of the source text input to the source processor, the only directives that may appear before the expression evaluation mode directive are `#line` directives.

When the source processor is in Fortran expression evaluation mode, all expressions shall have the form of initialization expressions. Each such initialization expression shall be evaluated as described in 1539-1, section 7. No defined operator shall appear in any expression in any directive.

When the source processor is in C expression evaluation mode, the evaluation of expressions supported by the source processor is as described in 6.3 of the C standard.

4.4 Macro definition and replacement

Rxxx *macro-def-directive* **is** *object-like-macro*
or *function-like-macro*

Rxxx *object-like-macro* **is** `#define` *name* [*sp-token*] ...

Rxxx *function-like-macro* **is** `#define` *name*(*dummy-arg-list*) *sp-token* ...

Constraint: The left parenthesis delimiter of the *dummy-arg-list* shall immediately follow the *name* with no intervening whitespace.

Rxxx *dummy-arg* **is** *name*

[Difference: The ANSI doc I used when drafting this paper and C9x as well allow the argument list to be empty. I can see no utility in this (and apparently neither did Sun because their fpp also requires at least one argument) so until I'm beaten up badly on this, I'm requiring at least one argument to be present.]

A name defined as a macro without a parenthesized *dummy-arg-list* is called an **object-like macro.** The name is also termed a **source processing variable**.

A name defined as a macro with a parenthesized *dummy-arg-list* is called a **function-like macro**.

The *name* following `#define` is called the **macro name**.

The collection of source processing tokens (*sp-token*s) following the macro name of an object-like macro or the dummy argument list of a function-like macro is called the **replacement list**.

Any whitespace preceding or following the replacement list is not considered part of the replacement list for either the object-like or function-like macros.

Two replacement lists are identical if and only if the source processing tokens in both have the same number, ordering, spelling, and white-space separation, where all whitespace separations are considered identical.

A macro definition is in effect until a `#undef` directive is encountered which specifies the macro name or (if none is encountered) until the end of the source text input file.

If a `#` character is followed by a sequence of characters that is not the same as any directive keyword (such as `define`) and the # and its following sequence of characters occur on a line where a directive could begin, the sequence of characters (which may appear to be a name) is not subject to macro replacement. Since the line does not follow the form of any directive described in this standard, it is not recognized as a source processing directive and thus is output to the output file.

NOTE 4.1

Output of such a line will probably cause a Fortran language processor to produce an error if the output file is submitted to a Fortran language processor.

A macro name has two characteristics associated with it: a definition state and a value, where the value is optional (the "value" is the replacement list). Thus, a macro name may be defined but have no value (there were no tokens specified in the replacement list or after macro expansion in the replacement list was complete, no tokens remain). The definition state and value of a macro name are of particular relevance to the *if-construct* directives (x.y).

Example:

```
#define  STATUS_VALUE                         /* This macro has no value but */
                                              /* the name is defined.        */
```

```
#define  STATUS       "Value: " STATUS_VALUE
```

If the definitions of `STATUS_VALUE` and `STATUS` are as shown and `STATUS` is referenced elsewhere in the source input file, after macro replacement in the replacement list of `STATUS`, its value `"Value:  "` (unchanged) because the replacement list for `STATUS_VALUE` contains no tokens.

NOTE 4.2

A more typical use of an object-like macro without a replacement list is to use it to determine whether or not Fortran source lines are to be output via an *if-construct*. For this usage, only a definition state is needed; the value of the macro is unimportant (and therefore unnecessary). For example:

```
#define  System_1
...
#ifdef System_1
...
#endif
```

The source processor shall provide a mechanism at processor invocation to initially define or undefine a macro name and shall provide a mechanism to specify an initial integer value for a source processing variable. If no value is specified with the source processing variable name, it is the same as if the name was specified to have the value 1. The mechanism by which the definition state or value are initially specified is processor dependent.

NOTE 4.3

The intent is that the initial definition state or value is provided via command line option. POSIX compliant and at least some other Unix source (pre)processors use the -D to specify the definition state and value, and use the -U option to specify that a macro name is undefined.

If a macro name is specified at processor invocation to be both defined and undefined, or to be defined multiple times, the order of determination of the final definition state and/or value of the macro name is processor dependent.

By default, the source processor only recognizes macro references in other directives. As described earlier in this section. directive keywords are not subject to macro replacement. The source processor shall also be capable of recognizing macro reference within Fortran source lines (outside of comments, IMPLICIT statement *implicit-spec-list*s (1539-1, 5.3), text in FORMAT statements, and string literals). The method for directing the processor to recognize macro references in Fortran source lines is processor dependent.

[Need to explain here in a note why contexts such as the IMPLICIT stmt are ignored. Question: Does "FORMAT statement" also include formats contained directly in I/O statements?]

NOTE 4.4

The intent is that the recognition of macro references in Fortran source lines be specified via a command line option.

The replacement list for a macro name shall replace only *name* tokens in other directive lines (and optionally in Fortran source lines). That is, the sequence of characters in *name* is not recognized in every context where the same sequence of characters occurs. The sequence of characters is only recognized as a macro reference if the sequence of characters constitutes an entire token. Thus, given

```
#define  INT      1
#define  INTEGER  2
```

If macro `INTEGER` is later referenced, the first three characters of `INTEGER` are not replaced by `1` (the value or replacement list for `INT`) because the first three characters do not constitute an entire token.

If the processor has been directed to recognize macro names in Fortran source lines, macro names are likewise only recognized if they correspond to Fortran tokens. A Fortran literal constant containing a kind type parameter value is considered to be a single token.

Example:

Assume the source processor has been directed to recognize macro names in Fortran source lines and suppose the following Fortran source lines are in fixed source form.

```
1.  #define  LARGE  8
2.  #define  CALLF(x)  f(x)
3.  INTEGER( L A R G E )  LARGE_ARRAY(1000)
4.  LARGE _ARRAY = 1_ LARGE
5.  CALL F(10)
6.  X = CALL F(10)
```

In line 3, the integer `8` is substituted for the kind type parameter value in the INTEGER attribute because the blanks in "`L  A  R  G  E`" are insignificant. The character sequence `LARGE` in `LARGE_ARRAY` is not subject to macro substitution because the character sequence does not constitute an entire token; rather, it is a part of the token `LARGE_ARRAY`.

In line 4, again the first character sequence `LARGE` is a part of a name token so no macro substitution is done; the blank between `LARGE` and the remainder of the token is insignificant. A numeric literal constant and its kind type parameter value form a single token, so the second `LARGE` is also not subject to macro substitution.

In line 5, even though the blank is insignificant, `CALL` and `F` constitute two individual tokens so no macro substitution is performed .

In line 6, again the blank between `CALL` and `F` is insignificant but in this case the significant characters constitute a single token since this is not a context where a CALL statement can appear, so macro substitution does take place.

After macro substitution, the four Fortran source lines would appear as follows:

```
INTEGER( 8 )  LARGE_ARRAY(1000)
LARGE _ARRAY = 1_ LARGE
CALL F(10)
X = f(x)
```

A macro name is recognized from the point of definition to the end of the source input file.

4.4.1 Object-like macro

The directive form

`#define` *name* *string*

defines the value of an object-like macro (source processing variable) *name* to be set of source processing tokens contained in *string*. A source processing variable may appear in the conditional expression of a conditional source processing directive and thus participate in the determination of which source text lines are produced by the source processor.

A source processing variable can be explicitly undefined using the `#undef` directive (x.y).

A source processing variable can be implicitly redefined by another `#define` directive provided that the second definition is also an object-like macro and the two replacement lists are identical.

The simplest use of a macro is an object-like macro that defines a constant value, as in

```
#define RELEASE         1
#define System_Name     "System 31"
#define TABLE_SIZE      100
```

If source processing was also being performed on the Fortran source lines then `TABLE_SIZE` could be used to declare the size of an array as in the following:

```
        INTEGER  table(TABLE_SIZE)
```

Since lower-case and upper-case letters are distinguishable when a macro name is being replaced in a Fortran source line, if the Fortran source line was written as

```
        INTEGER  table(Table_Size)
```

the macro replacement does not take place.

4.4.2 Function-like macro

A directive of the form

`#define` *name*`(`*dummy-arg-list*`)` *string*

defines a function-like macro with arguments. The dummy arguments are specified by the parenthesized list of names, whose scope is limited to this function-like macro definition. Each subsequent appearance of the function-like macro name followed by a parenthesized argument list introduces the sequence of tokens that replace the dummy argument list in the macro definition (each subsequent appearance of the macro name is termed an **invocation** or **expansion** of the macro). The replacement sequence of tokens is terminated by the matching right parenthesis, skipping intervening matched pairs of left and right parenthesis characters. As in the definition of a function-like macro, in a function-like macro reference the left parenthesis of the *dummy-arg-list* must immediately follow the macro name with no intervening whitespace.

In a macro invocation, the sequence of tokens bounded by the outermost matching parentheses forms the **actual argument list** of the function-like macro. Commas between inner matching parentheses do not separate the arguments of the list. If (before argument substitution) any argument consists of no tokens, the behavior is undefined. If there are sequences of tokens within the list of actual arguments that would otherwise act as source processing directives, the behavior is undefined.

The sequence of tokens making up an invocation of a function-like macro are not required to all be on the same line. That is, a function-like macro reference may be continued using the source processing continuation character '\'.

A function-like macro may be explicitly undefined using the `#undef` directive.

A function-like macro may be implicitly redefined by another `#define` directive provided that the second definition is also a function-like macro that has the same number and spelling of dummy arguments, and the two replacement lists are identical.

Example: The following defines a function-like macro whose value is the sum of its arguments and illustrates a use of a function-like macro. TABLE_SIZE is the object-like macro defined in the example in **[x.y]**.

```
#define   SUM(arg1, arg2)  arg1 + arg2
#define   INIT_SIZE  SUM(TABLE_SIZE, 100)
```

4.4.2.1 Function-like macro invocation and replacement

The number of actual arguments in an invocation of a function-like macro shall agree with the number of dummy arguments in the macro definition.

In IMPLICIT statement bounds and text in FORMAT statements, macros are not expanded only in case of conflict with valid literals in that context.

[Keith: Please find out what the above paragraph means. From fpp doc.]

After the actual arguments for the invocation of a function-like macro have been identified, argument substitution takes place. A dummy argument in the replacement list, unless preceded by a `#` or `##` operator or followed by a `##` operator (**[The # op]**, **[The ## op]**), is replaced by the corresponding actual argument after all macros contained therein have been expanded. Before being substituted, each actual argument's tokens are completely macro replaced.

[There seems to be some questions as to what the above paragraph is saying. I essentially duped it from the C standard so I don't want to further endanger the text by trying to rewrite it and get it wrong. Any C weenies out there care to translate?]

4.4.2.2 The # operator

Each `#` operator in the replacement list for a function-like macro shall be followed by a dummy argument name as the next source processing token in the replacement list.

For each such `#` operator followed by a dummy argument in the replacement list, both are replaced by a single character string literal that contains the spelling of the source processing token sequence for the corresponding actual argument. Each occurrence of white space between the actual argument's source processing tokens becomes a single space character in the character string literal. White space before the first source processing token and after the last source processing token comprising the actual argument is deleted. Otherwise, the original spelling of each source processing token in the actual argument list is retained in the character string literal, except for special handling for producing the spelling of string literals and character constants: a '\' character is inserted before each '"' and '\' character, including the delimiting '"' characters. If the replacement that results is not a valid character string literal, the behavior is undefined. The order of evaluation of `#` and `##` directive operators is processor dependent.

Example:

```
#define  str(s)  # s
```

An invocation of `str` as `str( 21:   3 )` produces the character string literal `"21: 3"` after the leading and trailing blanks have been deleted and the interior blanks have been replaced with a single blank.

4.4.2.3 The ## operator

A `##` directive operator shall not occur at the beginning or at the end of a replacement list for either form of macro definition.

If a dummy argument name in a replacement list is immediately preceded or followed by a `##` directive operator, the dummy argument name is replaced by the corresponding actual argument's token sequence.

For both object-like and function-like macro invocations, before the replacement list is reexamined for more macro names to replace, each instance of a `##` directive operator in the replacement list (not from an argument) is deleted and the preceding token is concatenated with the following token. If the result is not a valid token, the behavior is undefined. The resulting token is available for further macro replacement. The order of evaluation of `##` directive operators is processor dependent.

[I may add some detail from the C rationale about how this operator works.]

Example:

```
#define  str(s)      # s
```

```
#define  xstr(s)     str(s)
#define  INCFILE(n)  vers ## n
...
#include xstr(INCFILE(2))
```

The result of the invocation of `xstr` is `"vers2"`. The actions are as follows:

(1) The token `2` is the actual argument to `INCFILE` and corresponds to the dummy argument `n`. `INCFILE(2)` is replaced by `vers ## 2`. Before the replacement list `vers ## 2` is reexamined for more macro names to replace, the `##` directive operator is removed and `vers` is concatenated with `2`, producing `vers2`. `vers2` does not contain any more macro names so the replacement for `INCFILE(2)` is complete.

(2) `xstr(vers2)` is then replaced by `str(vers2)`.

(3) `str(vers2)` is replaced by `# vers2` which is in turn replaced by the character string literal `"vers2"`.

The invocation of `xstr(INCFILE(2))` is now complete so the `#include` line is

```
#include "vers2"
```

4.4.3 Rescanning and further replacement

After all dummy argument names in the replacement list have been substituted, the resulting token sequence is rescanned along with all subsequent tokens in the source file for more macro names to replace.

If the name of the macro being replaced is found during this scan of the replacement list (not including the rest of the source text file's source processing tokens), it is not replaced. Further, if any nested replacements encounter the name of the macro being replaced, it is not replaced. These nonreplaced macro name tokens are no longer available for further replacement even if they are later (re)examined in contexts in which that macro name would otherwise have been replaced.

The resulting completely macro-replaced token sequence is not processed as a source processing directive even if it resembles one.

The following example illustrates the rules for redefinition and reexamination.

```
# Macro definitions...
#define x    3
#define f(a)  f(x * (a))
#undef  x
#define x    2
#define g    f
#define z    z(0)
#define t(a) a

! Fortran source code
  q = f(y+1) + f(f(z)) * t(t(g)(0) + t)(1)
```

Evaluation of the macro invocations in the Fortran source code line:

(1) Since the replacement list for a macro is determined dynamically when the macro is referenced rather than statically when the macro is defined, and since in this example `x` was undefined and then redefined with the value 2, the macro replacement for `f(y+1)` is `f(2 * (y+1))`. After these macro replacements, the source code effectively is:

```
q = f(2 * (y+1)) + f(f(z)) * t(t(g)(0) + t)(1)
```

(2) Similarly, the first replacement of `f(f(z))` becomes `f(2 * (f(z))`. The rescan replaces the inner macro invocation so the tokens now become `f(2 * (f(2 * (z)))`. And, finally, macro replacement is done for z producing `f(2 * (f(2 * (z(0)))))`. The Fortran source now effectively appears as:

```
q = f(2 * (y+1)) + f(2 * (f(2 * (z(0))))) * t(t(g)(0) + t)(1)
```

(3) The series of replacements for `t(t(g)(0) + t)` is:

```
t(g)(0) + t

g(0) + t

f(0) + t

f(2 * (0)) + t
```

[I am told my descriptions of the actions of the above macro expansions are not correct even though I arrive at the same final text as does the C standard from which I stole this example. I will get together with people much more knowledgeable than me in C and get the wording right.]

When the replacement is done, the remainder of the Fortran source line appears as `f(2 * (0)) + t(1)` but the `t(1)` is not subject to further macro replacement as described in earlier in this section.

After all replacements, the statement has been transformed to:

```
q = f(2 * (y+1)) + f(2 * (f(2 * (z(0))))) * f(2 * (0)) + t(1)
```

To demonstrate the redefinition rules, the following sequence is valid.

```
#define   OBJ_LIKE       (1-1)
#define   OBJ_LIKE       /* white space */ (1-1) /* more */
#define   FUNC_LIKE(a)   ( a )
#define   FUNC_LIKE( a )(      /* Note the white space */   \
                                a    /* other stuff on this line
                               */   )
```

The following redefinitions are invalid:

`#define`	`OBJ_LIKE`	`(0)`	[different token sequence]
`#define`	`OBJ_LIKE`	`(1 - 1)`	[different white space]
`#define`	`FUNC_LIKE(b)`	`( a )`	[different parameter usage]
`#define`	`FUNC_LIKE(b)`	`( b )`	[different parameter spelling]

4.5 Explicit macro undefinition

Rxxx *macro-undef-directive* **is** `#undef` *name*

An `#undef` directive causes the specified name no longer to be defined as a macro name. This directive is ignored if the specified name is not currently defined as a macro name.

4.6 Conditional source text selection

Rxxx *if-construct* **is** *if-directive*
        source-text
    [*elif-directive*
        source-text] ...
    [*else-directive*
        source-text]
    endif-directive

Rxxx *if-directive* **is** `#if` *constant-expression*
    or `#ifdef` *name*
    or `#ifndef` *name*

Rxxx *elif-directive* **is** `#elif` *constant-expression*

or `#elseif` *constant-expression*
Rxxx *else-directive* **is** `#else`
Rxxx *endif-directive* **is** `#endif`

The `#elseif` directive has exactly the same uses and restrictions as the `#elif` directive. It simply acts as an alternate spelling for `#elif`.

When the source processor is evaluating expressions in C mode, the *constant-expression* of an *if-construct* may contain `defined` unary operator expressions of the form

`defined` *name*

or

`defined(`*name*`)`

These unary expressions evaluate to the integer value 1 if the name is currently defined as a macro name (that is, if it is predefined or if it has been the subject of a `#define` directive without an intervening `#undef` directive with the same name); otherwise, they evaluate to 0. The unary operator name `defined` shall not be the subject of a `#define` or a `#undef` directive.

If expressions are being evaluated according to 1539-1, the constant expression of an *if-construct* shall not contain these unary operator expressions.

[One reviewer would like `defined` to also be available when operating in Fortran expression mode. Opinions?]

Each conditional expression of an *if-construct* is a source processing expression involving macros (source processing variables specified by `#define` directives) and constants.

NOTE 4.5

Note that conditional expressions in a source processing *if-construct* are not required to be contained within parentheses. However, the rules for an expression in both the C standard and 1539-1 allow an expression to be contained with parentheses so the conditional expressions of an *if-construct* may be contained in parentheses if the user prefers that the *if-construct* look more similar to a Fortran language IF construct.

The `#if` and `#elif/#elseif` directives are used to inquire about the value of a macro where the `#ifdef` and `#ifndef` are used to inquire about the definition state of a macro.

The conditional expression of the `#if` and `#elif/#elseif` directives evaluates to a logical value if expressions are being evaluated in Fortran mode or to an integer value if expressions are being evaluated in C mode.

Prior to the evaluation of a *constant-expression* in an *if-construct*, any macros in the *constant-expression* are replaced (except for those macro names subject to `defined` unary operators when in C expression evaluation mode), just as in the remainder of the source text.

If expressions are being evaluated in Fortran mode then the resulting tokens comprise the initialization expression which is evaluated according to the rules of 1539-1, section 7.

If expressions are being evaluated in C mode, and if the token `defined` is generated as a result of this replacement process or use of the `defined` unary operator does not match one of the two specified forms prior to macro replacement, the behavior is undefined. After all replacements due to macro expansion and the `defined` unary operator have been performed, all remaining names are replaced with the token 0 (the number zero). The resulting tokens comprise the constant expression which is evaluated according to the rules in section 3.8.1 of the C standard, including those cases which the C standard notes are implementation-defined.

The conditional expression of the `#ifdef` and `#ifndef` directives determine whether the name is or is not currently defined as a macro name. The constant expression in these directives is equivalent to

```
#if defined name
```

and

```
#if !defined name
```

respectively.

NOTE 4.6

Note the difference between inquiring about the definition state of a macro name vs. its value: an `#ifdef` or `#ifndef` inquires about the definition state, but `#if` inquires about the value. If, for example, a source processing variable is defined as follows:

```
#define  OLD  0
```

then

```
#ifdef OLD
...           /* Fortran source text block. */
#endif
```

will output the Fortran source lines in the text block, but

```
#if OLD
...           /* Fortran source text block. */
#endif
```

will not.

The Fortran source lines in at most one of the source text blocks in an *if-construct* are output. If there is an `#else` directive in the *if-construct*, the Fortran source lines in exactly one of the source text blocks in the *if-construct* are output. The conditional expressions are evaluated in order of their appearance in the *if-construct* until a nonzero value (true value if evaluation is according to 1539-1) is found or an `#else` directive or `#endif` directive is encountered. If a nonzero (true) value or an `#else` directive is found, the Fortran source lines in the text block immediately following are output and this completes the output of the *if-construct*. The conditional expressions in any remaining `#elif` directives of the *if-construct* are not evaluated. If none of the evaluated expressions is nonzero (true) and there is no `#else` directive, no Fortran source lines in any text block in the construct are output. If Fortran source lines are output as a consequence of evaluating the *if-construct*, they are output after all macro replacements have been performed on the source lines, provided the source processor has been directed to perform macro replacements in Fortran source lines.

Example:

This example illustrates the default evaluation of an *if-construct* conditional expression as a C expression.

```
#define  LARGE_INTS
. . .
#if defined(LARGE_INTS)
    CALL sub(1_LARGE)
#else
    CALL sub(1)
#endif
```

This example illustrates the optional evaluation of a source processing variable value and an *if-construct* conditional expression according to 1539-1.

```
#define  LARGE_INTS  .TRUE.
. . .
#if (LARGE_INTS)
    CALL sub(1_LARGE)
#else
    CALL sub(1)
#endif
```

These examples are for illustration purposes only (the parens surrounding the conditional expression in the Fortran example are optional, for example). The same effects could be accomplished in other ways.

4.7 Including source text

Rxxx	*include-directive*	**is**	`#include` **<** *character* [*character*] ... **>**
		or	`#include` " *character* [*character*] ... "
		or	`#include` *sp-token* [*sp-token*] ...

The ", <, and > characters in the *include-directive* are a part of the syntax of the *include-directive*.

Additional source text, including source processing directives, may be incorporated into the program source text during source processing. This is accomplished with the `#include` directive.

The effect of the `#include` directive is as if the referenced source text physically replaced the `#include` directive prior to source processing. Included text may contain any source text, including additional `#include` directives. Any such nested `#include` directives are similarly replaced with the specified source text. The maximum depth of nesting of any nested `#include` directives is processor dependent. Inclusion of the source text referenced by an `#include` directive shall not, at any level of nesting, result in inclusion of the same source text.

[Difference: C apparently allows such recursive references. I went with the Fortran rules. Anyone think I should use the C rules instead?]

When an `#include` line is resolved, if the first included line is a Fortran statement line, it shall not be a Fortran statement continuation line. If the last included line is a Fortran statement line, it shall not be continued. If the first included line is a directive line, it shall not be a directive continuation line. If the last included line is a directive line, it shall not be continued.

The interpretation of the character string is processor dependent and varies according to the delimiters used or the lack of such delimiters. The source processor may ignore alphabetical case in the character string. The maximum allowed length of character string is processor dependent. The model used by this standard is that the character string is the name of a file that contains the source text to be included. If the file corresponding to the specified file name is not found, the source processor shall issue a diagnostic message. The behavior of further source processing is undefined.

The first form of the `#include` directive (the one that uses the < and > delimiters), directs the source processor to search a sequence of processor-dependent places for a file uniquely identified by the character sequence enclosed by the < and > delimiters, and causes the replacement of the `#include` directive by the entire contents of the file. The method by which the file is specified (the form of the file name) is processor dependent.

The second form of the #include directive (the one that uses the " character as the delimiters) causes the replacement of the directive by the entire contents of the file identified by the character sequence enclosed by the " delimiters. The file is searched for in a processor-dependent manner. If this search is not supported, or if the search fails, the directive is reprocessed as if it read

`#include` < *character* ... >

with the identical contained character sequence (including > characters, if any) from the original `#include` directive.

The tokens following `#include` in the third (undelimited) form of the `#include` directive are processed just as in normal text. That is, each name currently defined as a macro name is replaced by its replacement list of source processing tokens. The directive resulting after all replacements shall match one of the other forms of the `#include` directive. Adjacent character string literals are not concatenated into a single string literal; thus, an expansion that results in two string literals is an invalid directive. After all replacements have been made and the `#include` directive matches one of the other forms, the method by which a sequence of source processing tokens between the < and > delimiters or a pair of " delimiters is combined into a single file name is processor-dependent.

Examples:

Common uses of the `#include` directive are as follows:

```
#include <sys/ieee_defs>

#include "local_defs"
```

This example illustrates a macro-replaced `#include` directive:

```
#if VERSION == 1
    #define INCFILE  "version_1"
#elif VERSION == 2
    #define INCFILE  "version_2"
#else
    #define INCFILE  "version_3"
#endif

#include INCFILE
```

4.8 Error directive

Rxxx *error-directive* **is** `#error` [*character*] ...

The `#error` directive causes the source processor to produce a diagnostic message that includes the sequence of characters specified by the directive.

[One reviewer felt that the character strings included in the #error and #stop directives should be tokens so that macro substitution would take place in what the user supplied to these macros. I prefer that this *not* be the case. Opinions?]

4.9 Stop directive

Rxxx *stop-directive* **is** `#stop` [*character*] ...

The `#stop` directive causes termination of the source processor. At the time of termination, the sequence of characters specified by the directive, if any, is available in a processor-dependent manner.

4.10 Null directive

Rxxx *null-directive* **is** `#`

The null directive has no effect.

4.11 Line directive

Rxxx *line-directive* **is** `#line` *digit ...*
or `#line` *digit ...* "*character ...*"
or `#line` *sp-token ...*

In the `#line` directive, the " delimiter characters are a part of the syntax of the directive.

The syntactic class *sp-token* is defined by the description of source processing tokens in **[Low-level syntax]**.

The **line number** of the current source line is one greater that the number of lines that have been input to the source processor while processing the source text input file to the current token. Since each `#include` directive line exists as a line in the source text input file, each such `#include` line is considered when determing the line number for any given line. Thus, the `#line` directives output also account for `#include` lines.

The `#line` directive is the only directive that is entered into the output file by the source processor.

NOTE 4.7

The presence of `#line` directives in the output file implies that the output file does not strictly conform to 1539-1. It is generally assumed that if a source file is submitted to the source processor and then to a Fortran language processor that the Fortran language processor will recognize (or ignore) the `#line` directives, effectively removing them from the source processed by the remainder of the Fortran language processor such that the resulting source can constitute standard-conforming Fortran source input.

A `#line` directive of the form

`#line` *line-number*

indicates that the source line following the `#line` directive is to be treated as if the line number of that source line is *line-number* (interpreted as a decimal integer). The integer value of the line number shall not be zero. The upper bound on the integer value of the line number is processor dependent.

A `#line` directive of the form

`#line` *line-number* "*file-name*"

indicates the line number similarly and changes the presumed name of the source text input file to be the contents of the character string literal.

A `#line` directive of the form

`#line` *sp-token ...*

(that does not match one of the two previous forms) is permitted. The source processing tokens following the keyword `line` on the directive are processed just as in normal text (each name currently defined as a macro name is replaced by its replacement list of source processing tokens). The directive resulting after all replacements shall match one of the two previous forms and is then processed as appropriate.

[The `#line` directive can exist in both the input and output files. I need to separate these cases more and explain what they mean.]

Example:

Input to the source processor; assume the source is contained in file `t.F`:

```
! Line 1
```

```
! Line 2
! Line 3
#include "inc"
! Line 5
      END    ! Line 6
```

Contents of include file `"inc"`:

```
!  Inc line 1
!  Inc line 2
      i = 1   !  Inc line 3
!  Inc line 4
!  Inc line 5
```

Output from the source processor:

```
#line 1 "t.F"
! Line 1
! Line 2
! Line 3
#line 1 "inc"
!  Inc line 1
!  Inc line 2
      i = 1   !  Inc line 3
!  Inc line 4
!  Inc line 5
#line 5 "t.F"
! Line 5
      END    ! Line 6
```

4.12 Predefined macro names

The following macro names are defined intrinsically by the source processor:

__LINE__	The source file line number of the current source line (a decimal constant).
__FILE__	The name of the source input file (a character string literal).
__DATE__	The date the source processor began operating on the source text input file. The date shall be represented by a character string literal of the form "`Mmm dd yyyy`", where the names of the months are the same as those generated by the `asctime` intrinsic function as defined by the C standard, and the first character of `dd` is a space character if the value is less than 10. If the date the source processor began operating on the input file is not available, an implementation-defined valid date shall be supplied.
__TIME__	The time the source processor began operating on the source text input file. The time shall be represented by a character string literal of the form "`hh:mm:ss`" as in the time generated by the `asctime` intrinsic function as defined by the C standard. If the time the source processor began operating on the input file is not available, an implementation-defined valid time shall be supplied.

None of these predefined macro names shall be the subject of a `#define` or a `#undef` directive. If the processor supports a mechanism to initialize or override the definition state of a macro, these predefined macro names shall not be the subject of such a processor-dependent mechanism of initializing or overriding the definition state of macro name.