Fortran Language Reference Manual, Volume 3

007–3694–005

The CF90 compiler includes United States software patents 5,247,696, 5,257,372, and 5,361,354.

New Features

Fortran Language Reference Manual, Volume 3 007–3694–005

This manual describes the Fortran 95 language as implemented by the CF90 compiler, release 3.3, and the MIPSpro 7 Fortran 90 compiler, revision 7.3.

This revision contains miscellaneous corrections and additions to the previous revision.

Record of Revision

Version	*Description*
2.0	November 1995 Original Printing. The sections in this manuals previously appeared in the *CF90 Fortran Language Reference Manual*, revision 1.0, publication SR–3902, and the *CF90 Commands and Directives Reference Manual*.
3.0	May 1997 This printing supports the Cray Research CF90 3.0 release running on UNICOS and UNICOS/mk operating systems, and the MIPSpro 7 Fortran 90 compiler release 7.2 running on the IRIX operating system. The implementation of features on IRIX operating system platforms is deferred.
3.0.1	August 1997 This online revision supports the Cray Research CF90 3.0.1 release, running on UNICOS and UNICOS/mk operating systems, and the MIPSpro 7 Fortran 90 compiler 7.2 release, running on the IRIX operating system. Includes minor updates and corrections to revision 3.0.
3.0.2	March 1998 This online revision supports the Cray Research CF90 3.0.2 release, running on UNICOS and UNICOS/mk operating systems, and the MIPSpro 7 Fortran 90 compiler 7.2.1 release, running on the IRIX operating system. Includes minor updates and corrections to revision 3.0.1.
3.1	August 1998 This online revision supports the Cray Research CF90 3.1 release, running on UNICOS and UNICOS/mk operating systems, and the MIPSpro 7 Fortran 90 compiler 7.2.1 release, running on the IRIX operating system. Includes minor updates and corrections to revision 3.0.2.
3.2	January 1999 This revision (007–3694–004) supports the CF90 3.2 release, running on the UNICOS and UNICOS/mk operating systems, and the MIPSpro 7 Fortran 90 7.3 release, running on the IRIX operating system. It includes major updates to revision 3.1.
005	July 1999 This revision supports the CF90 3.3 release, running on the UNICOS and UNICOS/mk operating systems, and the MIPSpro 7 Fortran 90 7.3 release, running on the IRIX operating system. It includes minor updates to revision 3.2.

Contents

Page

Page

Page

Page

Page

Figures

Page

Page

Tables

Page

About This Manual

This manual describes the Fortran language as implemented by the CF90 compiler, revision 3.3, and by the MIPSpro 7 Fortran 90 compiler, revision 7.3. The CF90 and MIPSpro 7 Fortran 90 compilers implement the Fortran standard.

The CF90 and MIPSpro 7 Fortran 90 compilers run on UNICOS, UNICOS/mk, and IRIX operating systems. Specific hardware and operating system support information is as follows:

- The CF90 compiler runs under UNICOS 10.0.0.5, or later, on Cray SV1, Cray C90, Cray J90, and Cray T90 systems.
- The CF90 compiler runs under UNICOS/mk 2.0.4, or later, on Cray T3E systems.
- The MIPSpro 7 Fortran 90 compiler runs under IRIX 6.2, or later, on SGI systems.

The CF90 and MIPSpro 7 Fortran 90 compilers were developed to support the Fortran standard adopted by the American National Standards Institute (ANSI) and the International Standards Organization (ISO). This standard, commonly referred to in this manual as *the Fortran standard*, is ISO/IEC 1539–1:1997. Because the Fortran standard is, generally, a superset of previous standards, the CF90 and MIPSpro 7 Fortran 90 compilers will compile code written to previous standards.

Note: The Fortran 95 standard is a revision to the Fortran 90 standard. The standards organizations continue to interpret the Fortran standard for SGI and for other vendors. To maintain conformance to the Fortran standard, SGI may need to change the behavior of certain CF90 and MIPSpro 7 Fortran 90 compiler features in future releases based upon the outcomes of interpretations to the standard.

Related CF90 and MIPSpro 7 Fortran 90 Compiler Publications

This manual is one of a set of manuals that describes the CF90 and the MIPSpro 7 Fortran 90 compilers. The complete set of CF90 and MIPSpro 7 Fortran 90 compiler manuals is as follows:

- *Intrinsic Procedures Reference Manual*.

- *Fortran Language Reference Manual, Volume 1*. Chapters 1 through 8 correspond to sections 1 through 8 of the Fortran standard.
- *Fortran Language Reference Manual, Volume 2*. Chapters 1 through 6 of this manual correspond to sections 9 through 14 of the Fortran standard.
- *Fortran Language Reference Manual, Volume 3*. This manual contains CF90 and MIPSpro 7 Fortran 90 compiler information that supplements the Fortran standard. The standard is the complete, official description of the language. This manual also contains the complete Fortran syntax in Backus-Naur form (BNF).

The following publications contain information specific to the CF90 compiler:

- *CF90 Ready Reference*
- *CF90 Commands and Directives Reference Manual*
- *CF90 Co-array Programming Manual*

The following publication contains information specific to the MIPSpro 7 Fortran 90 compiler:

- *MIPSpro 7 Fortran 90 Commands and Directives Reference Manual*

CF90 and MIPSpro 7 Fortran 90 Compiler Messages

You can obtain CF90 and MIPSpro 7 Fortran 90 compiler message explanations by using the online `explain`(1) command.

CF90 and MIPSpro 7 Fortran 90 Compiler Man Pages

In addition to printed and online prose documentation, several online man pages describe aspects of the CF90 and MIPSpro 7 Fortran 90 compilers. Man pages exist for the library routines, the intrinsic procedures, and several programming environment tools.

You can print copies of online man pages by using the pipe symbol with the `man`(1), `col`(1), and `lpr`(1) commands. In the following example, these commands are used to print a copy of the `explain`(1) man page:

```
% man explain | col -b | lpr
```

Each man page includes a general description of one or more commands, routines, system calls, or other topics, and provides details of their usage

(command syntax, routine parameters, system call arguments, and so on). If more than one topic appears on a page, the entry in the printed manual is alphabetized under its primary name; online, secondary entry names are linked to these primary names. For example, `egrep` is a secondary entry on the page with a primary entry name of `grep`. To access `egrep` online, you can type `man grep` or `man egrep`. Both commands display the `grep` man page to your terminal.

Related Fortran Publications

The following commercially available reference books are among those that you can consult for more information on the history of Fortran and the Fortran language itself:

- Adams, J. C., W. S. Brainerd, J. T. Martin, B. T. Smith, and J. L. Wagener. *Fortran 95 Handbook : Complete ISO/ANSI Reference*. MIT Press, 1997. ISBN 0262510960.
- Chapman, S. *Fortran 90/95 for Scientists and Engineers*. McGraw Hill Text, 1998. ISBN 0070119384.
- Chapman, S. *Introduction to Fortran 90/95*. McGraw Hill Text, 1998. ISBN 0070119694.
- Counihan, M. *Fortran 95 : Including Fortran 90, Details of High Performance Fortran (HPF), and the Fortran Module for Variable-Length Character Strings*. UCL Press, 1997. ISBN 1857283678.
- Gehrke, W. *Fortran 95 Language Guide*. Springer Verlag, 1996. ISBN 3540760628.
- International Standards Organization. *ISO/IEC 1539–1:1997, Information technology — Programming languages — Fortran*. 1997.
- Metcalf, M. and J. Reid. *Fortran 90/95 Explained*. Oxford University Press, 1996. ISBN 0198518889.

Related Publications

Certain other publications from SGI may also interest you.

On UNICOS and UNICOS/mk systems, the following documents contain information that may be useful when using the CF90 compiler:

- *Segment Loader (SEGLDR) and ld Reference Manual*
- *UNICOS User Commands Reference Manual*
- *UNICOS Performance Utilities Reference Manual*
- *Scientific Library Reference Manual*
- *Introducing the Program Browser*
- *Application Programmer's Library Reference Manual*
- *Guide to Parallel Vector Applications*
- *Introducing the Cray TotalView Debugger*
- *Introducing the MPP Apprentice Tool*
- *Application Programmer's I/O Guide*
- *Optimizing Application Code on UNICOS Systems*
- *Compiler Information File (CIF) Reference Manual*

On IRIX systems, the following documents contain information that may be useful when using the MIPSpro 7 Fortran 90 compiler:

- *MIPSpro Compiling and Performance Tuning Guide*
- *MIPSpro Fortran 77 Programmer's Guide*
- *MIPSpro 64-Bit Porting and Transition Guide*
- *MIPSpro Assembly Language Programmer's Guide*

Obtaining Publications

The *User Publications Catalog* describes the availability and content of all Cray hardware and software documents that are available to customers. Customers who subscribe to the Cray Inform (CRInform) program can access this information on the CRInform system.

To order a document, call +1 651 683 5907. SGI employees may send e-mail to `orderdsk@sgi.com`.

Customers who subscribe to the CRInform program can order software release packages electronically by using the `Order Cray Software` option.

Customers outside of the United States and Canada should contact their local service organization for ordering and documentation information.

Conventions

The following conventions are used throughout this document:

Convention	Meaning
`command`	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.
variable	Italic typeface denotes variable entries and words or concepts being defined.
`user input`	This bold, fixed-space font denotes literal items that the user enters in interactive sessions. Output is shown in nonbold, fixed-space font.
[ ]	Brackets enclose optional portions of a command or directive line.
...	Ellipses indicate that a preceding element can be repeated.
DEL or DELETED	The DEL or DELETED notation indicates that the feature being described has been deleted from the Fortran standard. The CF90 and MIPSpro 7 Fortran 90 compilers support these features, but the compilers issue a message when a deleted feature is encountered.
EXT or EXTENSION	The EXT or EXTENSION notation indicates that the feature being described is an extension to the Fortran standard. The CF90 and MIPSpro 7 Fortran 90 compilers issue a message when extensions are encountered.
OBS or OBSOLESCENT	The OBS or OBSOLESCENT notation indicates that the feature being described is considered to be obsolete in the Fortran standard. The CF90 and MIPSpro 7 Fortran 90 compilers support these features, but the compilers issue a message when an obsolescent feature is encountered.

xyz_list	When *_list* is part of a syntax description, it means that several items may be specified. For example, *xyz_list* can be expanded to mean *xyz* [, *xyz*] ... .
*scalar_*	When *scalar_* is the first item in a syntax description, it indicates that the item is a scalar, not an array, value.
_name	When *_name* is part of a syntax definition, it indicates that the item is a name with no qualification. For example, the item must not have a subscript list, so `ARRAY` is a name, but `ARRAY(I)` is not.
(R*nnnn*)	Indicates that the Fortran 90 standard has rules regarding the characteristic of the language being discussed. All rules are numbered, and the numbered list appears in the *Fortran Language Reference Manual, Volume 3*. The numbering of the rules in the *Fortran Language Reference Manual, Volume 3* matches the numbering of the rules in the standard. The forms of the rules in the *Fortran Language Reference Manual, Volume 3* and the BNF syntax class terms that are used may differ from the rules and terms used in the standard.
`POINTER`	The term `POINTER` refers to the Fortran `POINTER` attribute.
Cray pointer	The term *Cray pointer* refers to the Cray pointer data type extension.
Fortran Fortran standard	These terms refer to the current Fortran standard, which is the Fortran 95 standard. For situations when it might otherwise be confusing, a specific standard is mentioned along with its numeric identifier (FORTRAN 77, Fortran 90, Fortran 95).

BNF Conventions

This section describes some of the commonly used Backus-Naur Form (BNF) conventions.

Terms such as *goto_stmt* are called *variable entries*, *nonterminal symbols*, or simply, *nonterminals*. The metalanguage term *goto_stmt*, for example, represents the `GO TO` statement, as follows:

goto_stmt	**is**	`GOTO` *label*

The syntax rule defines *goto_stmt* to be `GO TO` *label*, which describes the format of the `GO TO` statement. The description of the `GO TO` statement is incomplete until the definition of *label* is given. *label* is also a nonterminal symbol. A further search for *label* will result in a specification of *label* and thereby provide the complete statement definition. A *terminal* part of a syntax rule is one that does not need further definition. For example, `GO TO` is a terminal keyword and is a required part of the statement form. The complete BNF list appears in the *Fortran Language Reference Manual, Volume 3*.

The following abbreviations are commonly used in naming nonterminal keywords:

Abbreviation	Term
arg	argument
attr	attribute
char	character
decl	declaration
def	definition
desc	descriptor
expr	expression
int	integer
op	operator
spec	specifier or specification
stmt	statement

The term **is** separates the syntax class name from its definition. The term **or** indicates an alternative definition for the syntactic class being defined. The following example shows that *add_op*, the add operator, may be either a plus sign (+) or a minus sign (-):

add_op	**is**	+
	or	-

Indentation indicates syntax continuation. If a rule does not fit on one line, the second line is indented. This is shown in the following example:

dimension_stmt	**is**	DIMENSION [::] *array_name* (*array_spec*) [, *array_name* (*array_spec*)] ...

Reader Comments

If you have comments about the technical accuracy, content, or organization of this document, please tell us. Be sure to include the title and part number of the document with your comments.

You can contact us in any of the following ways:

- Send e-mail to the following address:

  techpubs@sgi.com

- Send a fax to the attention of "Technical Publications" at: +1 650 932 0801.
- Use the Feedback option on the Technical Publications Library World Wide Web page:

  http://techpubs.sgi.com

- Call the Technical Publications Group, through the Technical Assistance Center, using one of the following numbers:

  For SGI IRIX based operating systems: 1 800 800 4SGI

For UNICOS or UNICOS/mk based operating systems or Cray Origin 2000 systems: 1 800 950 2729 (toll free from the United States and Canada) or +1 651 683 5600

- Send mail to the following address:

  Technical Publications
  SGI
  1600 Amphitheatre Pkwy.
  Mountain View, California 94043–1351

We value your comments and will respond to them promptly.

Fortran Syntax [1]

This chapter contains a complete description of the Fortran syntax. Section 1.1 describes the format of the syntax. Section 1.2, page 5, contains the complete syntax and constraints as they appear in the Fortran standard. A high-level summary of the syntax appears in the *Fortran Language Reference Manual, Volume 1*.

1.1 Syntax Form

The syntax of Fortran programs is described using a variant of the Backus-Naur Form (BNF).

1.1.1 Syntax Rules Expressed in BNF

The BNF syntax rules are expressed as a definition. The metalanguage class being defined is first, followed by the symbol **is**, and finally the syntax definition, as in the following example:

goto_stmt **is** `GO TO` *label*

The term *goto_stmt* represents the `GO TO` statement; such terms are called *nonterminal symbols* or simply *nonterminals*. The syntax rule defines *goto_stmt* as `GO TO` *label*, which describes the form of the `GO TO` statement. The description of the `GO TO` statement is not complete until the definition of *label* is specified; *label* is also a nonterminal symbol. A further search for *label* in the BNF will result in a specification of *label* and thereby provide the complete statement definition. A *terminal* part of a syntax rule does not need further definition. For example, `GO TO` is a terminal and is a required part of the statement form.

In many cases, you can derive information about the metalanguage class from part of the descriptive term. The part can be a complete word, such as *_list*, or a common abbreviation. Some abbreviations used consistently in metalanguage classes are listed in Table 1, page 2.

Table 1. Syntax metalanguage abbreviations

Abbreviation	Term
arg	Argument
attr	Attribute
char	Character
decl	Declaration
def	Definition
desc	Descriptor
expr	Expression
int	Integer
op	Operator
spec	Specifier or specification
stmt	Statement

For example, all class definitions that end with *_stmt* might be used to generate a complete list of the statements in Fortran.

1.1.2 Definition Syntax Symbol: Is

As the following example shows, the symbol **is** separates the syntax class name from its definition:

goto_stmt	**is**	`GO TO` *label*
power_op	**is**	`**`

1.1.3 Alternative Syntax Symbol: Or

The symbol **or** indicates an alternative definition for the syntactic class being defined. The following example shows that *add_op*, the add operator, can be either plus or minus.

add_op	**is**	+
	or	-

1.1.4 Optional Symbol: [ ]

Some syntactic definitions contain optional items, which are enclosed in brackets. The term *sign* is optional in the following example:

signed_int_literal_constant **is** [*sign*] *int_literal_constant*

The fact that *sign* is optional indicates, for example, that both `75` and `+75` are *signed_int_literal_constant*s.

1.1.5 Symbol for Repeated Items: [ ] . . .

Enclosing an item in brackets followed by an ellipsis indicates that the item can occur 0 or more times. In the following example, the term *digit* is repeated as many times as required to define the *int_literal_constant*:

int_literal_constant **is** *digit* [*digit*] ...

For example, there are five digits in the integer literal constant `94024`.

1.1.6 Syntax Rule Continuation

If a rule does not fit on one line, the convention is to indent the second line of the syntax. This is shown in the following example:

allocatable_stmt **is** `ALLOCATABLE [ :: ]`
array_name `[(` *deferred_shape_spec_list* `)]`
`[,` *array_name* `[(` *deferred_shape_spec_list* `)]]...`

1.1.7 Assumed Syntax Rules

In order to minimize the number of syntax rules and still convey an appropriate meaning, some portions of the BNF metaterms have assumed meanings. In the following example, *xyz* represents any BNF phrase:

xyz_list	**means**	*xyz* [, *xyz*] ...
xyz_name	**is**	a name
scalar_xyz	**is**	an *xyz* that is a scalar

1.1.8 Example BNF Syntax

Consider the following example:

read_stmt	**is**	READ (*io_control_spec_list*) [*input_item_list*]
	or	READ *format* [, *input_item_list*]
format	**is**	*default_char_expr*
	or	*label*
	or	*
	or	*scalar_default_int_variable*

In this example, there are two alternatives to the READ statement. The first uses an input/output (I/O) control specification list; the second is a formatted READ statement where the unit is processor dependent. Both alternatives have an optional input item list, indicated by [ ]. The syntax class *format* (a nonterminal) is further defined as either a default character expression containing the format specifications, or a statement label referring to a separate FORMAT statement that contains the format specifications, or an asterisk (*) indicating that the READ statement is list-directed, or a scalar default integer variable whose value specifies the label of a FORMAT statement. In the standard, the last alternative is printed in a smaller font because it is an obsolescent feature that may be removed in a later revision of the standard, including the next revision; this convention is not used in this manual.

There are other nonterminal symbols in the description of the READ statement and further BNF rules need to be examined to determine the complete description of the READ statement.

1.1.9 Constraints

The BNF forms do not provide a complete description of the syntax; additional constraints are described with text. The BNF rules and the constraints both describe the syntax of Fortran. Constraints are restrictions to the syntax rules that limit the form of the statement described. If present, constraints appear following a syntax rule.

1.1.10 Identifying Numbers

In the text of the standard, each BNF rule is given an identifying number, R201 for example. The numbering of the rules in the following subsections matches the numbering of the rules in the standard.

BNF rules are also used to describe extensions. In the following BNF description, for example, "EXT" in the leftmost column indicates that the CF90 and MIPSpro 7 Fortran 90 compilers also allow *unit_name* to be used as an *io_unit*:

R901	*io_unit*	**is**	*external_file_unit*
		or	*
		or	*internal_file_unit*
EXT		**or**	*unit_name*

1.2 Syntax Rules and Constraints

Each of the following sections contains the syntax rules and constraints from a section of the Fortran standard. The following sections use an underscore, rather than a hyphen, as a separator; this differs from the Fortran standard. The rules in the following sections have been amended to include BNF for the CF90 and MIPSpro 7 Fortran 90 compiler extensions to the Fortran standard, but the constraints have not been modified to reflect the extensions.

1.2.1 Introduction

There are no syntax rules described in section 1, "Introduction," of the Fortran 95 standard.

1.2.2 Fortran Terms and Concepts

The following syntax rules are described in section 2, "Fortran terms and concepts," of the Fortran 95 standard.

R201	*executable_program*	**is**	*program_unit* [*program_unit*] . . .
R202	*program_unit*	**is**	*main_program*
		or	*external_subprogram*
		or	*module*
		or	*block_data*
R1101	*main_program*	**is**	[*program_stmt*] [*specification_part*] [*execution_part*] [*internal_subprogram_part*] *end_program_stmt*

Constraint: An *execution_part* must not contain an *end_function_stmt*, *end_program_stmt*, or *end_subroutine_stmt*.

R203	*external_subprogram*	**is**	*function_subprogram*
		or	*subroutine_subprogram*
R1215	*function_subprogram*	**is**	*function_stmt* [*specification_part*] [*execution_part*] [*internal_subprogram_part*] *end_function_stmt*

R1219	*subroutine_subprogram*	**is**	*subroutine_stmt* [*specification_part*] [*execution_part*] [*internal_subprogram_part*] *end_subroutine_stmt*
R1104	*module*	**is**	*module_stmt* [*specification_part*] [*module_subprogram_part*] *end_module_stmt*
R1112	*block_data*	**is**	*block_data_stmt* [*specification_part*] *end_block_data_stmt*
R204	*specification_part*	**is**	[*use_stmt*] . . . [*implicit_part*] [*declaration_construct*] . . .
R205	*implicit_part*	**is**	[*implicit_part_stmt*] . . . *implicit_stmt*
R206	*implicit_part_stmt*	**is**	*implicit_stmt*
		or	*parameter_stmt*
		or	*format_stmt*
		or	*entry_stmt*
R207	*declaration_construct*	**is**	*derived_type_def*
		or	*interface_block*
		or	*type_declaration_stmt*
		or	*specification_stmt*
		or	*parameter_stmt*
		or	*format_stmt*
		or	*entry_stmt*
		or	*stmt_function_stmt*
R208	*execution_part*	**is**	*executable_construct* [*execution_part_construct*] . . .
R209	*execution_part_construct*	**is**	*executable_construct*
		or	*format_stmt*

		or	*data_stmt*
		or	*entry_stmt*
R210	*internal_subprogram_part*	**is**	*contains_stmt* *internal_subprogram* [*internal_subprogram*]...
R211	*internal_subprogram*	**is**	*function_subprogram*
		or	*subroutine_subprogram*
R212	*module_subprogram_part*	**is**	*contains_stmt* *module_subprogram* [*module_subprogram*] ...
R213	*module_subprogram*	**is**	*function_subprogram*
		or	*subroutine_subprogram*
R214	*specification_stmt*	**is**	*access_stmt*
		or	*allocatable_stmt*
EXT		**or**	*automatic_stmt*
		or	*common_stmt*
		or	*data_stmt*
		or	*dimension_stmt*
		or	*equivalence_stmt*
		or	*external_stmt*
		or	*intent_stmt*
		or	*intrinsic_stmt*
		or	*namelist_stmt*
		or	*optional_stmt*
		or	*pointer_stmt*
		or	*save_stmt*
		or	*target_stmt*
EXT		**or**	*volatile_stmt*
R215	*executable_construct*	**is**	*action_stmt*
		or	*case_construct*

		or	*do_construct*
		or	*if_construct*
		or	*forall_construct*
		or	*where_construct*
R216	*action_stmt*	**is**	*allocate_stmt*
OBS		**or**	*arithmetic_if_stmt*
EXT		**or**	*assign_stmt*
EXT		**or**	*assigned_goto_stmt*
		or	*assignment_stmt*
		or	*backspace_stmt*
EXT		**or**	*buffer_in_stmt*
EXT		**or**	*buffer_out_stmt*
		or	*call_stmt*
		or	*close_stmt*
OBS		**or**	*computed_goto_stmt*
		or	*continue_stmt*
		or	*cycle_stmt*
		or	*deallocate_stmt*
		or	*endfile_stmt*
		or	*end_function_stmt*
		or	*end_program_stmt*
		or	*end_subroutine_stmt*
		or	*exit_stmt*
		or	*forall_stmt*
		or	*goto_stmt*
		or	*if_stmt*
		or	*inquire_stmt*
		or	*nullify_stmt*
		or	*open_stmt*

EXT	**or**	*pause_stmt*
	or	*pointer_assignment_stmt*
	or	*print_stmt*
	or	*read_stmt*
	or	*return_stmt*
	or	*rewind_stmt*
	or	*stop_stmt*
	or	*where_stmt*
	or	*write_stmt*

1.2.3 Characters, Lexical Tokens, and Source Form

The following syntax rules are described in section 3, "Characters, lexical tokens, and source form," of the Fortran 95 standard.

R301	*character*	**is**	*alphanumeric_character*
		or	*special_character*
R302	*alphanumeric_character*	**is**	*letter*
		or	*digit*
		or	*underscore*
EXT		**or**	*currency_symbol*
EXT		**or**	*at_sign*

Note: The MIPSpro 7 Fortran 90 compiler does not support the *at_sign* (@).

R303	*underscore*	**is**	_
EXT	*currency_symbol*	**is**	$
EXT	*at_sign*	**is**	@
R304	*name*	**is**	*letter* [*alphanumeric_character*] ...

Constraint: The maximum length of a name is 31 characters.

R305	*constant*	**is**	*literal_constant*
		or	*named_constant*
R306	*literal_constant*	**is**	*int_literal_constant*
		or	*real_literal_constant*
		or	*complex_literal_constant*
		or	*logical_literal_constant*
		or	*char_literal_constant*
		or	*boz_literal_constant*
R307	*named_constant*	**is**	*name*
R308	*int_constant*	**is**	*constant*

Constraint: *int_constant* must be of type integer.

R309	*char_constant*	**is**	*constant*

Constraint: *char_constant* must be of type character.

R310	*intrinsic_operator*	**is**	*power_op*
		or	*mult_op*
		or	*add_op*
		or	*concat_op*
		or	*rel_op*
		or	*not_op*
		or	*and_op*
		or	*or_op*
		or	*equiv_op*

R708	*power_op*	**is**	**
R709	*mult_op*	**is**	*
		or	/
R710	*add_op*	**is**	+
		or	-
R712	*concat_op*	**is**	//
R714	*rel_op*	**is**	.EQ.
		or	.NE.
		or	.LT.
		or	.LE.
		or	.GT.
		or	.GE.
EXT		**or**	.LG.
		or	==
		or	/=
		or	<
		or	<=
		or	>
		or	>=
EXT		**or**	<>
R719	*not_op*	**is**	.NOT.
EXT		**or**	.N.
R720	*and_op*	**is**	.AND.
EXT		**or**	.A.
R721	*or_op*	**is**	.OR.
EXT		**or**	.O.
R722	*equiv_op*	**is**	.EQV.
		or	.NEQV.
EXT	*exclusive_disjunct_op*	**is**	.XOR.

EXT		**or**	`.X.`
R311	*defined_operator*	**is**	*defined_unary_op*
		or	*defined_binary_op*
		or	*extended_intrinsic*
R704	*defined_unary_op*	**is**	`.` *letter* [*letter*] `...` `.`
R724	*defined_binary_op*	**is**	`.` *letter* [*letter*] `...` `.`
R312	*extended_intrinsic_op*	**is**	*intrinsic_operator*

Constraint: A *defined_unary_op* and a *defined_binary_op* must not contain more than 31 letters and must not be the same as any *intrinsic_operator* or *logical_literal_constant*.

R313	*label*	**is**	*digit* [*digit* [*digit* [*digit* [*digit*]]]]

Constraint: At least one digit in a *label* must be nonzero.

1.2.4 Intrinsic and Derived Data Types

The following syntax rules are described in section 4, "Intrinsic and derived data types," of the Fortran 95 standard.

R401	*signed_digit_string*	**is**	[*sign*] *digit_string*
R402	*digit_string*	**is**	*digit* [*digit*] `...`
R403	*signed_int_literal_constant*	**is**	[*sign*] *int_literal_constant*
R404	*int_literal_constant*	**is**	*digit_string* [`_` *kind_param*]
R405	*kind_param*	**is**	*digit_string*
		or	*scalar_int_constant_name*

Constraint: The value of *kind_param* must be nonnegative.

Constraint: The value of *kind_param* must specify a representation method that exists on the processor.

R406	*sign*	**is**	+
		or	-
R407	*boz_literal_constant*	**is**	*binary_constant*
		or	*octal_constant*
		or	*hex_constant*

Constraint: A *boz_literal_constant* may appear only in a DATA statement.

R408	*binary_constant*	**is**	B ' *digit* [*digit*] ... '
		or	B " *digit* [*digit*] ... "

Constraint: *digit* must have one of the values 0 or 1.

R409	*octal_constant*	**is**	O ' *digit* [*digit*] ... '
		or	O " *digit* [*digit*] ... "

Constraint: *digit* must have one of the values 0 through 7.

R410	*hex_constant*	**is**	Z ' *hex_digit* [*hex_digit*]... '
		or	Z " *hex_digit* [*hex_digit*] ... "
R411	*hex_digit*	**is**	*digit*
		or	A
		or	B
		or	C
		or	D
		or	E
		or	F
R412	*signed_real_literal_constant*	**is**	[*sign*] *real_literal_constant*

R413	*real_literal_constant*	**is**	*significand* [*exponent_letter exponent*] [_ *kind_param*]
		or	*digit_string exponent_letter exponent* [_ *kind_param*]
R414	*significand*	**is**	*digit_string* . [*digit_string*]
		or	. *digit_string*
R415	*exponent_letter*	**is**	E
		or	D
EXT		**or**	Q
R416	*exponent*	**is**	*signed_digit_string*

Constraint: If both *kind_param* and *exponent_letter* are present, *exponent_letter* must be E.

Constraint: The value of *kind_param* must specify an approximation method that exists on the processor.

R417	*complex_literal_constant*	**is**	(*real_part* , *imag_part*)
R418	*real_part*	**is**	*signed_int_literal_constant*
		or	*signed_real_literal_constant*
R419	*imag_part*	**is**	*signed_int_literal_constant*
		or	*signed_real_literal_constant*
R420	*char_literal_constant*	**is**	[*kind_param* _] ' [*ASCII_char*] ... '
		or	[*kind_param* _] " [*ASCII_char*] ... "

Constraint: The value of *kind_param* must specify a representation method that exists on the processor.

R421	*logical_literal_constant*	**is**	.TRUE. [_ *kind_param*]
		or	.FALSE. [_ *kind_param*]

Constraint: The value of *kind_param* must specify a representation method that exists on the processor.

R422	*derived_type_def*	**is**	*derived_type_stmt* [*private_sequence_stmt*] ... *component_def_stmt* [*component_def_stmt*] ... *end_type_stmt*
R423	*derived_type_stmt*	**is**	`TYPE` [[, *access_spec*] ::] *type_name*
R424	*private_sequence_stmt*	**is**	`PRIVATE`
		or	`SEQUENCE`

Constraint: The same *private_sequence_stmt* must not appear more than once in a given *derived_type_def*.

Constraint: If `SEQUENCE` is present, all derived types specified in component definitions must be sequence types.

Constraint: An *access_spec* or a `PRIVATE` statement within the definition is permitted only if the type definition is within the specification part of a module.

Constraint: A derived type *type_name* must not be the same as the name of any intrinsic type nor the same as any other accessible derived type *type_name*.

R425	*component_def_stmt*	**is**	*type_spec* [[, *component_attr_spec_list*] ::] *component_decl_list*
R426	*component_attr_spec*	**is**	`POINTER`
		or	`DIMENSION (`*component_array_spec*`)`
R427	*component_array_spec*	**is**	*explicit_shape_spec_list*
		or	*deferred_shape_spec_list*

Constraint: If a component of a derived type is of a type declared to be private, either the derived type definition must contain the `PRIVATE` statement or the derived type must be private.

Constraint: No *component_attr_spec* can appear more than once in a given *component_def_stmt*.

Constraint: If the `POINTER` attribute is not specified for a component, a *type_spec* in the *component_def_stmt* must specify an intrinsic type or a previously defined derived type.

Constraint: If the `POINTER` attribute is specified for a component, a *type_spec* in the *component_def_stmt* must specify an intrinsic type or any accessible derived type including the type being defined.

Constraint: If the `POINTER` attribute is not specified, each *component_array_spec* must be an *explicit_shape_spec_list*.

Constraint: If the `POINTER` attribute is specified, each *component_array_spec* must be a *deferred_shape_spec_list*.

R428	*component_decl*	**is**	*component_name* [(*component_array_spec*)] [* *char_length*] *component_initialization*
R429	*component_initialization*	**is**	= *initialization_expr*
		or	=> NULL()

Constraint: The * *char_length* option is permitted only if the type specified is character.

Constraint: The character length specified by a *char_length* in a *component_decl* or the *char_selector* in a *type_spec* must be a constant specification expression.

Constraint: Each bound in the *explicit_shape_spec* must be a constant specification expression.

Constraint: If *component_initialization* appears, a double colon separator (`::`) must appear before the *component_decl_list*.

Constraint: If `=>` appears in *component_initialization*, the `POINTER` attribute must appear in the *component_attr_spec_list*. If `=` appears in *component_initialization*, the `POINTER` attribute must not appear in the *component_attr_spec_list*.

R430	*end_type_stmt*	**is**	END TYPE [*type_name*]

Constraint: If `END TYPE` is followed by a *type_name*, the *type_name* must be the same as that in the corresponding *derived_type_stmt*.

R431	*structure_constructor*	**is**	*type_name* (*expr_list*)
R432	*array_constructor*	**is**	(/ *ac_value_list* /)
R433	*ac_value*	**is**	*expr*
		or	*ac_implied_do*
R434	*ac_implied_do*	**is**	(*ac_value_list* , *ac_implied_do_control*)
R435	*ac_implied_do_control*	**is**	*ac_do_variable* = *scalar_int_expr*, *scalar_int_expr* [, *scalar_int_expr*]
R436	*ac_do_variable*	**is**	*scalar_int_variable*

Constraint: *ac_do_variable* must be a named variable.

Constraint: Each *ac_value* expression in the *array_constructor* must have the same type and kind type parameters.

1.2.5 Data Object Declarations and Specifications

The following syntax rules are described in section 5, "Data object declarations and specifications," of the Fortran 95 standard.

R501	*type_declaration_stmt*	**is**	*type_spec* [[, *attr_spec*] ... ::] *entity_decl_list*
R502	*type_spec*	**is**	INTEGER [*kind_selector*]
EXT		**or**	INTEGER* *length_value*
		or	REAL [*kind_selector*]
EXT		**or**	REAL* *length_value*
		or	DOUBLE PRECISION
EXT		**or**	DOUBLE PRECISION* *length_value*
		or	COMPLEX [*kind_selector*]
EXT		**or**	COMPLEX* *length_value*
		or	CHARACTER [*char_selector*]
		or	LOGICAL [*kind_selector*]
EXT		**or**	LOGICAL* *length_value*
		or	TYPE (*type_name*)

EXT		**or**	`POINTER ( pointer_name, pointee_name [( array_spec )] )` `[, (pointer_name, pointee_name [( array_spec )] )] ...`
R503	*attr_spec*	**is**	`PARAMETER`
		or	*access_spec*
		or	`ALLOCATABLE`
EXT		**or**	`AUTOMATIC`
		or	`DIMENSION ( array_spec )`
		or	`EXTERNAL`
		or	`INTENT ( intent_spec )`
		or	`INTRINSIC`
		or	`OPTIONAL`
		or	`POINTER`
		or	`SAVE`
		or	`TARGET`
EXT		**or**	`VOLATILE`
R504	*entity_decl*	**is**	*object_name* `[( array_spec )] [ * char_length ] [ = initialization_expr ]`
		or	*function_name* `[ * char_length ]`
R505	*initialization*	**is**	`= initialization_expr`
		or	`=> NULL()`
R506	*kind_selector*	**is**	`([ KIND = ] scalar_int_initialization_expr )`

Constraint: The same *attr_spec* must not appear more than once in a given *type_declaration_stmt*.

Constraint: The *function_name* must be the name of an external function, an intrinsic function, a function dummy procedure, or a statement function.

Constraint: The *initialization_expr* must appear if the statement contains a `PARAMETER` attribute.

Constraint: If *initialization_expr* appears, a double colon separator (`::`) must appear before the *entity_decl_list*.

Constraint: The *initialization_expr* must not appear if *object_name* is a dummy argument, a function result, or an object in a named common block unless the

type declaration is in a block data program unit, an object in blank common, an allocatable array, an external name, an intrinsic name, or an automatic object.

Constraint: The **char_length* option is permitted only if the type specified is character.

Constraint: The ALLOCATABLE attribute may be used only when declaring an array that is not a dummy argument or a function result.

Constraint: An array declared with a POINTER or an ALLOCATABLE attribute must be specified with an *array_spec* that is a *deferred_shape_spec_list*.

Constraint: An *array_spec* for an *object_name* that is a function result that does not have the POINTER attribute must be an *explicit_shape_spec_list*.

Constraint: An *array_spec* for an *object_name* that is a function result that has the POINTER attribute must be a *deferred_shape_spec_list*.

Constraint: If the POINTER attribute is specified, the TARGET, INTENT, EXTERNAL, or INTRINSIC attribute must not be specified.

Constraint: If the TARGET attribute is specified, the POINTER, EXTERNAL, INTRINSIC, or PARAMETER attribute must not be specified.

Constraint: The PARAMETER attribute must not be specified for dummy arguments, pointers, allocatable arrays, functions, or objects in a common block.

Constraint: The INTENT and OPTIONAL attributes may be specified only for dummy arguments.

Constraint: An entity must not have the PUBLIC attribute if its type has the PRIVATE attribute.

Constraint: The SAVE attribute must not be specified for an object that is in a common block, a dummy argument, a procedure, a function result, an automatic data object, or an object with the PARAMETER attribute.

Constraint: An entity must not have the EXTERNAL attribute if it has the INTRINSIC attribute.

Constraint: An entity in an *entity_decl_list* must not have the EXTERNAL or INTRINSIC attribute specified unless it is a function.

Constraint: If => appears in *initialization*, the object must have the POINTER attribute. If = appears in *initialization*, the object must not have the POINTER attribute.

Constraint: An array must not have both the `ALLOCATABLE` attribute and the `POINTER` attribute.

Constraint: An entity must not be given explicitly any attribute more than once in a scoping unit.

Constraint: The value of *scalar_int_initialization_expr* must be nonnegative and must specify a representation method that exists on the processor.

R507	*char_selector*	**is**	*length_selector*
		or	`(` `LEN =` *char_len_param_value*`,` `KIND =` *scalar_int_initialization_expr* `)`
		or	`(` *char_len_param_value*`,` `[` `KIND =` `]` *scalar_int_initialization_expr* `)`
		or	`(` `KIND =` *scalar_int_initialization_expr* `[,` `LEN =` *char_len_param_value* `])`
R508	*length_selector*	**is**	`([` `LEN =` `]` *char_len_param_value* `)`
OBS		**or**	`*` *char_length* `[` `,` `]`
R509	*char_length*	**is**	`(` *char_len_param_value* `)`
		or	*scalar_int_literal_constant*

Obsolescent Constraint: The optional comma in a *length_selector* is permitted only in a *type_spec* in a *type_declaration_stmt*.

Obsolescent Constraint: The optional comma in a *length_selector* is permitted only if no double colon separator appears in the *type_declaration_stmt*.

Constraint: The value of *scalar_int_initialization_expr* must be nonnegative and must specify a representation method that exists on the processor.

Constraint: The *scalar_int_literal_constant* must not include a *kind_param*.

R510	*char_len_param_value*	**is**	*specification_expr*
		or	`*`

Obsolescent Constraint: A function name must not be declared with a `*`*char_len_param_value* unless it is the name of an external function or the name of a dummy function.

Constraint: A function name declared with a **char_len_param_value* must not be array-valued, pointer-valued, pure, or recursive.

R511	*access_spec*	**is**	`PUBLIC`
		or	`PRIVATE`

Constraint: An *access_spec* attribute may appear only in the specification part of a module.

R512	*intent_spec*	**is**	`IN`
		or	`OUT`
		or	`INOUT`

Constraint: The `INTENT` attribute must not be specified for a dummy argument that is a dummy procedure or a dummy pointer.

Constraint: A dummy argument with the `INTENT(IN)` attribute, or a subobject of such a dummy argument, must not appear as any of the following:

- The *variable* of an *assignment_stmt*
- The *pointer_object* of a *pointer_assignment_stmt*
- A `DO`-variable or implied `DO`-variable
- An *input_item* in a *read_stmt*
- A *variable_name* in a *namelist_stmt* if the *name_list_group_name* appears in an `NML=` specifier in a *read_stmt*
- An *internal_file_unit* in a *write_stmt*
- An `IOSTAT=` or `SIZE=` specifier in an I/O statement
- A definable variable in an `INQUIRE` statement
- A *stat_variable* or *allocate_object* in an *allocate_stmt* or a *deallocate_stmt*
- An actual argument in a reference to a procedure with an explicit interface when the associated dummy argument has the `INTENT(OUT)` or `INTENT(INOUT)` attribute

R513	*array_spec*	**is**	*explicit_shape_spec_list*
		or	*assumed_shape_spec_list*
		or	*deferred_shape_spec_list*
		or	*assumed_size_spec*

Constraint: The maximum rank is seven.

R514	*explicit_shape_spec*	**is**	[*lower_bound* :] *upper_bound*
R515	*lower_bound*	**is**	*specification_expr*
R516	*upper_bound*	**is**	*specification_expr*

Constraint: An explicit-shape array whose bounds depend on the values of nonconstant expressions must be a dummy argument, a function result, or an automatic array of a procedure.

R517	*assumed_shape_spec*	**is**	[*lower_bound*] :
R518	*deferred_shape_spec*	**is**	:
R519	*assumed_size_spec*	**is**	[*explicit_shape_spec_list*,] [*lower_bound* :] *

Constraint: The function name of an array-valued function must not be declared as an assumed-size array.

Constraint: An assumed-size array with `INTENT(OUT)` must not be of a type for which default initialization is specified.

R520	*intent_stmt*	**is**	`INTENT (` *intent_spec* `)` [`::`] *dummy_arg_name_list*

Constraint: An *intent_stmt* may appear only in the *specification_part* of a subprogram or an interface body.

Constraint: *dummy_arg_name* must not be the name of a dummy procedure or a dummy pointer.

R521	*optional_stmt*	**is**	OPTIONAL [::] *dummy_arg_name_list*

Constraint: An *optional_stmt* can occur only in the *specification_part* of a subprogram or an interface body.

R522	*access_stmt*	**is**	*access_spec* [[::] *access_id_list*]
R523	*access_id*	**is**	*use_name*
		or	*generic_spec*

Constraint: An *access_stmt* can appear only in the *specification_part* of a module. Only one accessibility statement with an omitted *access_id_list* is permitted in the *specification_part* of a module.

Constraint: Each *use_name* must be the name of a named variable, procedure, derived type, named constant, or namelist group.

Constraint: A module procedure that has a dummy argument or function result of a type that has PRIVATE accessibility must have PRIVATE accessibility and must not have a generic identifier that has PUBLIC accessibility.

R524	*save_stmt*	**is**	SAVE [[::] *saved_entity_list*]
R525	*saved_entity*	**is**	*object_name*
		or	/ *common_block_name* /

Constraint: An *object_name* must not be the name of an object in a common block, a dummy argument name, a procedure name, a function result name, an automatic data object name, or the name of an object with the PARAMETER attribute.

Constraint: If a SAVE statement with an omitted saved entity list occurs in a scoping unit, no other explicit occurrence of the SAVE attribute or SAVE statement is permitted in the same scoping unit.

R526	*dimension_stmt*	**is**	`DIMENSION [ :: ]` *array_name*`(`*array_spec*`)` `[,` *array_name*`(`*array_spec*`)] ...`
R527	*allocatable_stmt*	**is**	`ALLOCATABLE [ :: ]` *array_name* `[(`*deferred_shape_spec_list*`)]` `[,` *array_name* `[(`*deferred_shape_spec_list*`)] ] ...`

Constraint: The *array_name* must not be a dummy argument or function result.

Constraint: If the `DIMENSION` attribute for an *array_name* is specified elsewhere in the scoping unit, the *array_spec* must be a *deferred_shape_spec_list*.

R528	*pointer_stmt*	**is**	`POINTER [ :: ]` *object_name* `[ (`*deferred_shape_spec_list*`) ]` `[,` *object_name* `[ (`*deferred_shape_spec_list*`) ] ] ...`

Constraint: The `INTENT` attribute must not be specified for an *object_name*.

Constraint: If the `DIMENSION` attribute for an *object_name* is specified elsewhere in the scoping unit, the *array_spec* must be a *deferred_shape_spec_list*.

Constraint: The `PARAMETER` attribute must not be specified for an *object_name*.

R529	*target_stmt*	**is**	`TARGET [ :: ]` *object_name* `[ (`*array_spec*`) ]` `[,` *object_name* `[ (`*array_spec*`) ] ] ...`

Constraint: The `PARAMETER` attribute must not be specified for an *object_name*.

R532	*data_stmt*	**is**	`DATA` *data_stmt_set* `[ [ , ]` *data_stmt_set* `] ...`
R533	*data_stmt_set*	**is**	*data_stmt_object_list* `/` *data_stmt_value_list* `/` `[ [ , ]` *data_stmt_object_list* `/` *data_stmt_value_list* `/ ] ...`
R534	*data_stmt_object*	**is**	*variable*
		or	*data_implied_do*

R535	*data_implied_do*	**is**	(*data_i_do_object_list*, *data_i_do_variable* = *scalar_int_expr*, *scalar_int_expr* [, *scalar_int_expr*])
R536	*data_i_do_object*	**is**	*array_element*
		or	*scalar_structure_component*
		or	*data_implied_do*
R538	*data_stmt_value*	**is**	[*data_stmt_repeat* *] *data_stmt_constant*
R539	*data_stmt_repeat*	**is**	*scalar_int_constant*
		or	*scalar_int_constant_subobject*
R540	*data_stmt_constant*	**is**	*scalar_constant*
		or	*scalar_constant_subobject*
		or	*signed_int_literal_constant*
		or	*signed_real_literal_constant*
		or	*structure_constructor*
		or	NULL()
		or	*boz_literal_constant*
EXT		**or**	*typeless_constant*

A *data_i_do_variable* must be a named variable.

Constraint: The *array_element* must not have a constant parent.

Constraint: The *scalar_structure_component* must not have a constant parent.

Constraint: In a *scalar_int_constant_subobject* that is a *data_stmt_repeat*, any subscript must be an initialization expression.

Constraint: In a *scalar_constant_subobject* that is a *data_stmt_constant*, any subscript, substring starting point, or substring ending point must be an initialization expression.

Constraint: If a DATA statement constant value is a named constant or a structure constructor, the named constant or derived type must have been declared previously in the scoping unit or made accessible by USE or host association.

R537	*data_i_do_variable*	**is**	*scalar_int_variable*

Constraint: *data_i_do_variable* must be a named variable.

Constraint: The DATA statement repeat factor must be positive or zero. If the DATA statement repeat factor is a named constant, it must have been declared previously in the scoping unit or made accessible by use association or host association.

Constraint: In a *scalar_int_constant_subobject* that is a *data_stmt_repeat*, any subscript must be an initialization expression.

Constraint: In a *scalar_constant_subobject* that is a *data_stmt_constant*, any subscript, substring starting point, or substring ending point must be an initialization expression.

Constraint: If a *data_stmt_constant* is a *structure_constructor*, each component must be an initialization expression.

Constraint: In a variable that is a *data_stmt_object*, any subscript, section subscript, substring starting point, and substring ending point must be an initialization expression.

Constraint: A variable whose name or designator is included in a *data_stmt_object_list* or a *data_i_do_object_list* must not be: a dummy argument; made accessible by use association or host association; in a named common block unless the DATA statement is in a block data program unit; in a blank common block, a function name, a function result name, an automatic object, or an allocatable array.

Constraint: In an *array_element* or a *scalar_structure_component* that is a *data_i_do_object*, any subscript must be an expression whose primaries are either constants, subobjects of constants, or DO variables of the containing *data_implied_do* elements, and each operation must be intrinsic.

Constraint: A *scalar_int_expr* of a *data_implied_do* must involve as primaries only constants, subobjects of constants, or DO variables of the containing *data_implied_do*s, and each operation must be intrinsic.

Constraint: The *scalar_structure_component* must contain at least one *part_ref* that contains a *subscript_list*.

EXT	*typeless_constant*	**is**	*octal_typeless_constant*
		or	*hexadecimal_typeless_constant*
		or	*binary_typeless_constant*
EXT	*octal_typeless_constant*	**is**	*digit* [*digit*] ... B
		or	O" *digit* [*digit*] ... "
		or	O' *digit* [*digit*] ... '
		or	" *digit* [*digit*] ... "O
		or	' *digit* [*digit*] ... 'O
EXT	*hexadecimal_typeless_constant*	**is**	X' *hex_digit* [*hex_digit*] ... '
		or	X" *hex_digit* [*hex_digit*] ... "
		or	' *hex_digit* [*hex_digit*] ... 'X
		or	" *hex_digit* [*hex_digit*] ... "X
		or	Z' *hex_digit* [*hex_digit*] ... '
		or	Z" *hex_digit* [*hex_digit*] ... "
EXT	*binary_typeless_constant*	**is**	B' *bin_digit* [*bin_digit*] ... '
		or	B" *bin_digit* [*bin_digit*] ... "

The following notes pertain to the definitions for *typeless_constant*, *octal_typeless_constant*, *hexadecimal_typeless_constant*, and *binary_typeless_constant*:

- *digit* must have one of the values 0 through 7 in *octal_typeless_constant*
- *digit* must have a value of 0 or 1 in *binary_typeless_constant*
- The B, O, X, and Z characters can be in uppercase or lowercase.

R530	*parameter_stmt*	**is**	PARAMETER (*named_constant_def_list*)
R531	*named_constant_def*	**is**	*named_constant* = *initialization_expr*
R541	*implicit_stmt*	**is**	IMPLICIT *implicit_spec_list*
		or	IMPLICIT NONE
EXT		**or**	IMPLICIT UNDEFINED

R542	*implicit_spec*	**is**	*type_spec* (*letter_spec_list*)
R543	*letter_spec*	**is**	*letter* [- *letter*]

Constraint: If `IMPLICIT NONE` is specified in a scoping unit, it must precede any `PARAMETER` statements that appear in the scoping unit and there must be no other `IMPLICIT` statements in the scoping unit.

Constraint: If the minus and second letter appear, the second letter must follow the first letter alphabetically.

R544	*namelist_stmt*	**is**	`NAMELIST /` *namelist_group_name* `/` *namelist_group_object_list* [[,] `/` *namelist_group_name* `/` *namelist_group_object_list*] ...
R545	*namelist_group_object*	**is**	*variable_name*

Constraint: A *namelist_group_object* must not be an array dummy argument with a nonconstant bound, a variable with nonconstant character length, an automatic object, a pointer, a variable of a type that has an ultimate component that is a pointer, or an allocatable array.

Constraint: If a *namelist_group_name* has the `PUBLIC` attribute, no item in the *namelist_group_object_list* can have the `PRIVATE` attribute or have private components.

Constraint: The *namelist_group_name* must not be a name made accessible by `USE` association.

R546	*equivalence_stmt*	**is**	`EQUIVALENCE` *equivalence_set_list*
R547	*equivalence_set*	**is**	(*equivalence_object*, *equivalence_object_list*)
R548	*equivalence_object*	**is**	*variable_name*
		or	*array_element*
		or	*substring*

Constraint: An *equivalence_object* must not be a dummy argument, a pointer, an allocatable array, an object of a nonsequence derived type or of a sequence derived type containing a pointer at any level of component selection, an

automatic object, a function name, an entry name, a result name, a named constant, a structure component, or a subobject of any of the preceding objects.

Constraint: An *equivalence_object* must not have the TARGET attribute.

Constraint: Each subscript or substring range expression in an *equivalence_object* must be an integer initialization expression.

Constraint: If an *equivalence_object* is of type default integer, default real, double-precision real, default complex, default logical, or numeric sequence type, all of the objects in the equivalence set must be of these types.

Constraint: If an *equivalence_object* is of type default character or character sequence type, all of the objects in the equivalence set must be of these types.

Constraint: If an *equivalence_object* is of a derived type that is not a numeric sequence or character sequence type, all of the objects in the equivalence set must be of the same type.

Constraint: If an *equivalence_object* is of an intrinsic type other than default integer, default real, double-precision real, default complex, default logical, or default character, all of the objects in the equivalence set must be of the same type with the same kind type parameter value.

Constraint: The name of an *equivalence_object* must not be a name made accessible by USE association.

Constraint: A *substring* must not have length zero.

R549	*common_stmt*	**is**	COMMON [/ [*common_block_name*] /] *common_block_object_list* [[,] / [*common_block_name*] / *common_block_object_list*] ...
R550	*common_block_object*	**is**	*variable_name* [(*explicit_shape_spec_list*)]

Constraint: Only one appearance of a given *variable_name* is permitted in all *common_block_object_list*s within a scoping unit.

Constraint: A *common_block_object* must not be a dummy argument, an allocatable array, an automatic object, a function name, an entry name, or a result name.

Constraint: Each bound in the *explicit_shape_spec* must be a constant specification expression.

Constraint: If a *common_block_object* is of a derived type, it must be a sequence type with no default initialization.

Constraint: If a *variable_name* appears with an *explicit_shape_spec_list*, it must not have the `POINTER` attribute.

Constraint: A variable name must not be a name made accessible by `USE` association.

1.2.6 Use of Data Objects

The following syntax rules are described in section 6, "Use of data objects," of the Fortran 95 standard.

R601	*variable*	**is**	*scalar_variable_name*
		or	*array_variable_name*
		or	*subobject*

Constraint: *array_variable_name* must be the name of a data object that is an array.

Constraint: *array_variable_name* must not have the `PARAMETER` attribute.

Constraint: *scalar_variable_name* must not have the `PARAMETER` attribute.

Constraint: *subobject* must not be a subobject designator (for example, a substring) whose parent is a constant.

R602	*subobject*	**is**	*array_element*
		or	*array_section*
		or	*structure_component*
		or	*substring*
R603	*logical_variable*	**is**	*variable*

Constraint: *logical_variable* must be of type logical.

R604	*default_logical_variable*	**is**	*variable*

Constraint: *default_logical_variable* must be of type default logical.

R605	*char_variable*	**is**	*variable*

Constraint: *char_variable* must be of type character.

R606	*default_char_variable*	**is**	*variable*

Constraint: *default_char_variable* must be of type default character.

R607	*int_variable*	**is**	*variable*

Constraint: *int_variable* must be of type integer.

R608	*default_int_variable*	**is**	*variable*

Constraint: *default_int_variable* must be of type default integer.

R609	*substring*	**is**	*parent_string* (*substring_range*)
R610	*parent_string*	**is**	*scalar_variable_name*
		or	*array_element*
		or	*scalar_structure_component*
		or	*scalar_constant*
R611	*substring_range*	**is**	[*scalar_int_expr*] : [*scalar_int_expr*]

Constraint: *parent_string* must be of type character.

R612	*data_ref*	**is**	*part_ref* [% *part_ref*] ...
R613	*part_ref*	**is**	*part_name* [(*section_subscript_list*)]

Constraint: In a *data_ref*, each *part_name* except the rightmost must be of derived type.

Constraint: In a *data_ref*, each *part_name* except the leftmost must be the name of a component of the derived type definition of the type of the preceding *part_name*.

Constraint: In a *part_ref* containing a *section_subscript_list*, the number of *section_subscripts* must equal the rank of *part_name*.

Constraint: In a *data_ref*, there must not be more than one *part_ref* with nonzero rank. A *part_name* to the right of a *part_ref* with nonzero rank must not have the `POINTER` attribute.

R614	*structure_component*	**is**	*data_ref*

Constraint: In a *structure_component*, there must be more than one *part_ref* and the rightmost *part_ref* must be of the form *part_name*.

R615	*array_element*	**is**	*data_ref*

Constraint: In an *array_element*, every *part_ref* must have rank zero and the last *part_ref* must contain a *subscript_list*.

R616	*array_section*	**is**	*data_ref* [(*substring_range*)]

Constraint: In an *array_section*, exactly one *part_ref* must have nonzero rank, and either the final *part_ref* has a *section_subscript_list* with nonzero rank or another *part_ref* must have nonzero rank.

Constraint: In an *array_section* with a *substring_range*, the rightmost *part_name* must be of type character.

R617	*subscript*	**is**	*scalar_int_exp*
R618	*section_subscript*	**is**	*subscript*
		or	*subscript_triplet*
		or	*vector_subscript*
R619	*subscript_triplet*	**is**	[*subscript*] : [*subscript*] [: *stride*]
R620	*stride*	**is**	*scalar_int_expr*
R621	*vector_subscript*	**is**	*int_expr*

Constraint: A *vector_subscript* must be an integer array expression of rank one.

Constraint: The second subscript must not be omitted from a *subscript_triplet* in the last dimension of an assumed-size array.

R622	*allocate_stmt*	**is**	ALLOCATE (*allocation_list* [, STAT = *stat_variable*])
R623	*stat_variable*	**is**	*scalar_int_variable*
R624	*allocation*	**is**	*allocate_object* [(*allocate_shape_spec_list*)]
R625	*allocate_object*	**is**	*variable_name*
		or	*structure_component*
R626	*allocate_shape_spec*	**is**	[*allocate_lower_bound* :] *allocate_upper_bound*
R627	*allocate_lower_bound*	**is**	*scalar_int_expr*
R628	*allocate_upper_bound*	**is**	*scalar_int_expr*

Constraint: Each *allocate_object* must be a pointer or an allocatable array.

Constraint: The number of *allocate_shape_spec*s in an *allocate_shape_spec_list* must be the same as the rank of the pointer or allocatable array.

R629	*nullify_stmt*	**is**	NULLIFY (*pointer_object_list*)
R630	*pointer_object*	**is**	*variable_name*
		or	*structure_component*

Constraint: Each *pointer_object* must have the POINTER attribute.

R631	*deallocate_stmt*	**is**	DEALLOCATE (*allocate_object_list* [, STAT = *stat_variable*])

Constraint: Each *allocate_object* must be a pointer or an allocatable array.

1.2.7 Expressions and Assignment

The following syntax rules are described in section 7, "Expressions and assignment," of the Fortran 95 standard.

Note: The language of the Fortran 95 standard is presented in this subsection in its original form. Chapter 7 of the *Fortran Language Reference Manual, Volume 1*, however, sometimes uses terms that are different from those found in the standard. The terminology was changed to improve clarity. The following list shows the terms used in this compiler manual set and the equivalent term used in the Fortran 95 standard.

Standard	SGI term
level_1_expr	*defined_unary_expr*
defined_unary_op	*defined_operator*
mult_operand	*exponentiation_expr*
power_op	**
add_operand	*multiplication_expr*
mult_op	* or /
level_2_expr	*summation_expr*
add_op	+ or -
level_3_expr	*concatenation_expr*
concat_op	//
level_4_expr	*comparison_expr*
rel_op	*rel_op*
and_operand	*not_expr*
not_op	.NOT.

or_operand	*conjunct_expr*
and_op	.AND.
or_op	.OR.
equiv_operand	*inclusive_disjunct_expr*
level_5_expr	*equivalence_expr*
mask_expr	*logical_expr*

R701	*primary*	**is**	*constant*
		or	*constant_subobject*
		or	*variable*
		or	*array_constructor*
		or	*structure_constructor*
		or	*function_reference*
		or	(*expr*)
R702	*constant_subobject*	**is**	*subobject*

Constraint: *subobject* must be a subobject designator whose parent is a constant. A *variable* that is a primary must not be an assumed-size array.

R703	*level_1_expr*	**is**	[*defined_unary_op*] *primary*
R704	*defined_unary_op*	**is**	. *letter* [*letter*] ... .

Constraint: A *defined_unary_op* must not contain more than 31 letters and must not be the same as any *intrinsic_operator* or *logical_literal_constant*.

R705	*mult_operand*	**is**	*level_1_expr* [*power_op mult_operand*]
R706	*add_operand*	**is**	[*add_operand mult_op*] *mult_operand*
R707	*level_2_expr*	**is**	[[*level_2_expr*] *add_op*] *add_operand*

R708	*power_op*	**is**	`**`
R709	*mult_op*	**is**	`*`
		or	`/`
R710	*add_op*	**is**	`+`
		or	`-`
R711	*level_3_expr*	**is**	[*level_3_expr concat_op*] *level_2_expr*
R712	*concat_op*	**is**	`//`
R713	*level_4_expr*	**is**	[*level_3_expr rel_op*] *level_3_expr*
R714	*rel_op*	**is**	`.EQ.`
		or	`.NE.`
		or	`.LT.`
		or	`.LE.`
		or	`.GT.`
		or	`.GE.`
EXT		**or**	`.LG.`
		or	`==`
		or	`/=`
		or	`<`
		or	`<=`
		or	`>`
		or	`>=`
EXT		**or**	`<>`
R715	*and_operand*	**is**	[*not_op*] *level_4_expr*
R716	*or_operand*	**is**	[*or_operand and_op*] *and_operand*
R717	*equiv_operand*	**is**	[*equiv_operand or_op*] *or_operand*
R718	*level_5_expr*	**is**	[*level_5_expr equiv_op*] *equiv_operand*
R719	*not_op*	**is**	`.NOT.`
R720	*and_op*	**is**	`.AND.`
R721	*or_op*	**is**	`.OR.`

R722	*equiv_op*	**is**	`.EQV.`
		or	`.NEQV.`
R723	*expr*	**is**	[*expr defined_binary_op*] *level_5_expr*
R724	*defined_binary_op*	**is**	`.` *letter* [*letter*] `...` `.`

Constraint: A *defined_binary_op* must not contain more than 31 letters and must not be the same as any *intrinsic_operator* or *logical_literal_constant*.

R725	*logical_expr*	**is**	*expr*

Constraint: *logical_expr* must be type logical.

R726	*char_expr*	**is**	*expr*

Constraint: *char_expr* must be type character.

R727	*default_char_expr*	**is**	*expr*

Constraint: *default_char_expr* must be of type default character.

R728	*int_expr*	**is**	*expr*

Constraint: *int_expr* must be type integer.

R729	*numeric_expr*	**is**	*expr*

Constraint: *numeric_expr* must be of type integer, real, or complex.

R730	*initialization_expr*	**is**	*expr*

Constraint: An *initialization_expr* must be an initialization expression.

R731	*char_initialization_expr*	**is**	*char_expr*

Constraint: A *char_initialization_expr* must be an initialization expression.

R732	*int_initialization_expr*	**is**	*int_expr*

Constraint: An *int_initialization_expr* must be an initialization expression.

R733	*logical_initialization_expr*	**is**	*logical_expr*

Constraint: A *logical_initialization_expr* must be an initialization expression.

R734	*specification_expr*	**is**	*scalar_int_expr*

Constraint: The *scalar_int_expr* must be a restricted expression.

R735	*assignment_stmt*	**is**	*variable* = *expr*

Constraint: A variable in an *assignment_stmt* must not be an assumed-size array.

R736	*pointer_assignment_stmt*	**is**	*pointer_object* => *target*
R737	*target*	**is**	*variable*
		or	*expr*

Constraint: The *pointer_object* must have the `POINTER` attribute.

Constraint: The variable must have the TARGET attribute or be a subobject of an object with the TARGET attribute, or it must have the POINTER attribute.

Constraint: The *target* must be of the same type, kind type parameters, and rank as the pointer.

Constraint: The *target* must not be an array section with a vector subscript.

Constraint: The *expr* must deliver a pointer result.

R738	*where_stmt*	**is**	WHERE (*mask_expr*) *where_assignment_stmt*
R739	*where_construct*	**is**	*where_construct_stmt* [*where_body_construct*] ... [*masked_elsewhere_stmt* [*where_body_construct*] ...] ... [*elsewhere_stmt* [*where_body_construct*] ...] *end_where_stmt*
R740	*where_construct_stmt*	**is**	[*where_construct_name:*]WHERE (*mask_expr*)
R741	*where_construct_stmt*	**is**	*where_assignment_stmt*
		or	*where_stmt*
		or	*where_construct*
R742	*where_assignment_stmt*	**is**	*assignment_stmt*
R743	*mask_expr*	**is**	*logical_expr*
R742	*masked_elsewhere_stmt*	**is**	ELSEWHERE (*mask_expr*) [*where_construct_name*]
R745	*elsewhere_stmt*	**is**	ELSEWHERE [*where_construct_name*]
R746	*end_where_stmt*	**is**	END WHERE [*where_construct_name*]

Constraint: A *where_assignment_stmt* that is a defined assignment must be elemental.

Constraint: If the *where_construct_stmt* is identified by a *where_construct_name*, the corresponding *end_where_stmt* must specify the same *where_construct_name*. If the *where_construct_stmt* is not identified by a *where_construct_name*, the corresponding *end_where_stmt* must not specify a *where_construct_name*. If an *elsewhere_stmt* or a *masked_elsewhere_stmt* is identified by a *where_construct_name*,

the corresponding *where_construct_stmt* must specify the same *where_construct_name*.

R747	*forall_construct*	**is**	*forall_construct_stmt* [*forall_body_construct*] ... *end_forall_stmt*
R748	*forall_construct_stmt*	**is**	[*forall_construct_name:*]FORALL *forall_header*
R749	*forall_header*	**is**	(*forall_triplet_spec_list* [, *scalar_mask_expr*])
R750	*forall_triplet_spec*	**is**	*index_name* = *subscript* : *subscript* [: *stride*]
R617	*subscript*	**is**	*scalar_int_expr*
R603	*stride*	**is**	*scalar_int_expr*
R751	*forall_body_construct*	**is**	*forall_assignment_stmt*
R751	*forall_body_construct*	**is**	*forall_assignment_stmt*
		or	*where_stmt*
		or	*where_construct*
		or	*forall_construct*
		or	*forall_stmt*
R752	*forall_assignment_stmt*	**is**	*assignment_stmt*
		or	*pointer_assignment_stmt*
R753	*end_forall_stmt*	**is**	END FORALL [*forall_construct_name*]

Constraint: If the *forall_construct_stmt* has a *forall_construct_name*, the *end_forall_stmt* must have the same *forall_construct_name*. If the *end_forall_stmt* has a *forall_construct_name*, the *forall_construct_stmt* must have the same *forall_construct_name*.

Constraint: The *scalar_mask_expr* must be scalar and of type logical.

Constraint: Any procedure referenced in the *scalar_mask_expr*, including one referenced by a defined operation, must be a pure procedure.

Constraint: The *index_name* must be a named scalar variable of type integer.

Constraint: A *subscript* or *stride* in a *forall_triplet_spec* must not contain a reference to any *index_name* in the *forall_triplet_spec_list* in which it appears.

Constraint: A statement in a *forall_body_construct* must not define an *index_name* of the *forall_construct*.

Constraint: Any procedure referenced in a *forall_body_construct*, including one referenced by a defined operation or assignment, must be a pure procedure.

Constraint: A *forall_body_construct* must not be a branch target.

R754	*forall_stmt*	**is**	FORALL *forall_header forall_assignment_stmt*

1.2.8 Execution Control

The following syntax rules are described in section 8, "Execution control," of the Fortran 95 standard.

R801	*block*	**is**	[*execution_part_construct*] ...
R802	*if_construct*	**is**	*if_then_stmt* *block* [*else_if_stmt* *block*] ... [*else_stmt* *block*] *end_if_stmt*
R803	*if_then_stmt*	**is**	[*if_construct_name* :] IF (*scalar_logical_expr*) THEN
R804	*else_if_stmt*	**is**	ELSE IF (*scalar_logical_expr*) THEN [*if_construct_name*]
R805	*else_stmt*	**is**	ELSE [*if_construct_name*]
R806	*end_if_stmt*	**is**	END IF [*if_construct_name*]

Constraint: If the *if_then_stmt* of an *if_construct* is identified by an *if_construct_name*, the corresponding *end_if_stmt* must specify the same *if_construct_name*. If the *if_then_stmt* of an *if_construct* is not identified by an *if_construct_name*, the corresponding *end_if_stmt* must not specify an *if_construct_name*. If an *else_if_stmt* or *else_stmt* is identified by an *if_construct_name*, the corresponding *if_then_stmt* must specify the same *if_construct_name*.

R807	*if_stmt*	**is**	`IF (`*scalar_logical_expr*`)` *action_stmt*

Constraint: The *action_stmt* in the *if_stmt* must not be an *if_stmt*, *end_program_stmt*, *end_function_stmt*, or *end_subroutine_stmt*.

R808	*case_construct*	**is**	*select_case_stmt* [*case_stmt* *block*] ... *end_select_stmt*
R809	*select_case_stmt*	**is**	[*case_construct_name*] `: SELECT CASE (` *case_expr* `)`
R810	*case_stmt*	**is**	`CASE` *case_selector* [*case_construct_name*]
R811	*end_select_stmt*	**is**	`END SELECT` [*case_construct_name*]

Constraint: If the *select_case_stmt* of a *case_construct* is identified by a *case_construct_name*, the corresponding *end_select_stmt* must specify the same *case_construct_name*. If the *select_case_stmt* of a *case_construct* is not identified by a *case_construct_name*, the corresponding *end_select_stmt* must not specify a *case_construct_name*. If a *case_stmt* is identified by a *case_construct_name*, the corresponding *select_case_stmt* must specify the same *case_construct_name*.

R812	*case_expr*	**is**	*scalar_int_expr*
		or	*scalar_char_expr*
		or	*scalar_logical_expr*
R813	*case_selector*	**is**	`(` *case_value_range_list* `)`
		or	`DEFAULT`

Constraint: No more than one of the selectors of one of the `CASE` statements may be `DEFAULT`.

R814	*case_value_range*	**is**	*case_value*
		or	*case_value* :
		or	: *case_value*
		or	*case_value* : *case_value*
R815	*case_value*	**is**	*scalar_int_initialization_expr*
		or	*scalar_char_initialization_expr*
		or	*scalar_logical_initialization_expr*

Constraint: For a given *case_construct*, each *case_value* must be of the same type as *case_expr*. For character type, length differences are allowed, but the kind type parameters must be the same.

Constraint: A *case_value_range* using a colon must not be used if *case_expr* is of type logical.

Constraint: For a given *case_construct*, the *case_value_range*s must not overlap; that is, there must be no possible value of the *case_expr* that matches more than one *case_value_range*.

R816	*do_construct*	**is**	*block_do_construct*
		or	*nonblock_do_construct*
R817	*block_do_construct*	**is**	*do_stmt* *do_block* *end_do*
R818	*do_stmt*	**is**	*label_do_stmt*
		or	*nonlabel_do_stmt*
R819	*label_do_stmt*	**is**	[*do_construct_name* :] DO *label* [*loop_control*]
R820	*nonlabel_do_stmt*	**is**	[*do_construct_name* :] DO [*loop_control*]
R821	*loop_control*	**is**	[,] *do_variable* = *scalar_int_expr*,
		or	*scalar_int_expr* [, *scalar_int_expr*]
			[,] WHILE (*scalar_logical_expr*)
R822	*do_variable*	**is**	*scalar_int_variable*

Constraint: The *do_variable* must be a named scalar variable of type integer, default real, or double-precision real. The Fortran standard does not allow for *do_variable*s of type default real or double-precision real.

Constraint: Each *scalar_numeric_expr* in *loop_control* must be of type integer, default real, or double-precision real. The Fortran standard does not allow for *do_variable*s of type default real or double-precision real.

R823	*do_block*	**is**	*block*
R824	*end_do*	**is**	*end_do_stmt*
		or	*continue_stmt*
R825	*end_do_stmt*	**is**	`END DO` [*do_construct_name*]

Constraint: If the *do_stmt* of a *block_do_construct* is identified by a *do_construct_name*, the corresponding *end_do* must be an *end_do_stmt* specifying the same *do_construct_name*. If the *do_stmt* of a *block_do_construct* is not identified by a *do_construct_name*, the corresponding *end_do* must not specify a *do_construct_name*.

Constraint: If the *do_stmt* is a *nonlabel_do_stmt*, the corresponding *end_do* must be an *end_do_stmt*.

Constraint: If the *do_stmt* is a *label_do_stmt*, the corresponding *end_do* must be identified with the same label.

OBS	*nonblock_do_construct*	**is**	*action_term_do_construct*
		or	*outer_shared_do_construct*
OBS	*action_term_do_construct*	**is**	*label_do_stmt* *do_body* *do_term_action_stmt*
OBS	*do_body*	**is**	[*execution_part_construct*] ...
OBS	*do_term_action_stmt*	**is**	*action_stmt*

Obsolescent Constraint: A *do_term_action_stmt* must not be a *continue_stmt*, a *goto_stmt*, a *return_stmt*, a *stop_stmt*, an *exit_stmt*, a *cycle_stmt*, an

end_function_stmt, an *end_subroutine_stmt*, an *end_program_stmt*, or an *arithmetic_if_stmt*.

Obsolescent Constraint: The *do_term_action_stmt* must be identified with a label and the corresponding *label_do_stmt* must refer to the same label.

OBS	*outer_shared_do_construct*	**is**	*label_do_stmt* *do_body* *shared_term_do_construct*
OBS	*shared_term_do_construct*	**is**	*outer_shared_do_construct*
		or	*inner_shared_do_construct*
OBS	*inner_shared_do_construct*	**is**	*label_do_stmt* *do_body* *do_term_shared_stmt*
OBS	*do_term_shared_stmt*	**is**	*action_stmt*

Obsolescent Constraint: A *do_term_shared_stmt* must not be a *goto_stmt*, a *return_stmt*, a *stop_stmt*, an *exit_stmt*, a *cycle_stmt*, an *end_function_stmt*, an *end_subroutine_stmt*, an *end_program_stmt*, or an *arithmetic_if_stmt*.

Obsolescent Constraint: The *do_term_shared_stmt* must be identified with a label, and all of the *label_do_stmts* of the *shared_term_do_construct* must refer to the same label.

R834	*cycle_stmt*	**is**	`CYCLE` [*do_construct_name*]

Constraint: If a *cycle_stmt* refers to a *do_construct_name*, it must be within the range of that *do_construct*; otherwise, it must be within the range of at least one *do_construct*.

R835	*exit_stmt*	**is**	`EXIT` [*do_construct_name*]

Constraint: If an *exit_stmt* refers to a *do_construct_name*, it must be within the range of that *do_construct*; otherwise, it must be within the range of at least one *do_construct*.

R836	*goto_stmt*	**is**	`GO TO` *label*

Constraint: The *label* must be the statement label of a branch target statement that appears in the same scoping unit as the *goto_stmt*.

OBS	*computed_goto_stmt*	**is**	`GO TO (` *label_list* `) [ , ]` *scalar_int_expr*

Obsolescent Constraint: Each *label* in *label_list* must be the statement label of a branch target statement that appears in the same scoping unit as the *computed_goto_stmt*.

EXT	*assign_stmt*	**is**	`ASSIGN` *label* `TO` *scalar_int_variable*

Extension Constraint: The *label* must be the statement label of a branch target statement or *format_stmt* that appears in the same scoping unit as the *assign_stmt*.

Extension Constraint: *scalar_int_variable* must be named and of type default integer.

EXT	*assigned_goto_stmt*	**is**	`GO TO` *scalar_int_variable* `[ [ , ] (`*label_list*`) ]`

Extension Constraint: Each *label* in *label_list* must be the statement label of a branch target statement that appears in the same scoping unit as the *assigned_goto_stmt*.

Extension Constraint: *scalar_int_variable* must be named and of type default integer.

OBS	*arithmetic_if_stmt*	**is**	`IF (` *scalar_numeric_expr* `)` *label*`,` *label*`,` *label*

Obsolescent Constraint: Each *label* must be the label of a branch target statement that appears in the same scoping unit as the *arithmetic_if_stmt*.

Obsolescent Constraint: The *scalar_numeric_expr* must not be of type complex.

R839	*continue_stmt*	**is**	CONTINUE
R840	*stop_stmt*	**is**	STOP [*stop_code*]
R841	*stop_code*	**is**	*scalar_char_constant*
EXT		**or**	*digit* [*digit* [*digit* [*digit* [*digit*]]]]

Constraint: *scalar_char_constant* must be of type default character.

EXT	*pause_stmt*	**is**	PAUSE [*stop_code*]

1.2.9 Input/Output (I/O) Statements

The following syntax rules are described in section 9, "Input/Output statements," of the Fortran 95 standard.

R901	*io_unit*	**is**	*external_file_unit*
		or	*
		or	*internal_file_unit*
EXT		**or**	*unit_name*
R902	*external_file_unit*	**is**	*scalar_int_expr*
R903	*internal_file_unit*	**is**	*default_char_variable*

Constraint: The *default_char_variable* must not be an array section with a vector subscript.

R904	*open_stmt*	**is**	`OPEN (` *connect_spec_list* `)`
R905	*connect_spec*	**is**	`[ UNIT = ]` *external_file_unit*
		or	`IOSTAT =` *scalar_default_int_variable*
		or	`ERR =` *label*
		or	`FILE =` *file_name_expr*
		or	`STATUS =` *scalar_char_expr*
		or	`ACCESS =` *scalar_char_expr*
		or	`FORM =` *scalar_char_expr*
		or	`RECL =` *scalar_int_expr*
		or	`BLANK =` *scalar_char_expr*
		or	`POSITION =` *scalar_char_expr*
		or	`ACTION =` *scalar_char_expr*
		or	`DELIM =` *scalar_char_expr*
		or	`PAD =` *scalar_char_expr*
R906	*file_name_expr*	**is**	*scalar_char_expr*

Constraint: If the optional characters `UNIT=` are omitted from the unit specifier, the unit specifier must be the first item in the *connect_spec_list*.

Constraint: Each specifier must not appear more than once in a given *open_stmt*; an *external_file_unit* must be specified.

Constraint: The *label* used in the `ERR=` specifier must be the statement label of a branch target statement that appears in the same scoping unit as the `OPEN` statement.

R907	*close_stmt*	**is**	`CLOSE (` *close_spec_list* `)`
R908	*close_spec*	**is**	`[ UNIT = ]` *external_file_unit*
		or	`IOSTAT =` *scalar_default_int_variable*
		or	`ERR =` *label*
		or	`STATUS =` *scalar_char_expr*

Constraint: If the optional characters UNIT= are omitted from the unit specifier, the unit specifier must be the first item in the *close_spec_list*.

Constraint: Each specifier must not appear more than once in a given *close_stmt*; an *external_file_unit* must be specified.

Constraint: The label used in the ERR= specifier must be the statement label of a branch target statement that appears in the same scoping unit as the CLOSE statement.

R909	*read_stmt*	**is**	READ (*io_control_spec_list*) [*input_item_list*]
EXT		**or**	READ *format* [, *input_item_list*]
R910	*write_stmt*	**is**	WRITE (*io_control_spec_list*) [*output_item_list*]
EXT		**or**	WRITE *format* [, *output_item_list*]
R911	*print_stmt*	**is**	PRINT *format* [, *output_item_list*]
R912	*io_control_spec*	**is**	[UNIT =] *io_unit*
		or	[FMT =] *format*
		or	[NML =] *namelist_group_name*
		or	REC = *scalar_int_expr*
		or	IOSTAT = *scalar_default_int_variable*
		or	ERR = *label*
		or	END = *label*
		or	ADVANCE = *scalar_default_char_expr*
		or	SIZE = *scalar_default_int_variable*
		or	EOR = *label*

Constraint: An *io_control_spec_list* must contain exactly one *io_unit* and may contain at most one of each of the other specifiers.

Constraint: An END=, EOR=, or SIZE= specifier must not appear in a *write_stmt*.

Constraint: The label in the ERR=, EOR=, or END= specifier must be the statement label of a branch target statement that appears in the same scoping unit as the data transfer statement.

Constraint: A *namelist_group_name* must not be present if an *input_item_list* or an *output_item_list* is present in the data transfer statement.

Constraint: An *io_control_spec_list* must not contain both a format and a *namelist_group_name*.

Constraint: If the optional characters `UNIT=` are omitted from the unit specifier, the unit specifier must be the first item in the control information list.

Constraint: If the optional characters `FMT=` are omitted from the format specifier, the format specifier must be the second item in the control information list and the first item must be the unit specifier without the optional characters `UNIT=`.

Constraint: If the optional characters `NML=` are omitted from the namelist specifier, the namelist specifier must be the second item in the control information list and the first item must be the unit specifier without the optional characters `UNIT=`.

Constraint: If the unit specifier specifies an internal file, the *io_control_spec_list* must not contain a `REC=` specifier or a *namelist_group_name*.

Constraint: If the `REC=` specifier is present, an `END=` specifier must not appear, a *namelist_group_name* must not appear, and the format, if any, must not be an asterisk specifying *list_directed* I/O.

Constraint: An `ADVANCE=` specifier may be present only in a formatted sequential I/O statement with explicit format specification whose control information list does not contain an internal file unit specifier.

Constraint: If an `EOR=` specifier is present, an `ADVANCE=` specifier also must appear.

Constraint: If a `SIZE=` specifier is present, an `ADVANCE=` specifier must also appear.

R913	*format*	**is**	*default_char_expr*
		or	*label*
		or	*
EXT		**or**	*scalar_default_int_variable*

Constraint: The *label* must be the label of a `FORMAT` statement that appears in the same scoping unit as the statement containing the format specifier.

R914	*input_item*	**is**	*variable*
		or	*io_implied_do*
R915	*output_item*	**is**	*expr*
		or	*io_implied_do*
R916	*io_implied_do*	**is**	(*io_implied_do_object_list*, *io_implied_do_control*)
R917	*io_implied_do_object*	**is**	*input_item*
		or	*output_item*
R918	*io_implied_do_control*	**is**	*do_variable* = *scalar_int_expr*, *scalar_int_expr* [, *scalar_int_expr*]

Constraint: A variable that is an *input_item* must not be an assumed-size array.

Constraint: The DO variable must be a named scalar of type integer.

Constraint: In an *input_item_list*, an *io_implied_do_object* must be an *input_item*. In an *output_item_list*, an *io_implied_do_object* must be an *output_item*.

EXT	*buffer_in_stmt*	**is**	BUFFER IN (*io_unit*, *mode*) (*start_loc*, *end_loc*)
EXT	*buffer_out_stmt*	**is**	BUFFER OUT (*io_unit*, *mode*) (*start_loc*, *end_loc*)
EXT	*io_unit*	**is**	*external_file_unit*
		or	*file_name_expr*
EXT	*mode*	**is**	*scalar_integer_expr*
EXT	*start_loc*	**is**	*variable*
EXT	*end_loc*	**is**	*variable*

In the preceding definition, the *variable* specified for *start_loc* and *end_loc* cannot be of a derived type if you are performing implicit data conversion. The data items between *start_loc* and *end_loc* must be of the same type and same kind type.

R919	*backspace_stmt*	**is**	`BACKSPACE` *external_file_unit*
		or	`BACKSPACE (` *position_spec_list* `)`
R920	*endfile_stmt*	**is**	`ENDFILE` *external_file_unit*
		or	`ENDFILE (` *position_spec_list* `)`
R921	*rewind_stmt*	**is**	`REWIND` *external_file_unit*
		or	`REWIND (` *position_spec_list* `)`
R922	*position_spec*	**is**	`[ UNIT = ]` *external_file_unit*
		or	`IOSTAT =` *scalar_default_int_variable*
		or	`ERR =` *label*

Constraint: The *label* in the `ERR=` specifier must be the statement label of a branch target statement that appears in the same scoping unit as the file positioning statement.

Constraint: If the optional characters `UNIT=` are omitted from the unit specifier; the unit specifier must be the first item in the *position_spec_list*.

Constraint: A *position_spec_list* must contain exactly one *external_file_unit* and may contain at most one of each of the other specifiers.

R923	*inquire_stmt*	**is**	`INQUIRE (` *inquire_spec_list* `)`
		or	`INQUIRE ( IOLENGTH =` *scalar_default_int_variable* `)` *output_item_list*
R924	*inquire_spec*	**is**	`[ UNIT = ]` *external_file_unit*
		or	`FILE =` *file_name_expr*
		or	`IOSTAT =` *scalar_default_int_variable*
		or	`ERR =` *label*
		or	`EXIST =` *scalar_default_logical_variable*
		or	`OPENED =` *scalar_default_logical_variable*
		or	`NUMBER =` *scalar_default_int_variable*
		or	`NAMED =` *scalar_default_logical_variable*
		or	`NAME =` *scalar_default_ char_variable*
		or	`ACCESS =` *scalar_default_char_variable*

or	`SEQUENTIAL =` *scalar_default_char_variable*
or	`DIRECT =` *scalar_default_char_variable*
or	`FORM =` *scalar_default_char_variable*
or	`FORMATTED =` *scalar_default_char_variable*
or	`UNFORMATTED =` *scalar_default_char_variable*
or	`RECL =` *scalar_default_int_variable*
or	`NEXTREC =` *scalar_default_int_variable*
or	`BLANK =` *scalar_default_char_variable*
or	`POSITION =` *scalar_default_char_variable*
or	`ACTION =` *scalar_default_char_variable*
or	`READ =` *scalar_default_char_variable*
or	`WRITE =` *scalar_default_char_variable*
or	`READWRITE =` *scalar_default_char_variable*
or	`DELIM =` *scalar_default_char_variable*
or	`PAD =` *scalar_default_char_variable*

Constraint: An *inquire_spec_list* must contain one `FILE=` specifier or one `UNIT=` specifier, but not both, and at most one of each of the other specifiers.

Constraint: In the inquire by unit form of the `INQUIRE` statement, if the optional characters `UNIT=` are omitted from the unit specifier, the unit specifier must be the first item in the *inquire_spec_list*.

1.2.10 I/O Editing

The following syntax rules are described in section 10, "Input/Output editing," of the Fortran 95 standard.

R1001	*format_stmt*	**is**	`FORMAT` *format_specification*
R1002	*format_specification*	**is**	`(` [*format_item_list*] `)`

Constraint: The *format_stmt* must be labeled.

Constraint: The comma used to separate *format_items* in a *format_item_list* may be omitted as follows:

- Between a P edit descriptor and an immediately following F, E, EN, ES, D, or G edit descriptor
- Before a slash edit descriptor when the optional repeat specification is not present
- After a slash edit descriptor
- Before or after a colon edit descriptor

R1003	*format_item*	**is**	[*r*] *data_edit_desc*
		or	*control_edit_desc*
		or	*char_string_edit_desc*
		or	[*r*] (*format_item_list*)
R1004	*r*	**is**	*int_literal_constant*

Constraint: *r* must be positive.

Constraint: *r* must not have kind parameter specified for it.

R1005	*data_edit_desc*	**is**	I *w* [. *m*]
		or	B *w* [. *m*]
		or	O *w* [. *m*]
		or	Z *w* [. *m*]
		or	F *w* . *d*
		or	E *w* . *d* [E *e*]
		or	EN *w* . *d* [E *e*]
		or	ES *w* . *d* [E *e*]
		or	G *w* . *d* [E *e*]
		or	L *w*

		or	`A` [*w*]
		or	`D` *w* `.` *d*
EXT		**or**	`D` *w* `.` *d* `E` *e*
EXT		**or**	`R` *w*
EXT		**or**	`Q`
R1006	*w*	**is**	*int_literal_constant*
R1007	*m*	**is**	*int_literal_constant*
R1008	*d*	**is**	*int_literal_constant*
R1009	*e*	**is**	*int_literal_constant*

Constraint: *e* must be positive.

Constraint: *w* must be zero or positive for the `I`, `B`, `O`, `Z`, and `F` edit descriptors. *w* must be positive for all other edit descriptors.

Constraint: *w*, *m*, *d*, and *e* must not have kind parameters specified for them.

R1010	*control_edit_desc*	**is**	*position_edit_desc*
		or	[*r*] `/`
		or	`:`
		or	*sign_edit_desc*
		or	*k* `P`
		or	*blank_interp_edit_desc*
R1011	*k*	**is**	*signed_int_literal_constant*

Constraint: *k* must not have a kind parameter specified for it.

R1012	*position_edit_desc*	**is**	`T` *n*
		or	`TL` *n*
		or	`TR` *n*
		or	*n* `X`

EXT		**or**	\
EXT		**or**	$
R1013	*n*	**is**	*int_literal_constant*

Constraint: *n* must be positive.

Constraint: *n* must not have a kind parameter specified for it.

R1014	*sign_edit_desc*	**is**	S
		or	SP
		or	SS
R1015	*blank_interp_edit_desc*	**is**	BN
		or	BZ
R1016	*char_string_edit_desc*	**is**	*char_literal_constant*
EXT		**or**	*c* H *rep_char* [*rep_char*] ...
EXT	*c*	**is**	*int_literal_constant*

Constraint: The *char_literal_constant* must not have a kind parameter specified for it.

1.2.11 Program Units

The following syntax rules are described in section 11, "Program units," of the Fortran 95 standard.

R1101	*main_program*	**is**	[*program_stmt*] [*specification_part*] [*execution_part*] [*internal_subprogram_part*] *end_program_stmt*
R1102	*program_stmt*	**is**	PROGRAM *program_name* [(*args*)]
EXT	*args*	**is**	Any character in the CF90 character

			set. The CF90 compiler ignores any *args* specified after *program_name*.
R1103	*end_program_stmt*	**is**	END [PROGRAM [*program_name*]]

Constraint: In a *main_program*, the *execution_part* must not contain a RETURN statement or an ENTRY statement.

Constraint: The *program_name* may be included in the *end_program_stmt* only if the optional *program_stmt* is used and, if included, must be identical to the *program_name* specified in the *program_stmt*.

Constraint: An automatic object must not appear in the *specification_part* of a main program.

R1104	*module*	**is**	*module_stmt* [*specification_part*] [*module_subprogram_part*] *end_module_stmt*
R1105	*module_stmt*	**is**	MODULE *module_name*
R1106	*end_module_stmt*	**is**	END [MODULE [*module_name*]]

Constraint: If the *module_name* is specified in the *end_module_stmt*, it must be identical to the *module_name* specified in the *module_stmt*.

Constraint: A module *specification_part* must not contain a *stmt_function_stmt*, an *entry_stmt*, or a *format_stmt*.

Constraint: An automatic object must not appear in the *specification_part* of a module.

Constraint: If an object of a type for which *component_initialization* is specified appears in the *specification_part* of a module and does not have the ALLOCATABLE or POINTER attribute, the object must have the SAVE attribute.

R1107	*use_stmt*	**is**	USE *module_name* [, *rename_list*]
		or	USE *module_name*, ONLY : [*only_list*]
R1108	*rename*	**is**	*local_name* => *use_name*

R1109	*only*	**is**	*generic_spec*
		or	*only_use_name*
		or	*only_rename*
R1110	*only_use_name*	**is**	*use_name*
R1111	*only_rename*	**is**	*local_name* `=>` *use_name*

Constraint: Each *generic_spec* must be a public entity in the module.

Constraint: Each *use_name* must be the name of a public entity in the module.

R1112	*block_data*	**is**	*block_data_stmt* [*specification_part*] *end_block_data_stmt*
R1113	*block_data_stmt*	**is**	`BLOCK DATA` [*block_data_name*]
R1114	*end_block_data_stmt*	**is**	`END` [`BLOCK DATA` [*block_data_name*]]

Constraint: The *block_data_name* may be included in the *end_block_data_stmt* only if it was provided in the *block_data_stmt* and, if included, must be identical to the *block_data_name* in the *block_data_stmt*.

Constraint: A *block_data specification_part* may contain only `USE` statements, type declaration statements, `IMPLICIT` statements, `PARAMETER` statements, derived-type definitions, and the following specification statements: `COMMON`, `DATA`, `DIMENSION`, `EQUIVALENCE`, `INTRINSIC`, `POINTER`, `SAVE`, and `TARGET`.

Constraint: A type declaration statement in a *block_data specification_part* must not contain `ALLOCATABLE`, `EXTERNAL`, `INTENT`, `OPTIONAL`, `PRIVATE`, or `PUBLIC` attribute specifiers.

1.2.12 Procedures

The following syntax rules are described in section 12, "Procedures," of the Fortran 95 standard.

R1201	*interface_block*	**is**	*interface_stmt* [*interface_specification*] ... *end_interface_stmt*
R1202	*interface_specification*	**is**	*interface_body*
		or	*module_procedure_stmt*
R1203	*interface_stmt*	**is**	`INTERFACE` [*generic_spec*]
R1204	*end_interface_stmt*	**is**	`END INTERFACE` [*generic_spec*]
R1204	*interface_body*	**is**	*function_stmt* [*specification_part*] *end_function_stmt*
		or	*subroutine_stmt* [*specification_part*] *end_subroutine_stmt*
R1206	*module_procedure_stmt*	**is**	`MODULE PROCEDURE` *procedure_name_list*
R1207	*generic_spec*	**is**	*generic_name*
		or	`OPERATOR (` *defined_operator* `)`
		or	`ASSIGNMENT ( = )`

Constraint: An interface body of a pure procedure must specify the intents of all dummy procedures except pointer, alternate return, and procedure arguments.

Constraint: An *interface_body* must not contain an *entry_stmt*, *data_stmt*, *format_stmt*, or *stmt_function_stmt*.

Constraint: The `MODULE PROCEDURE` statement is allowed only if the *interface_block* has a *generic_spec* and is in a scoping unit where each *procedure_name* is accessible as a module procedure.

Constraint: An *interface_block* in a subprogram must not contain an *interface_body* for a procedure defined by that subprogram.

Constraint: A *procedure_name* in a *module_procedure_stmt* must not be one that previously had been established to be associated with the *generic_spec* of the *interface_block* in which it appears, either by a previous appearance in an *interface_block* or by use or host association.

Constraint: The *generic_spec* can be included in the *end_interface_stmt* only if it was provided in the *interface_stmt*. If included, it must be identical to the *generic_spec* in the *interface_stmt*.

R1208	*external_stmt*	**is**	`EXTERNAL [::]` *external_name_list*
R1209	*intrinsic_stmt*	**is**	`INTRINSIC [::]` *intrinsic_procedure_name_list*

Constraint: Each *intrinsic_procedure_name* must be the name of an intrinsic procedure.

R1210	*function_reference*	**is**	*function_name* `([` *actual_arg_spec_list* `])`

Constraint: The *actual_arg_spec_list* for a function reference must not contain an *alt_return_spec*.

R1211	*call_stmt*	**is**	`CALL` *subroutine_name* `[([` *actual_arg_spec_list* `])]`
R1212	*actual_arg_spec*	**is**	`[` *keyword* `=` `]` *actual_arg*
R1213	*keyword*	**is**	*dummy_arg_name*
R1214	*actual_arg*	**is**	*expr*
		or	*variable*
		or	*procedure_name*
OBS		**or**	*alt_return_spec*
R1215	*alt_return_spec*	**is**	`*` *label*

Constraint: The *keyword* = must not appear if the interface of the procedure is implicit in the scoping unit.

Constraint: The *keyword* = may be omitted from an *actual_arg_spec* only if the *keyword* = has been omitted from each preceding *actual_arg_spec* in the argument list.

Constraint: Each *keyword* must be the name of a dummy argument in the explicit interface of the procedure.

Constraint: A *procedure_name actual_arg* must not be the name of an internal procedure or of a statement function and must not be the generic name of a procedure.

Constraint: The *label* used in the *alt_return_spec* must be the statement label of a branch target statement that appears in the same scoping unit as the *call_stmt*.

Constraint: A nonintrinsic elemental procedure must not be used as an actual argument.

Constraint: In a reference to a pure procedure, a *procedure_name actual_arg* must be the name of a pure procedure.

R1216	*function_subprogram*	**is**	*function_stmt* [*specification_part*] [*execution_part*] [*internal_subprogram_part*] *end_function_stmt*
R1217	*function_stmt*	**is**	[*prefix*] `FUNCTION` *function_name* `(` [*dummy_arg_name_list*] `)` [`RESULT (` *result_name* `)`]

Constraint: If `RESULT` is specified, the *function_name* must not appear in any specification statement in the scoping unit of the function subprogram.

Constraint: A *prefix* must contain at most one of each *prefix_spec*.

Constraint: If `ELEMENTAL` is present, `RECURSIVE` must not be present.

R1218	*prefix*	**is**	*prefix_spec* [*prefix_spec*] ...
R1219	*prefix_spec*	**is**	*type_spec*
		or	`RECURSIVE`
		or	`PURE`
		or	`ELEMENTAL`
R1220	*end_function_stmt*	**is**	`END` [`FUNCTION` [*function_name*]]

Constraint: If RESULT is specified, *result_name* must not be the same as *function_name*.

Constraint: FUNCTION must be present on the *end_function_stmt* of an internal or module function.

Constraint: An internal function subprogram must not contain an ENTRY statement.

Constraint: An internal function subprogram must not contain an *internal_subprogram_part*.

Constraint: If a *function_name* is present on the *end_function_stmt*, it must be identical to the *function_name* specified in the *function_stmt*.

R1221	*subroutine_subprogram*	**is**	*subroutine_stmt* [*specification_part*] [*execution_part*] [*internal_subprogram_part*] *end_subroutine_stmt*
R1222	*subroutine_stmt*	**is**	[*prefix*] SUBROUTINE *subroutine_name* [([*dummy_arg_list*])]
R1223	*dummy_arg*	**is**	*dummy_arg_name*
		or	*
R1224	*end_subroutine_stmt*	**is**	END [SUBROUTINE [*subroutine_name*]]

Constraint: The prefix of a *subroutine_stmt* must not contain a *type_spec*.

Constraint: SUBROUTINE must be present on the *end_subroutine_stmt* of an internal or module subroutine.

Constraint: An internal subroutine must not contain an ENTRY statement.

Constraint: An internal subroutine must not contain an *internal_subprogram_part*.

Constraint: If a *subroutine_name* is present on the *end_subroutine_stmt*, it must be identical to the *subroutine_name* specified in the *subroutine_stmt*.

R1225	*entry_stmt*	**is**	ENTRY *entry_name* [([*dummy_arg_list*]) [RESULT (*result_name*)]]

Constraint: If RESULT is specified, the *entry_name* must not appear in any specification statement in the scoping unit of the function program.

Constraint: An *entry_stmt* may appear only in an *external_subprogram* or *module_subprogram*. An *entry_stmt* must not appear within an *executable_construct*.

Constraint: RESULT may be present only if the *entry_stmt* is contained in a function subprogram.

Constraint: Within the subprogram containing the *entry_stmt*, the *entry_name* must not appear as a dummy argument in the FUNCTION or SUBROUTINE statement or in another ENTRY statement and it must not appear in an EXTERNAL or INTRINSIC statement.

Constraint: A *dummy_arg* can be an alternate return indicator only if the ENTRY statement is in a subroutine subprogram.

Constraint: If RESULT is specified, *result_name* must not be the same as *entry_name*.

R1226	*return_stmt*	**is**	RETURN
OBS		**or**	RETURN [*scalar_int_expr*]

Constraint: The *return_stmt* must be in the scoping unit of a function or subroutine subprogram.

Obsolescent Constraint: The *scalar_int_expr* is allowed only in the scoping unit of a subroutine subprogram.

R1227	*contains_stmt*	**is**	CONTAINS
R1228	*stmt_function_stmt*	**is**	*function_name* ([*dummy_arg_name_list*]) = *scalar_expr*

Obsolescent Constraint: The *primaries* of the *scalar_expr* must be constants (literal and named), references to variables, references to functions and function dummy procedures, and intrinsic operations. If *scalar_expr* contains a reference to a function or function dummy procedure, the reference must not require an explicit interface; the function must not require an explicit interface unless it is an intrinsic; the function must not be a transformational intrinsic; and the result

must be scalar. If an argument to a function or a function dummy procedure is array valued, it must be an array name. If a reference to a statement function appears in *scalar_expr*, its definition must have been provided earlier in the scoping unit and must not be the name of the statement function being defined.

Obsolescent Constraint: Named constants in *scalar_expr* must have been declared earlier in the scoping unit or made accessible by USE or host association. If array elements appear in *scalar_expr*, the parent array must not have been declared as an array earlier in the scoping unit or made accessible by USE or host association.

Obsolescent Constraint: If a *dummy_arg_name*, variable, function reference, or dummy function reference is typed by the implicit typing rules, its appearance in any subsequent type declaration statement must confirm this implied type and the values of any implied type parameters.

Constraint: The *function_name* and each *dummy_arg_name* must be specified, explicitly or implicitly, to be scalar data objects.

Constraint: A given *dummy_arg_name* may appear only once in any *dummy_arg_name_list*.

Constraint: Each scalar variable reference in *scalar_expr* may be either a reference to a dummy argument of the statement function or a reference to a variable local to the same scoping unit as the statement function statement.

Constraint: The *specification_part* of a pure function subprogram must specify that all dummy arguments have `INTENT(IN)` except procedure arguments and arguments with the `POINTER` attribute.

Constraint: The *specification_part* of a pure subroutine subprogram must specify the intents of all dummy arguments except procedure arguments, alternate return indicators, and arguments with the `POINTER` attribute. Note that alternate return indicators are obsolete.

Constraint: A local variable declared in the *specification_part* or *internal_subprogram_part* of a pure subprogram must not have the SAVE attribute.

Constraint: The *specification_part* of a pure subprogram must specify that all dummy arguments that are procedure arguments are pure.

Constraint: If a procedure that is neither an intrinsic procedure nor a statement function is used in a context that requires it to be pure, its interface must be explicit in the scope of that use. The interface must specify that the procedure is pure. Note that statement functions are obsolete.

Constraint: All internal subprograms in a pure subprogram must be pure.

Constraint: In a pure subprogram, any variable that is in common or is accessed by host or USE association, is a dummy argument to a pure function, is a dummy argument with INTENT(IN) to a pure subroutine, or an object that is storage associated with any such variable, must not be used in the following contexts:

- As the *variable* of an *assignment_stmt*
- As a DO variable or implied DO variable
- As an *input_item* in a *read_stmt* from an internal file
- As an *internal_file_unit* in a *write_stmt*
- As an IOSTAT= specifier in an input or output statement with an internal file
- As the *pointer_object* of a *pointer_assignment_stmt*
- As the *target* of a *pointer_assignment_stmt*
- As the *expr* of an *assignment_stmt* in which the *variable* is of a derived type if the derived type has a pointer component at any level of component selection
- As an *allocate_object* or *stat_variable* in an *allocate_stmt* or *deallocate_stmt*, or as a *pointer_object* in a *nullify_stmt*
- As an actual argument associated with a dummy argument with INTENT(OUT) or INTENT(INOUT) or with the POINTER attribute

Constraint: Any procedure referenced in a pure subprogram, including one referenced via a defined operation or assignment, must be pure.

Constraint: A pure subprogram must not contain a *print_stmt*, *open_stmt*, *close_stmt*, *backspace_stmt*, *endfile_stmt*, *rewind_stmt*, or *inquire_stmt*.

Constraint: A pure subprogram must not contain a *read_stmt* or *write_stmt* with an *io_unit* that is an *external_file_unit* or an asterisk (*).

Constraint: A pure subprogram must not contain a *stop_stmt*.

1.2.13 Intrinsic Procedures

There are no syntax rules described in section 13, "Intrinsic procedures," of the Fortran 95 standard.

1.2.14 Scope, Association, and Definition

There are no syntax rules described in section 14, "Scope, association, and definition," of the Fortran 95 standard.

Decremental Features [2]

This chapter describes Fortran features that have been deleted or declared obsolescent in the current (Fortran 95) standard. The CF90 and MIPSpro 7 Fortran 90 compilers continue to support these features as extensions to the standard. The CF90 and MIPSpro 7 Fortran 90 compilers can generate messages when the deleted or obsolescent features are detected; to enable these messages, specify one of the following on the `f90`(1) command line: `-ansi` and `-fullwarn` (on IRIX systems) or `-e n` (on UNICOS or UNICOS/mk systems). For more information on these options, see the `f90`(1) man page.

2.1 Deleted Features

This section describes features that the Fortran 95 standard declares to be deleted from the Fortran language. These features had been included in previous revisions of the Fortran standard. The CF90 and MIPSpro 7 Fortran 90 compilers include these features as extensions. The deleted features are as follows:

- Real and double-precision `DO` variables. The preferred alternative is integer.
- The ability to branch to an `END IF` statement from outside its block. The preferred alternative is to branch to the statement following the `END IF`.
- The `PAUSE` statement. For more information on this, see Section 2.1.1, page 69.
- The `ASSIGN` statement, assigned `GO TO` statements, and assigned format specifiers. For more information on this, see Section 2.1.2, page 70.
- The `H` edit descriptor. For more information on this, see Section 2.1.3, page 72.

2.1.1 `PAUSE` Statement

Execution of a `PAUSE` statement requires operator or system-specific intervention to resume execution. In most cases, the same functionality can be achieved as effectively and in a more portable way with the use of an appropriate `READ` statement that awaits some input data.

The execution of the `PAUSE` statement suspends the execution of a program. This is now redundant, because a `WRITE` statement can be used to send a

message to any device, and a READ statement can be used to wait for and receive a message from the same device.

The PAUSE statement is defined as follows:

EXT	*pause_stmt*	**is**	PAUSE [*stop_code*]

The character constant or list of digits identifying the PAUSE statement is called the *stop_code* because it follows the same rules as those for the STOP statement's stop code. The stop code is accessible following program suspension. The CF90 and MIPSpro 7 Fortran 90 compilers send the *stop_code* to the standard error file (`stderr`). The following are examples of PAUSE statements:

```
PAUSE
PAUSE 'Wait #823'
PAUSE 100
```

2.1.2 ASSIGN, Assigned GO TO Statements, and Assigned Format Specifiers

The ASSIGN statement assigns a statement label to an integer variable. During program execution, the variable can be assigned labels of branch target statements, providing a dynamic branching capability in a program. The unsatisfactory property of these statements is that the integer variable name can be used to hold both a label and an ordinary integer value, leading to errors that can be hard to discover and programs that can be difficult to read.

A frequent use of the ASSIGN statement and assigned GO TO statement is to simulate internal procedures, using the ASSIGN statement to record the return point after a reusable block of code has completed. The internal procedure mechanism of Fortran now provides this capability.

A second use of the ASSIGN statement is to simulate dynamic format specifications by assigning labels corresponding to different format statements to an integer variable and using this variable in I/O statements as a format specifier. This use can be accomplished in a clearer way by using character strings as format specifications. Thus, it is no longer necessary to use either the ASSIGN statement or the assigned GO TO statement.

Execution of an ASSIGN statement causes the variable in the statement to become defined with a statement label value.

When a numeric storage unit becomes defined, all associated numeric storage units of the same type become defined. Variables associated with the variable in an ASSIGN statement, however, become undefined as integers when the ASSIGN statement is executed. When an entity of double-precision real type becomes defined, all totally associated entities of double-precision real type become defined.

Execution of an ASSIGN statement causes the variable in the statement to become undefined as an integer. Variables that are associated with the variable also become undefined.

2.1.2.1 Form of the ASSIGN and Assigned GO TO Statements

Execution of an ASSIGN statement assigns a label to an integer variable. Subsequently, this value can be used by an assigned GO TO statement or by an I/O statement to reference a FORMAT statement. The ASSIGN statement is defined as follows:

EXT	*assign_stmt*	**is**	ASSIGN *label* TO *scalar_int_variable*

The term *default integer type* in this section means that the integer variable must occupy a full word in order to be able to hold the address of the statement label. On Cray T3E systems, programs that contain an ASSIGN statement and are compiled with `-i 32` or `-s default32` must ensure that the *scalar_int_variable* is declared as `INTEGER(KIND=8)`. This ensures that it occupies a full word.

The variable must be a named variable of default integer type. It must not be an array element, an integer component of a structure, or an object of nondefault integer type.

The label must be the label of a branch target statement or the label of a FORMAT statement in the same scoping unit as the ASSIGN statement.

When defined with an integer value, the integer variable cannot be used as a label.

When assigned a label, the integer variable cannot be used as anything other than a label.

When the integer variable is used in an assigned GO TO statement, it must be assigned a label.

As the following example shows, the variable can be redefined during program execution with either another label or an integer value:

```
ASSIGN 100 TO K
```

Execution of the assigned `GO TO` statement causes a transfer of control to the branch target statement with the label that had previously been assigned to the integer variable.

The assigned `GO TO` statement is defined as follows:

EXT	*assigned_goto_stmt*	**is**	`GO TO` *scalar_int_variable* [[,] (*label_list*)]

The variable must be a named variable of default integer type. That is, it must not be an array element, a component of a structure, or an object of nondefault integer type.

The variable must be assigned the label of a branch target statement in the same scoping unit as the assigned `GO TO` statement.

If a label list appears, such as in the following examples, the variable must have been assigned a label value that is in the list:

```
GO TO K
GO TO K (10, 20, 100)
```

The `ASSIGN` statement also allows the label of a `FORMAT` statement to be dynamically assigned to an integer variable, which can later be used as a format specifier in `READ`, `WRITE`, or `PRINT` statements. This hinders readability, permits inconsistent usage of the integer variable, and can be an obscure source of error.

This functionality is available through character variables, arrays, and constants.

2.1.2.2 Assigned Format Specifiers

When an I/O statement containing the integer variable as a format specifier is executed, the integer variable can be defined with the label of a `FORMAT` specifier.

2.1.3 `H` Edit Descriptor

This edit descriptor can be a source of error because the number of characters following the descriptor can be miscounted easily. The same functionality is

available using the character constant edit descriptor, for which no count is required.

The following information pertains to the H edit descriptor:

Table 2. Summary of string edit descriptors

Descriptor	Description
H	Transfer of text character to output record
'text'	Transfer of a character literal constant to output record
"text"	Transfer of a character literal constant to output record

For more information on edit descriptors, see the *Fortran Language Reference Manual, Volume 2*.

2.2 Obsolescent Features

The obsolescent features are those features of previous Fortran standards that are considered by the Fortran 95 standard to be redundant. The Fortran 95 standard states that these features are obsolescent and provides preferred alternatives.

The obsolescent features and their preferred alternatives are as follows:

- Arithmetic IF. The preferred alternative is the IF statement or IF construct. For more information on the arithmetic IF, see the *Fortran Language Reference Manual, Volume 1*.
- Shared DO termination and termination on a statement other than END DO or CONTINUE statements. The preferred alternative is an END DO or a CONTINUE statement for each DO statement. For more information on this DO termination, see the *Fortran Language Reference Manual, Volume 1*.
- Alternate return. An alternate return introduces labels into an argument list to allow the called procedure to direct the execution of the caller upon return. The preferred alternative is to use a return code that is used in a CASE construct on return. This avoids an irregularity in the syntax and semantics of argument association. Consider the following statement:

```
CALL SUBR_NAME (X, Y, Z, *100, *200, *300)
```

The preceding statement can be replaced by the following code:

```
CALL SUBR_NAME (X, Y, Z, RETURN_CODE)
SELECT CASE (RETURN_CODE)
   CASE (1)
      ...
   CASE (2)
      ...
   CASE (3)
      ...
   CASE DEFAULT
      ...
END SELECT
```

For more information on alternate returns, see *Fortran Language Reference Manual, Volume 2*.

- Computed GO TO statement. The preferred alternative is the CASE construct, which is a generalized, easier to use, and more efficient means of expressing the same computation. For more information on the computed GO TO statement, see the *Fortran Language Reference Manual, Volume 1*.
- Statement functions. These are subject to a number of nonintuitive restrictions and are a potential source of error because their syntax is easily confused with that of an assignment statement. The preferred alternative is the internal function, which is a more generalized form of the statement function that competely superseded the statement function construct. For more information on statement functions, see the *Fortran Language Reference Manual, Volume 1*.
- DATA statements among executables. The statement ordering rules of previous Fortran standards allowed DATA statements to appear anywhere in a program unit after the specification statements. The ability to position DATA statements amongst executable statements is rarely used and is a potential source of error. For more information on the DATA statement, see the *Fortran Language Reference Manual, Volume 1*.
- Assumed character length functions. Assumed character length for functions is an irregularity in the language because the typical Fortran philosophy is that the attributes of a function result depend only on the actual arguments of the invocation and on any data accessible by the function through host or USE association. Some uses of this facility can be replaced with an automatic character length function in which the length of the function result is declared in a specification expression. Other uses can be replaced by the use

of a subroutine with arguments that correspond to the function result and the function arguments.

Note that dummy arguments of a function can be assumed character length. For more information on assumed character length functions, see the *Fortran Language Reference Manual, Volume 2*.

- Fixed source form. Fixed source form was designed when the principal machine-readable input medium for new programs was punched cards. Now that new and amended programs are typically entered at a keyboard with a screen display terminal, it is unnecessary overhead and is potentially error-prone to have to locate positions 6, 7, or 72 on a line. Free form source was designed expressly for this more modern technology. It is a simple matter to convert from fixed form to free form. For more information on fixed source form, see the *Fortran Language Reference Manual, Volume 1*.

- CHARACTER* form of CHARACTER declaration. The CHARACTER**length* form of specifying character declarations is redundant. For more information on the CHARACTER* form, see the *Fortran Language Reference Manual, Volume 1*.

Character Set [3]

The ASCII character set contains the control and graphic characters shown in the following table. Numbers, letters, and special characters in the character set are identified by the letter "C" in the *Notes* column. All other characters are members of the auxiliary character set. The letter "A" identifies the characters that belong to the Fortran character set as defined by the standard. Letters in parentheses following the descriptions in the *Description* column indicate the following control character usage:

- "CC" stands for Communication Control
- "FE" stands for Format Effector
- "IS" stands for Information Separator

Table 3. Character set

Character	Octal	Decimal	Hex	Notes	Description
NUL	000	000	00		Null
SOH	001	001	01		Start of heading (CC)
STX	002	002	02		Start of text (CC)
ETX	003	003	03		End of text (CC)
EOT	004	004	04		End of transmission (CC)
ENQ	005	005	05		Enquiry (CC)
ACK	006	006	06		Acknowledge (CC)
BEL	007	007	07		Bell (audible signal)
BS	010	008	08		Backspace (FE)
HT	011	009	09	C	Horizontal tabulation (FE)
LF	012	010	0A		Line feed (FE)
VT	013	011	0B		Vertical tabulation (FE)
FF	014	012	0C		Form feed (FE)
CR	015	013	0D		Carriage return (FE)

Character	Octal	Decimal	Hex	Notes	Description
SO	016	014	0E		Shift out
SI	017	015	0F		Shift in
DLE	020	016	10		Data link escape (CC)
DC1	021	017	11		Device control 1
DC2	022	018	12		Device control 2
DC3	023	019	13		Device control 3
DC4	024	020	14		Device control 4 (stop)
NAK	025	021	15		Negative acknowledge (CC)
SYN	026	022	16		Synchronous idle (CC)
ETB	027	023	17		End of transmission block (CC)
CAN	030	024	18		Cancel
EM	031	025	19		End of medium
SUB	032	026	1A		Substitute
ESC	033	027	1B		Escape
FS	034	028	1C		File separator (IS)
GS	035	029	1D		Group separator (IS)
RS	036	030	1E		Record separator (IS)
US	037	031	1F		Unit separator (IS)
space	040	032	20	A, C	(blank)
!	041	033	21	A, C	Exclamation point
"	042	034	22	A, C	Quotation mark
#	043	035	23		Number sign
$	044	036	24	A, C	Dollar sign (currency symbol)
%	045	037	25	A, C	Percent
&	046	038	26	A, C	Ampersand
’	047	039	27	A, C	Apostrophe (single quote)
(	050	040	28	A, C	Opening (left) parenthesis
)	051	041	29	A, C	Closing (right) parenthesis

Character	Octal	Decimal	Hex	Notes	Description
*	052	042	2A	A, C	Asterisk
+	053	043	2B	A, C	Plus
,	054	044	2C	A, C	Comma (cedilla)
-	055	045	2D	A, C	Minus (hyphen)
.	056	046	2E	A, C	Period (decimal point)
/	057	047	2F	A, C	Slant (slash, virgule)
0	060	048	30	A, C	Zero
1	061	049	31	A, C	One
2	062	050	32	A, C	Two
3	063	051	33	A, C	Three
4	064	052	34	A, C	Four
5	065	053	35	A, C	Five
6	066	054	36	A, C	Six
7	067	055	37	A, C	Seven
8	070	056	38	A, C	Eight
9	071	057	39	A, C	Nine
:	072	058	3A	A, C	Colon
;	073	059	3B	A, C	Semicolon
<	074	060	3C	A, C	Less than
=	075	061	3D	A, C	Equal
>	076	062	3E	A, C	Greater than
?	077	063	3F	A, C	Question mark
@	100	064	40	C	"At" sign. Reserved for internal use. Not a valid character on IRIX systems.
A	101	065	41	A, C	Uppercase letter
B	102	066	42	A, C	Uppercase letter
C	103	067	43	A, C	Uppercase letter
D	104	068	44	A, C	Uppercase letter

Character	Octal	Decimal	Hex	Notes	Description
E	105	069	45	A, C	Uppercase letter
F	106	070	46	A, C	Uppercase letter
G	107	071	47	A, C	Uppercase letter
H	110	072	48	A, C	Uppercase letter
I	111	073	49	A, C	Uppercase letter
J	112	074	4A	A, C	Uppercase letter
K	113	075	4B	A, C	Uppercase letter
L	114	076	4C	A, C	Uppercase letter
M	115	077	4D	A, C	Uppercase letter
N	116	078	4E	A, C	Uppercase letter
O	117	079	4F	A, C	Uppercase letter
P	120	080	50	A, C	Uppercase letter
Q	121	081	51	A, C	Uppercase letter
R	122	082	52	A, C	Uppercase letter
S	123	083	53	A, C	Uppercase letter
T	124	084	54	A, C	Uppercase letter
U	125	085	55	A, C	Uppercase letter
V	126	086	56	A, C	Uppercase letter
W	127	087	57	A, C	Uppercase letter
X	130	088	58	A, C	Uppercase letter
Y	131	089	59	A, C	Uppercase letter
Z	132	090	5A	A, C	Uppercase letter
{	133	091	5B		Opening (left) brace
\	134	092	5C		Reverse slant (backslash)
}	135	093	5D		Closing (right) brace
^	136	094	5E		Caret (circumflex)
_	137	095	5F	A, C	Underline
‘	140	096	60		Grave accent

Character	Octal	Decimal	Hex	Notes	Description
a	141	097	61	A, C	Lowercase letter
b	142	098	62	A, C	Lowercase letter
c	143	099	63	A, C	Lowercase letter
d	144	100	64	A, C	Lowercase letter
e	145	101	65	A, C	Lowercase letter
f	146	102	66	A, C	Lowercase letter
g	147	103	67	A, C	Lowercase letter
h	150	104	68	A, C	Lowercase letter
i	151	105	69	A, C	Lowercase letter
j	152	106	6A	A, C	Lowercase letter
k	153	107	6B	A, C	Lowercase letter
l	154	108	6C	A, C	Lowercase letter
m	155	109	6D	A, C	Lowercase letter
n	156	110	6E	A, C	Lowercase letter
o	157	111	6F	A, C	Lowercase letter
p	160	112	70	A, C	Lowercase letter
q	161	113	71	A, C	Lowercase letter
r	162	114	72	A, C	Lowercase letter
s	163	115	73	A, C	Lowercase letter
t	164	116	74	A, C	Lowercase letter
u	165	117	75	A, C	Lowercase letter
v	166	118	76	A, C	Lowercase letter
w	167	119	77	A, C	Lowercase letter
x	170	120	78	A, C	Lowercase letter
y	171	121	79	A, C	Lowercase letter
z	172	122	7A	A, C	Lowercase letter
[	173	123	7B		Opening (left) bracket
|	174	124	7C		Vertical line

Character	Octal	Decimal	Hex	Notes	Description
]	175	125	7D		Closing (right) bracket
~	176	126	7E		Overline (tilde, general accent)
DEL	177	127	7F		Delete

Extensions and Differences [4]

This chapter describes the differences various Fortran implementations. This chapter is divided into the following sections:

- Fortran 95 standard differences and incompatibilities with FORTRAN 77 implementations
- CF90 extensions to Fortran 95
- MIPSpro 7 Fortran 90 extensions to Fortran 95
- CF90, MIPSpro Fortran 77 and MIPSpro 7 Fortran 90 differences

4.1 Fortran 95 Standard Differences and Incompatibilities With FORTRAN 77 Implementations

This section discusses the following topics:

- Fortran 95 and the `G` edit descriptor output differences
- Fortran 95 and list-directed output differences
- Delimited and undelimited character strings in list-directed I/O
- List-directed I/O and floating-point zero

4.1.1 Fortran 95 and the `G` Edit Descriptor Output Differences

The format of a floating-point zero written with a `G` edit descriptor is different in Fortran 95.

The floating-point zero was written with an `E`*w*`.`*d* edit descriptor in FORTRAN 77, but it is written with an `F`*w*`.`*d* edit descriptor in the CF90 and MIPSpro 7 Fortran 90 compilers.

The `G` edit descriptor has been expanded to be a general edit descriptor that can read or write any data type including character, integer, and logical data. FORTRAN 77 allows only floating-point data types.

Fortran 95 is specific about the rounding of floating-point values with the `G` format. The change in rules may cause asterisks in the output field for some

floating-point values. Other values will be written as an E*w*.*d*-formatted value by the CF90 and MIPSpro 7 Fortran 90 compilers.

Consider the following code fragment:

```
      DOUBLE PRECISION AVD, BVD, CVD
      AVD = 0.0D0
      WRITE(6, 1) AVD
1     FORMAT(G28.2)
      END
```

It generates the following output when compiled by CF90:

```
0.0
```

4.1.2 Fortran 95 and List-directed Output Differences

Fortran 95 requires a separator between noncharacter data and character data in list-directed output. FORTRAN 77 disallows a separator in this instance.

Consider the following example output list:

```
'This is a one(',1,')'
```

This output list generates different output under the two standards:

- Fortran 95 output:

  ```
  >This is a one( 1 )
  ```

- FORTRAN 77 output:

  ```
  >This is a one(1)
  ```

4.1.3 Delimited and Undelimited Character Strings in List-directed I/O

FORTRAN 77 compilers support only delimited character string input to a list-directed item that will be stored to a list item of type character.

4.1.4 List-directed I/O and Floating-point Zero

Fortran 95 specifies a different form of output constant for a floating-point zero in list-directed output records. Consider the following program:

```
      PRINT *,0.0
      PRINT 1, 0.0
1     FORMAT(1X,G12.2)
      END
```

The preceding code generates the following output for the CF90 compiler:

```
% f90 tt.f
% a.out
 0.E+0
      0.0
```

4.2 CF90 extensions to Fortran 95

This section discusses the following topics:

- List-directed I/O and Hollerith constants
- Differences in the B, O, and Z edit descriptors

4.2.1 List-directed I/O and Hollerith Constants

Fortran 95 does not support Hollerith constants in list-directed input files, but the CF90 and MIPSpro 7 Fortran 90 compilers provides this as an extension. Note that Hollerith data is an deleted feature. See Chapter 6, page 137, for information on deprecated features and preferred alternatives.

4.2.2 Differences in the B, O, and Z Edit Descriptors

The B, O, and Z edit descriptors are available in Fortran 95. They are limited to integer I/O list items.

Signed octal and hexadecimal values are not allowed in Fortran 95. The CF90 and MIPSpro 7 Fortran 90 compilers allow signed input values, but they write only unsigned values.

If the size of the value is less than *w* in O*w* or Z*w* on output, Fortran 95 requires blank padding on the left. If the edit descriptor O*w*.*m* or Z*w*.*m* is used, the field must contain at least *m* digits. The .*m* form is one way to get leading zeros with Fortran 95. If the size of the value is greater than *w* in O*w* or Z*w* on output, the CF90 and MIPSpro 7 Fortran 90 compilers fill the field with asterisks. The CF90 and MIPSpro 7 Fortran 90 compilers provide the Fortran 95 form of O*w* and Z*w* output.

4.3 MIPSpro 7 Fortran 90 extensions to Fortran 95

The MIPSpro 7 Fortran 90 compiler supports the following extensions to Fortran 95:

- Formatted I/O:
  - `Q` edit descriptor returns the number of characters remaining in the record.
  - The dollar sign (`$`) and backslash (\) edit descriptors suppress the newline character at the end of a record.
  - The field width of a data edit descriptor other than the `A` format cannot be present. The data edit descriptors are `B`, `D`, `E`, `F`, `G`, `I`, `L`, `O`, and `Z`. The field width defaults to a size chosen by the I/O library. The Fortran 95 standard allows a zero for the field width for the `I`, `B`, `O`, and `Z` data edit descriptors.
- Internal I/O:
  - `ENCODE` statement
  - `DECODE` statement
- `NAMELIST` I/O:
  - Skip unmatched namelist groups on input without error.
  - Accept either an ampersand (`&`) or dollar sign (`$`) as a prefix to the namelist group name on input. If the dollar sign is used, the slash (`/`) or `$END` can be used to terminate the namelist input group. If the ampersand is used, the slash or `$END` can be used to terminate the namelist input group.
- Unformatted I/O:
  - Specify `FORM='SYSTEM'` to ensure that no record headers exist in the file. This is available on all systems but is only necessary where the `f77` layer is the default for an unformatted file.
  - Specify `FORM='BINARY'` to ensure that no record headers exist in the file.

4.4 CF90, MIPSpro 7 Fortran 90, and MIPSpro Fortran 77 Differences

This section describes differences between the various Fortran compilers supported on IRIX, UNICOS, and UNICOS/mk systems.

4.4.1 MIPSpro 7 Fortran 90 and CF90 Compiler Differences

The following sections describe various differences found when compiling Fortran programs with the MIPSpro 7 Fortran 90 compiler and the CF90 compiler.

4.4.1.1 Numerical Model Differences

The model differences are as follows:

- The model for the CF90 `REAL(KIND=16)` data type on Cray T90 systems that support IEEE floating-point arithmetic is different from the model for the MIPSpro 7 Fortran 90 compiler. This means that the results of math functions, arithmetic calculations, I/O, and other library routines are different for this particular data type.
- The internal size of `INTEGER(KIND=1)`, `INTEGER(KIND=2)`, `LOGICAL(KIND=1)`, and `LOGICAL(KIND=2)` on the MIPSpro 7 Fortran 90 compiler is actually 1 and 2 bytes, respectively. The CF90 compiler treats these kind type parameters as `INTEGER(KIND=4)` and `LOGICAL(KIND=4)`.
- The default sizes of the MIPSpro 7 Fortran 90 integer, real, and logical data types are 32 bits. This differs from the CF90 default of 64 bits. The default data type sizes for the MIPSpro 7 Fortran 90 compiler may be incorrect for routines such as `IRTC`(3I) and SHMEM.
- The MIPSpro 7 Fortran 90 compiler does not support Cray character pointers.
- Pointer arithmetic is in default numeric storage units when using the CF90 compiler. Pointer arithmetic is in bytes when using the MIPSpro 7 Fortran 90 compiler.

For more information on the model, see the `models`(3I) man page.

4.4.1.2 Fortran Statement Differences

The Fortran 95 statement differences are as follows:

- When using the MIPSpro 7 Fortran 90 compiler, the execution of the `STOP` statement does not cause the word `STOP` to be written to `stdout` unless there is an argument to the `STOP` statement. The CF90 compiler always writes `STOP` to `stdout`.
- When using the MIPSpro 7 Fortran 90 compiler, the initialization of entities in a common block in a `DATA` statement can only be done in one program unit. That is, if a common block contains two variables initialized in a `DATA`

statement, those DATA statements must be in one program unit. The load indicates the presence of multiple initializations, and only one initialization is done.

With the CF90 compiler, different variables can be initialized in DATA statements in separate program units.

4.4.1.3 Function and Procedure Differences

The CF90 typeless functions (such as MASK(3I), SHIFTL(3I), SHIFTR(3I), SHIFT(3I), CVM(3I), and so on) are typed as integer functions by the MIPSpro 7 Fortran 90 compiler. Conversion occur in expressions involving a mixture of floating point and integer functions. When called by the CF90 compiler, these functions are typeless and no conversion occurs when there is a mixture of floating point and these typeless functions.

4.4.1.4 Modules Differences

When using the MIPSpro 7 Fortran 90 compiler, the compilation of Fortran 95 modules creates a *file*.mod for each module in the source file and creates a *file*.o for any module procedures.

To load compiled module procedures, specify *module*.o on the command line.

When using the CF90 compiler, compiling modules creates one *file*.o that contains all the Fortran 95 modules in the source file.

4.4.1.5 I/O Library Differences

The I/O library differences are as follows:

- Direct access formatted output files cannot be read as sequential formatted files by MIPSpro 7 Fortran 90 programs unless an assign(1) command with -s unblocked, -F cachea, or -F cache is supplied for the particular file.
- The set of I/O library errors begins at 4000 for MIPSpro 7 Fortran 90 programs. The error numbers begin at 1000 for CF90 programs.
- The FILENV environment variable must be set for MIPSpro 7 Fortran 90 programs when using the assign(1) command. For CF90 users, this environment variable need not be set.

4.4.1.6 Library Function and Procedure Differences

The library function and intrinsic procedure differences are as follows:

- The `CRI_IEEE_DEFINITIONS` module is available for the MIPSpro 7 Fortran 90 compiler, but the preferred name is `FTN_IEEE_DEFINITIONS` for the IEEE module and the interface to the IEEE procedures.
- The `MAXVAL`(3I) intrinsic procedure returns negative infinity for a zero-sized input array when called from a MIPSpro 7 Fortran 90 program and returns `-HUGE`(3I) when called from a CF90 program. A request for interpretation has been submitted to the Fortran standards committee.
- The `MINVAL`(3I) intrinsic procedure returns positive infinity for a zero-sized input array when called from a MIPSpro 7 Fortran 90 program and returns `+HUGE`(3I) when called from a CF90 program. A request for interpretation has been submitted to the Fortran standards committee.

4.4.1.7 Math Library Differences

The math library differences are as follows:

- The math routines from the MIPSpro 7 Fortran 90 compiler are referenced from the compiler. The results of the math routines from the MIPSpro 7 Fortran 90 compiler may differ from the results returned by the math routines for the CF90 compiler.
- Signaling of errors during references to the MIPSpro 7 Fortran 90 compiler math routines is not turned off. For the CF90 compiler, the math routines turn off signaling of errors and detect input data errors through source code checks.

4.4.2 MIPSpro FORTRAN 77 and MIPSpro 7 Fortran 90 Compiler Differences

The following sections describe various differences found when compiling Fortran programs with the MIPSpro FORTRAN 77 compiler and the MIPSpro 7 Fortran 90 compiler.

4.4.2.1 Intrinsic Function and Subroutine Differences

The MIPSpro FORTRAN 77 compiler supports the `TIME` intrinsic function and the MIPSpro 7 Fortran 90 compiler does not. The Fortran 95 standard defines the `DATE_AND_TIME`(3I) function, and its use is recommended when using the MIPSpro 7 Fortran 90 compiler.

4.4.2.2 DATA Statement Initialization Differences

The Fortran 95 standard explicitly disallows multiple explicit intitializations of the same variable or part of a variable. Doing so results in undefined behavior.

Some codes initialize the same local variable or part of a variable in a `DATA` statement. Some codes initialize data in `COMMON` blocks more than once, either in the same or in different program units.

The MIPSpro 7 Fortran 90 compiler, like many other implementations, allows `COMMON` blocks to be initialized in program units other than `BLOCKDATA` subprograms. Multiple initializations are not detected by the system. As a result, different processors may exhibit different behavior in cases of multiple initializations. For example, one processor may use the last value seen as the value of the initialized variable, while another may use the first value seen. Porting a code from one of these processors to another may result in differing results due to this difference.

Permitting multiple initializations of the same or part of a variable is not an extension. It is a user error that cannot be detected, in all cases, by the compiler. Behavior of multiple intializations is different across the IRIX, UNICOS, and UNICOS/mk platforms. For the program to be a standard conforming program with predictable results, you must remove multiple initializations.

4.4.2.3 I/O Record Length Differences

Fortran 95 standard I/O always specifies record lengths in I/O statements in bytes. By default, FORTRAN 77 direct-access unformatted I/O specifies the record length in words. This can cause incompatibilities when moving codes from FORTRAN 77 to Fortran 95 and vice versa. The `-bytereclen` option to the `f77`(1) command causes the FORTRAN 77 compiler to interpret all record lengths in bytes.

4.4.2.4 Special File Formats Differences

The MIPSpro FORTRAN 77 compiler permits you to specify the following two special modes in the `FORM=` clause of the `OPEN` statement:

- `FORM="BINARY"`, which permits reading and writing binary data from character variables.
- `FORM="SYSTEM"`, which allows input ignoring record boundaries.

These special modes are permitted in early releases of the MIPSpro 7 Fortran 90 compiler. However, neither form is supported by the Fortran 95 standard or by

the MIPSpro 7 Fortran 90 compiler, releases 7.2 and later. Either type of file access can also be achieved by using the `read`(2) and `write`(2) IRIX kernel functions.

4.4.2.5 MIscellaneous Differences

The following miscellaneous items in the MIPSpro Fortran 77 compiler are not in the MIPSpro 7 Fortran 90 compiler:

- Keyed access (`ACCESS='KEYED'`):
  - `OPEN` statement extensions:

    `ACCESS= 'KEYED'` specifier

    `KEY= ()` specifier

    `RECORDSIZE=` specifier (same as `RECL`)
  - `INQUIRE` statement extension:

    `KEYED=` specifier
  - `READ` statement extensions:

    `KEY=` specifier

    `KEYID=` specifier
  - `DELETE` statement
  - `REWRITE` statement
  - `UNLOCK` statement
- Direct access (`ACCESS ='DIRECT'`):
  - `OPEN` statement extension:

    `ASSOCIATEVARIABLE=` specifier to retain the `REC=` information.

    `MAXREC=` specifier contains the maximum number of records allowed for a direct access file. If not present, no limit exists.

    `RECORDSIZE=` specifier (same as `RECL`).
  - `DEFINE FILE` statement for a direct access unformatted file.
  - `FIND` statement for a direct access file.

  - Record length of one byte indicates a binary data stream without record boundaries.
  - Efficient reading of direct unformatted record in reverse order.
- Sequential access (`ACCESS='SEQUENTIAL'`). Extension for `ENDFILE` to allow multifile file by using a tab character rather than an `EOF`.
- `stdin`, `stdout`, and `stderr` I/O:
  - `ACCEPT` statement on `stdin` only. Same as a `READ` statement with an asterisk (*) as the unit.
  - `OPEN` statement extensions:

    ```
    FILE='SYS$INPUT', 'SYS$OUTPUT', 'SYS$ERROR'
    ```

  - `TYPE` statement on `stdout` only; same as a `PRINT` statement.
  - Allow the double asterisk (**) to indicate `stderr` unit; that is, specifying `WRITE(**,...)`.
  - Special compile time option to allow reread from `stdin` after EOF is `-vms_stdin`.
- Unformatted I/O:
  - Allow unformatted I/O on an `INTERNAL` file.
  - Nonstandard aggregate reference in an I/O list is restricted to unformatted I/O and only one aggregate item is allowed.
- Formatted I/O:
  - Carriage control with an ASCII `NUL` to cause overprint with no advance; that is, it does not return to left margin after printing.
  - Variable format in which the width can be an expression that is evaluated at execution.
  - `INQUIRE` statement extension:

    ```
    CARRIAGECONTROL= specifier
    ```

  - Allow the use of the letter `Q` for an exponent letter on input for a floating point value.
  - Special compile time option to interpret the carriage control character and replace it with the proper character in `stdout`.

- List-directed I/O:
  - No repeat counts are generated for list-directed output.
  - Allow the use of the letter `Q` for an exponent letter on input for a floating-point value.
- `NAMELIST` I/O:
  - No repeat counts are generated for namelist output.
  - Allow the use of the letter `Q` for an exponent letter on input for a floating point value.
- `INQUIRE` statement extensions:
  - `ORGANIZATION=` specifier specifier to also contain `ACCESS=` specifications.
  - Use of different-sized variables for more specifiers than allowed in Fortran 95.
- `OPEN` statement extensions:
  - `DEFAULTFILE=` specifier to contain an alternate path or prefix for the opened unit. The specifier is concatenated with the string in the `FILE=` specifier or with the unit number if `FILE=` is absent.
  - `DISP=` specifier to designate how the file is handled after the file is closed. This is another form of the `STATUS` specifier on the `CLOSE` statement for Fortran 95.
  - `FORM='BINARY'` to indicate a formatted read with the `A` format on binary data. The implemented form of binary is unformatted I/O without record headers.
  - `READONLY` specifier indicates the file is read only. Fortran 95 provides the `ACTION=` specifier with more options.
  - `RECORDTYPE=` specifier to indicate type of records the file can contain. This may be present for direct, sequential, and keyed files.
  - `SHARED=` specifier indicates the file must be flushed after each record is written.
  - `TYPE=` specifier is the same as the Fortran 95 `STATUS=` specifier.

- Environment variable `FORTRAN_BUFFER_SIZE` to set buffer size and determine if direct I/O should be used.
- Interoperability with UNIX routines for I/O such as `fseek`(3), `ftell`(3), `flush`(3), `fgetc`(3), `fstat`(2), and others. The PXF interface routines allow the use of these routines with a proper Fortran interface.
- Ability to query for the error status of the last I/O operation by unit.
- Allow MP-safe I/O at the subroutine level so non-MP-safe and MP-safe I/O can be mixed in one program but not in a program unit.

Data Representation and Storage [5]

This chapter shows how different data types are represented in storage and describes how the CF90 and MIPSpro 7 Fortran 90 compilers use storage.

Numbers shown on the formats are bit positions, which represent powers of 2 in binary notation. Code that depends on internal representation is not portable and might not conform with the Fortran standard.

Note: Storage words are represented here with bits counted from the right, making bit 0 the low-order bit and bit 31 or 63 the high-order bit. This agrees with the convention used in the integer-type bit functions as well as the convention used in Cray hardware documentation. It does not agree with some conventions used in some other UNICOS and UNICOS/mk software documentation.

This chapter describes the machine representation of data. The last sections in this chapter describe storage issues, including overindexing.

5.1 Data Representation for UNICOS Systems

The following sections describe the representation of data on UNICOS systems, including Cray T90 systems that support Cray floating-point arithmetic. These subsections do not describe data representation on Cray T90 systems that support IEEE floating-point arithmetic. For information pertaining to Cray T90 systems that support IEEE floating-point arithmetic, see Section 5.4, page 120.

5.1.1 Integer Type

All integer data is 64 bits (`KIND=8`), 2's complement.

When slower integer operations (`f90 -O nofastint`) are in effect, the range for `INTEGER(KIND=8)` operations is $-2^{63} < I < 2^{63}$ or approximately $-10^{18} < I < 10^{18}$.

When fast integer operations (`f90 -O fastint`) are in effect, which is the default, the range for `INTEGER(KIND=8)` operations is $-2^{46} < I < 2^{46}$ or approximately $-10^{13} < I < 10^{13}$.

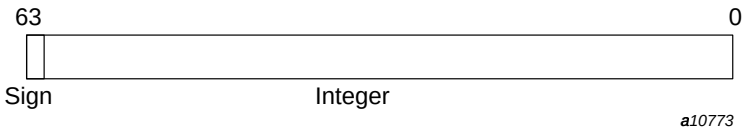

Figure 1. Default 64-bit integers

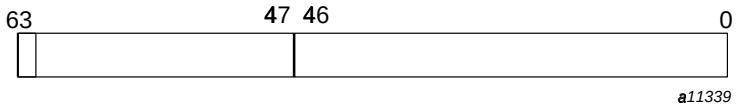

Figure 2. Fast integer operations with `INTEGER(KIND=8)`, UNICOS systems (except Cray T90 systems that support IEEE floating-point arithmetic)

To declare an entity to be of type integer, specify one of the following:

- `KIND=1`, `KIND=2`, `KIND=4`, or `KIND=8`.
- `KIND=KIND(`*kind_expr*`)`, where *kind_expr* is a scalar initialization expression with a kind type parameter that evaluates to 1, 2, 4, or 8.

5.1.2 Real Type

Real (floating-point) numbers are represented in a packed representation of a binary mantissa and an exponent (power of 2). The bits in a Cray word are used to represent a real number as follows:

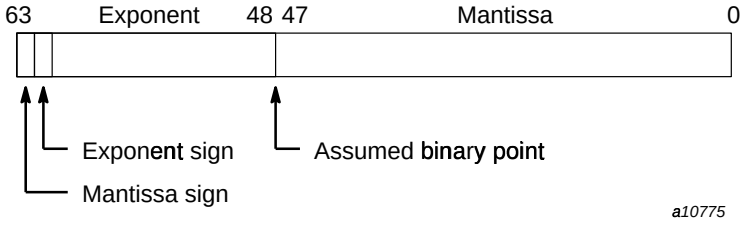

Figure 3. Real type

Notes on real data type representation:

The exponent is a power of 2, represented by a number that is 40000_8 higher than the actual value; this is called a *bias*. The effect of the bias is that the second bit in the word serves as the exponent's sign bit. This bit's usage is the inverse of the mantissa's sign bit, as follows:

Bit	Applies to	1 value indicates
63	Mantissa	Negative
62	Exponent	Positive

The exponent is represented by the second through sixth digits in an octal printout; these digits have the range 40000 through 57776_8 for a positive exponent, and 37777 through 20003_8 for a negative exponent.

When the bias is accounted for, the range of all exponents is as follows (notice the negative range is one smaller):

- 2^{-17775} to 2^{17776} (octal)

  or

- 2^{-8189} to 2^{8190} (decimal)

The mantissa is a 48-bit signed fraction. The sign of the mantissa is separated from the rest of the mantissa as shown in the preceding diagram. The mantissa is not complemented for negative values. That is, the mantissa for –10.0 is the same as for +10.0.

In terms of decimal values, the floating-point format of the CPU allows representation of numbers to about 15 significant decimal digits in the following approximate decimal range:

$.367 \times 10^{-2465} < R < .273 \times 10^{2466}$

A zero value is not biased and is represented as a word of all zeros.

Following are some sample numbers as represented within memory:

Decimal	Octal	Hexadecimal
10.0	`0400045000000000000`	`4004A00000000000`
–10.0	`1400045000000000000`	`C004A00000000000`

0.1	037775631463146314631 5	3FFDCCCCCCCCCCCD
–0.1	137775631463146314631 5	BFFDCCCCCCCCCCCD

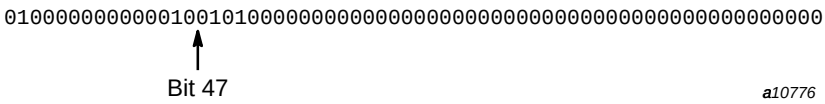

Figure 4. Binary version of 10.0

The leftmost bit, with a 0 value, indicates a positive mantissa; that is, the real value is positive. The next bit, set to 1, is the sign bit of the exponent, indicating a positive exponent value; that is, the absolute value of the number is 1.0 or greater. The value 4 in the exponent (100 appearing to the left of bit 47) means that the binary fraction in the mantissa is multiplied by 2^4 (or, to express it another way, the binary point is moved 4 bits to the right from the highest bit of the mantissa.) Interpreted in this way, the first 4 digits of the mantissa, 1010, indicate the real decimal value 10.0. You can display other values by printing them with formats `O22`, `Z16`, or `B64`.

To declare an entity to be of type real, specify one of the following:

- `KIND=4`, `KIND=8`
- `KIND=KIND(`*kind_expr*`)`, where *kind_expr* is a scalar initialization expression with a kind type parameter that evaluates to `4` or `8`

Note that a real data object with `KIND=4` has the same internal representation as a real data object with `KIND=8`. Numeric inquiry functions on a real data object with `KIND=4` return different values than on a real data object with `KIND=8`. A numeric operation on a real data object with `KIND=4` returns the same result as the same numeric operation on a real data object with `KIND=8`.

5.1.2.1 Normalized Floating-point Numbers

A nonzero, floating-point number is normalized if the most significant bit of the mantissa is nonzero. This condition implies that the mantissa has been shifted as far left as possible and the exponent adjusted accordingly. Therefore, the floating-point number has no leading zeros in the mantissa. The exception is that a normalized floating-point zero is all zeros.

When your program creates a floating-point number by inserting an exponent of 40060_8 into a `KIND=8` integer word, you should normalize the result before

using it in a floating-point operation. To do this, add the unnormalized floating-point operand to 0. Compiler optimization suppresses an operation such as `X=X+0`. You can perform it with code such as the following:

```
DATA REALZERO /0./
X = X + REALZERO
```

5.1.3 Double-precision Type

A double-precision value is represented by 2 words. The first has the same format as the real type. The second word uses bits 0 through 47 as 48 additional bits of the mantissa. The other 16 bits of the second word must be zeros. Double-precision numbers can be in the following range:

- 2^{-8188} .LE. R < 2^{8189}

  or approximately

- $.367 \times 10^{-2465} < R < .273 \times 10^{2466}$

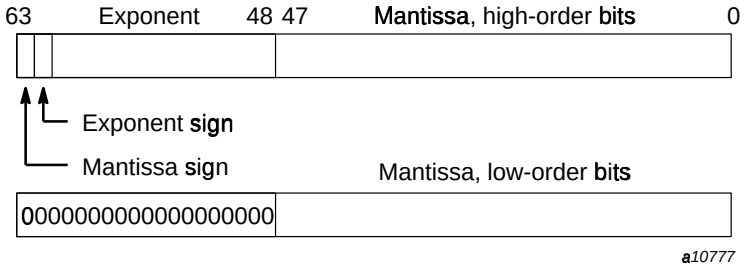

Figure 5. Double-precision type

To declare an entity to be of type double precision, specify one of the following:

- `REAL(KIND=16)`.
- `REAL(KIND=KIND(`*kind_expr*`))`, where *kind_expr* is a scalar initialization expression with a kind type parameter that evaluates to 16.

5.1.4 Single-precision Complex Type

A single-precision complex value is represented by 2 words, each of which has the same format as the real type. The first word represents the real part, and

the second represents the imaginary part. Each word has the same range as a real value.

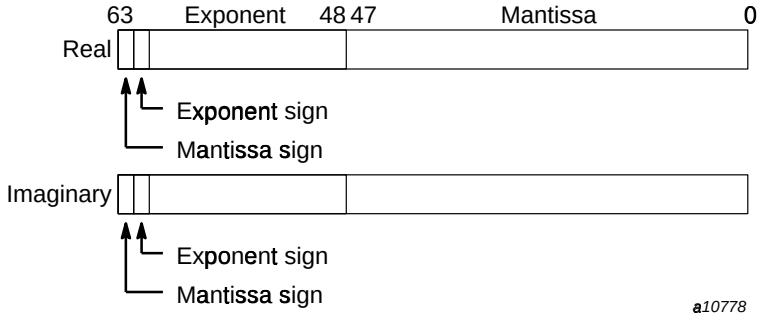

Figure 6. Single-precision complex type

To declare an entity to be of single-precision complex type, specify one of the following:

- `KIND=4` or `KIND=8`.
- `KIND=KIND(`*kind_expr*`)`, where *kind_expr* is a scalar initialization expression that evaluates to 4 or 8.

Note that a complex data object with `KIND=4` has the same internal representation as a complex data object with `KIND=8`. Numeric inquiry functions on a complex data object with `KIND=4` return different values than on a complex data object with `KIND=8`. A numeric operation on a complex data object with `KIND=4` returns the same result as the same numeric operation on a complex data object with `KIND=8`.

5.1.5 Double-precision Complex Type

Values of double precision complex type are represented by 4 words. The first 2 words are the real part, and the last 2 words are the imaginary part. The real part and the imaginary part each have the same range as a double precision value.

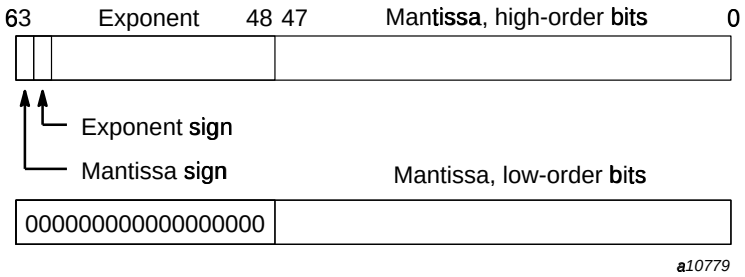

Figure 7. Double-precision complex type (real portion)

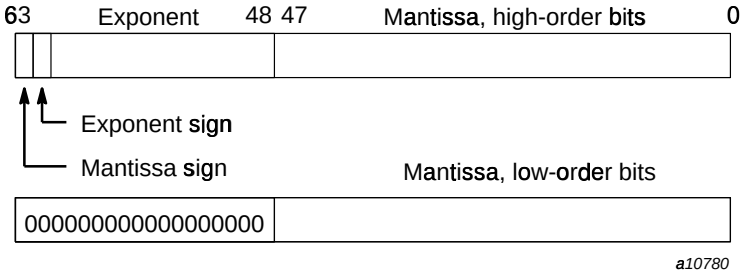

Figure 8. Double-precision complex type (imaginary portion)

To declare an entity to be of double-precision complex type, specify one of the following:

- `KIND=16`.
- `KIND=KIND(`*kind_expr*`)`, where *kind_expr* is a scalar initialization expression with a kind type parameter that evaluates to 16.

5.1.6 Character Type

Characters are represented by 8-bit ASCII codes packed eight per word.

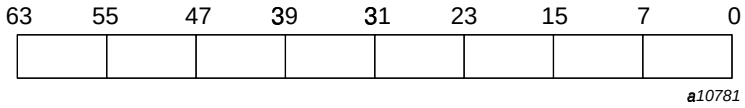

Figure 9. Character type

The CF90 compiler does not support a nondefault character type. The only kind value supported is 1.

5.1.7 Logical Type

A logical variable uses one 64-bit word. Its value is true if the numeric value in the word is negative (typically, –1), and it is false if the numeric value in the word is nonnegative (typically, 0).

Note: SGI does not guarantee a particular internal representation of logical values on any machine or system; the CF90 compiler is designed on the assumption that logical values will be used only as described in the Fortran standard. Therefore, it is not good programming practice to exploit gaps in type checking, such as between a function reference and its function value, to use logical values as numbers or vice versa.

To declare an entity to be of logical type, you can specify one of the following:

- `KIND=1`, `KIND=2`, `KIND=4`, or `KIND=8`.
- `KIND=KIND(`*kind_expr*`)`, where *kind_expr* is a scalar initialization expression with a kind type parameter that evaluates to 1, 2, 4, or 8.

Note that logical entities with `KIND=1`, `KIND=2`, `KIND=4`, and `KIND=8` all occupy 64 bits.

5.1.8 Cray Character Pointers

Cray character pointers include a word address, bit offset, and bit length field.

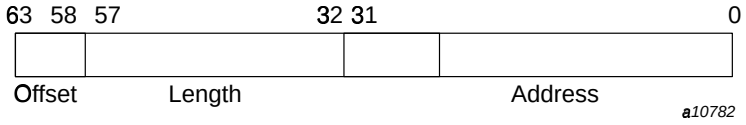

Figure 10. 64-bit addressing for UNICOS systems (except Cray T90 systems)

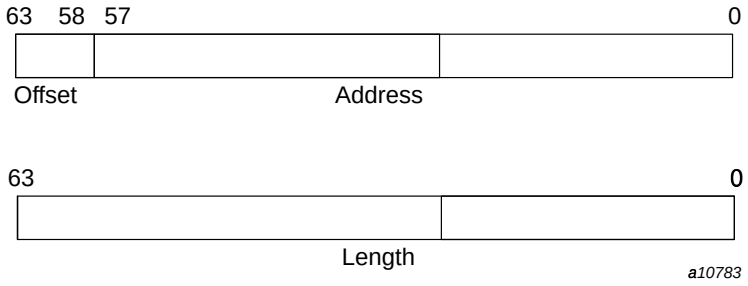

Figure 11. 64-bit addressing for Cray T90 systems

5.2 Data Representation for IRIX systems

The following sections describe the representation of data on IRIX systems.

Note: On IRIX systems, `KIND=4` values are stored in 32 bits and can be packed two per word.

5.2.1 Integer Type

The following sections describe integer data representation of `KIND=1`, `2`, `4`, and `8` on IRIX systems.

5.2.1.1 `KIND=1`

Range: $-2^7 < I < 2^7$ or approximately $-10^2 < I < 10^2$

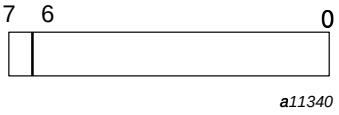

Figure 12. `INTEGER(KIND=1)` on IRIX systems

To declare 8-bit integers, specify one of the following:

- `KIND=1`.
- `KIND=KIND(`*kind_expr*`)`, where *kind_expr* is a scalar initialization expression with a kind type parameter that evaluates to 1.

5.2.1.2 KIND=2

Range: $-2^{15} < I < 2^{15}$ or approximately $-10^4 < I < 10^4$

Figure 13. `INTEGER(KIND=2)` on IRIX systems

To declare 16-bit integers, specify one of the following:

- `KIND=2`.
- `KIND=KIND(`*kind_expr*`)`, where *kind_expr* is a scalar initialization expression with a kind type parameter that evaluates to 2.

5.2.1.3 KIND=4

Range: $-2^{31} < I < 2^{31}$ or approximately $-10^9 < I < 10^9$

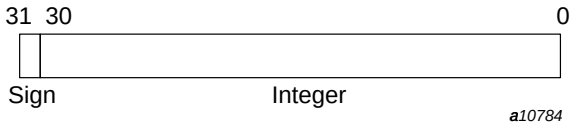

Figure 14. `INTEGER(KIND=4)` on IRIX systems

To declare 32-bit integers, specify one of the following:

- `KIND=4`.
- `KIND=KIND(`*kind_expr*`)`, where *kind_expr* is a scalar initialization expression with a kind type parameter that evaluates to 4.

5.2.1.4 KIND=8

Range: $-2^{63} < I < 2^{63}$ or approximately $-10^{18} < I < 10^{18}$

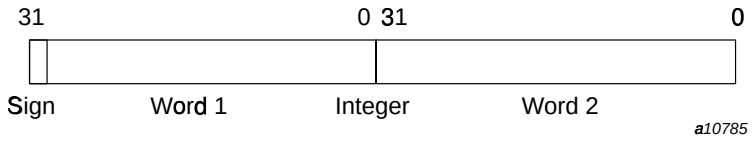

Figure 15. `INTEGER(KIND=8)` on IRIX systems

To declare 64-bit integers, specify one of the following:

- `KIND=8`.
- `KIND=KIND(`*kind_expr*`)`, where *kind_expr* is a scalar initialization expression with a kind type parameter that evaluates to 8.

5.2.2 Real Type

The following sections describe real data representation of `KIND= 4`, `8`, and `16` on IRIX systems. Real (floating-point) numbers are represented in a packed representation of a sign, an exponent (power of 2), and a binary mantissa.

5.2.2.1 `KIND=4`

Range: -2^{-125} .LE. I < 2^{128} or approximately -10^{-38} .LE. I < 10^{38}

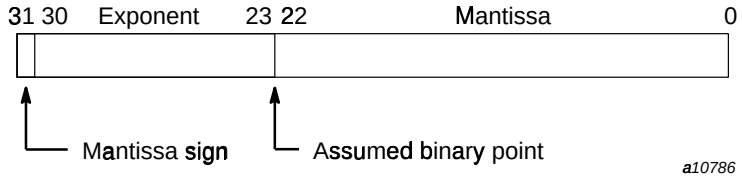

Figure 16. `REAL(KIND=4)` on IRIX systems

To declare 32-bit reals, specify one of the following:

- `KIND=4`.
- `KIND=KIND(`*kind_expr*`)`, where *kind_expr* is a scalar initialization expression with a kind type parameter that evaluates to 4.

Notes on real data type representation:

The exponent is a power of 2, represented by a number that is 177_8 higher than the actual value; this is called a *bias*. The effect of the bias is that the second bit in the word serves as the exponent's sign bit. This bit's usage is the inverse of the mantissa's sign bit, as follows:

Bit	Applies to	1 value indicates
31	Mantissa	Negative
30	Exponent	Positive (> 0)

The exponent is represented by the second through ninth digits in a binary printout; these digits have the range 01111111_2 through 11111110_2 for a positive exponent, and 00000000_2 through 01111110_2 for a negative exponent.

When the bias is accounted for, the range of all exponents is as follows:

- 2^{-177} to 2^{177} (octal)

  or

- 2^{-127} to 2^{127} (decimal)

The mantissa is a 24-bit fraction with an assumed leading 1; that is, the leading 1 is not stored. The only exception is for the value 0, which has an assumed leading 0. The sign of the mantissa is separated from the rest of the mantissa as shown in the preceding diagram. The mantissa is not complemented for negative values. That is, the mantissa for –10.0 is the same as for +10.0.

In terms of decimal values, the 32-bit floating-point format allows representation of numbers to about 7 significant decimal digits in the following approximate decimal range:

$1.18 \times 10^{-38} < R < 3.4 \times 10^{38}$

A zero value is not biased and is represented as a word of all zeros.

The following are some sample numbers as represented within memory:

Decimal	Octal	Hexadecimal
10.0	`010110000000`	`41200000`
–10.0	`030110000000`	`C1200000`

Decimal	Octal	Hexadecimal
0.1	007563146315	3DCCCCCD
–0.1	027563146315	BDCCCCCD

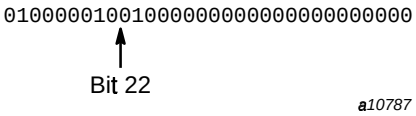

Figure 17. Binary version of 10.0

The leftmost bit, with a 0 value, indicates a positive mantissa; that is, the real value is positive. The next 8 bits (10000010, or decimal 130) are the exponent. Subtracting the bias of 127 yields an exponent of 3, meaning that the binary fraction in the mantissa is multiplied by 2^3. To express it another way, the binary point is moved 3 bits to the right from the mantissa's highest bit. Interpreted this way, the first 4 bits of the mantissa, [1]010, indicate the real decimal value 10.0 (remember that there is an assumed 1 to the left of the mantissa in the IEEE floating-point format with a binary point to its immediate right). You can display other values by printing them with formats `O11`, `Z8`, or `B32`.

5.2.2.2 `KIND=8`

Double precision, `REAL(KIND=8)`, values are represented in 2 words on IRIX systems.

Range: -2^{-1021} .LE. I < 2^{1024} or approximately -10^{-308} .LE. I < 10^{308}

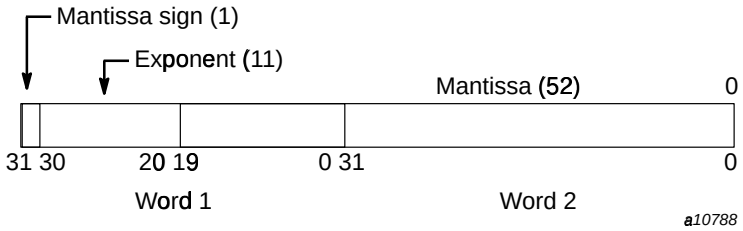

Figure 18. `REAL(KIND=8)` on IRIX systems

To declare 64-bit reals, specify one of the following:

- `KIND=8`.
- `KIND=KIND(`*kind_expr*`)`, where *kind_expr* is a scalar initialization expression with a kind type parameter that evaluates to 8.

5.2.2.3 KIND=16

Quad precision, `REAL(KIND=16)`, values are represented in 4 words on IRIX systems. For more information on quad precision representation IRIX systems, see `math`(3M).

Range: -2^{-967} .LE. I < 2^{1023} or approximately -10^{-292} .LE. I < 10^{308}

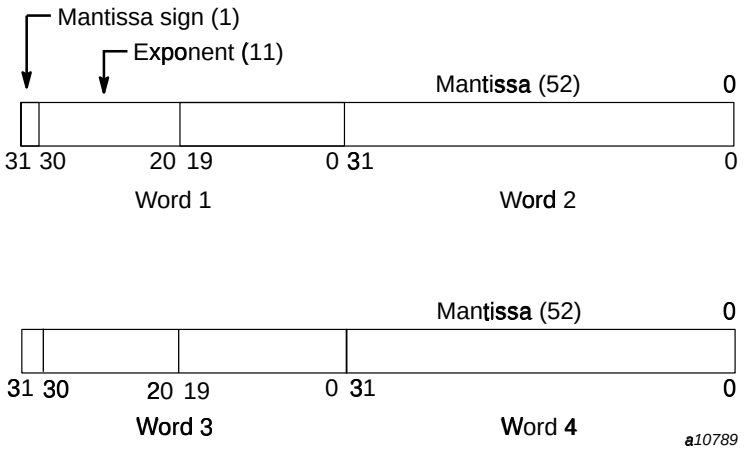

Figure 19. `REAL(KIND=16)` on IRIX systems

To declare 128-bit reals, specify one of the following:

- `KIND=16`.
- `KIND=KIND(`*kind_expr*`)`, where *kind_expr* is a scalar initialization expression with a kind type parameter that evaluates to 16.

5.2.3 Complex Type

The following sections describe complex data representation of `KIND=4`, `8`, and `16` on IRIX systems. A complex value has two parts. The first part represents the real part, and the second represents the imaginary part. Each word has the same range as a real value.

5.2.3.1 `KIND=4`

A single-precision, `KIND=4`, complex value is represented by 2 words. The first word represents the real part, and the second represents the imaginary part. Each word has the same range as a real value.

Range: -2^{-125} .LE. I < 2^{128} or approximately -10^{38} .LE. I < 10^{38}

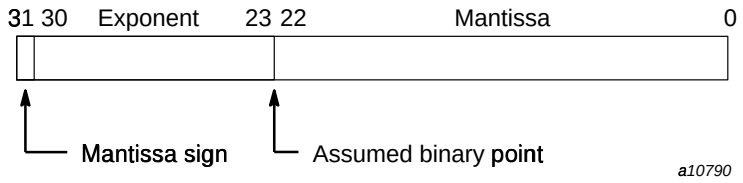

Figure 20. `COMPLEX(KIND=4)` on IRIX systems (real portion)

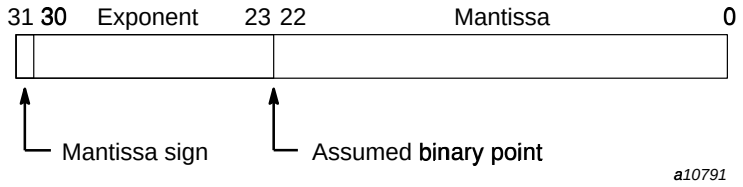

Figure 21. `COMPLEX(KIND=4)` on IRIX systems (imaginary portion)

To declare an entity to be of single-precision, complex type, specify one of the following:

- `KIND=4`.
- `KIND=KIND(`*kind_expr*`)`, where *kind_expr* is a scalar initialization expression with a kind type parameter that evaluates to 4.

5.2.3.2 KIND=8

A double-precision, KIND=8, complex value is represented by 4 words. The first 2 words represent the real part, and the second 2 words represent the imaginary part. Each word has the same range as a real value.

Range: -2^{-1021} .LE. I < 2^{1024} or approximately -10^{308} .LE. I < 10^{308}

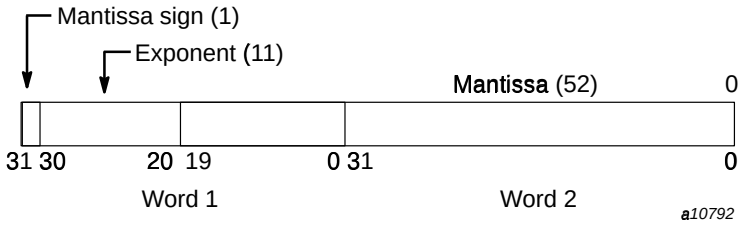

Figure 22. COMPLEX(KIND=8) on IRIX systems (real portion)

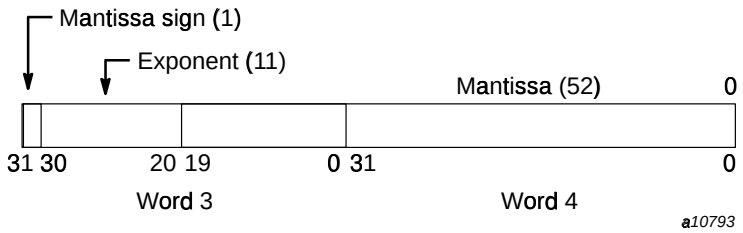

Figure 23. COMPLEX(KIND=8) on IRIX systems (imaginary portion)

To declare an entity to be of double-precision, complex type, specify one of the following:

- KIND=8.
- KIND=KIND(*kind_expr*), where *kind_expr* is a scalar initialization expression with a kind type parameter that evaluates to 8.

5.2.3.3 KIND=16

A quad precision, KIND=16, complex value is represented by 8 words. The first 4 words represent the real part, and the second 4 words represent the imaginary part. Each word has the same range as a real value.

Range: -2^{-967} .LE. I < 2^{1023} or approximately -10^{-292} .LE. I < 10^{308}

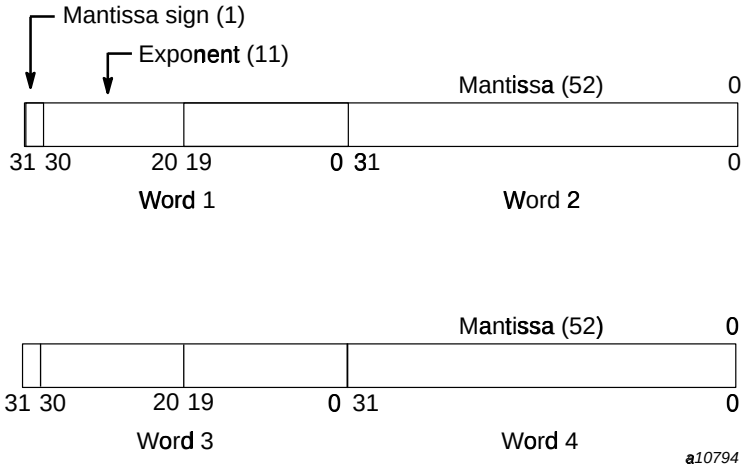

Figure 24. `COMPLEX(KIND=16)` on IRIX systems (real portion)

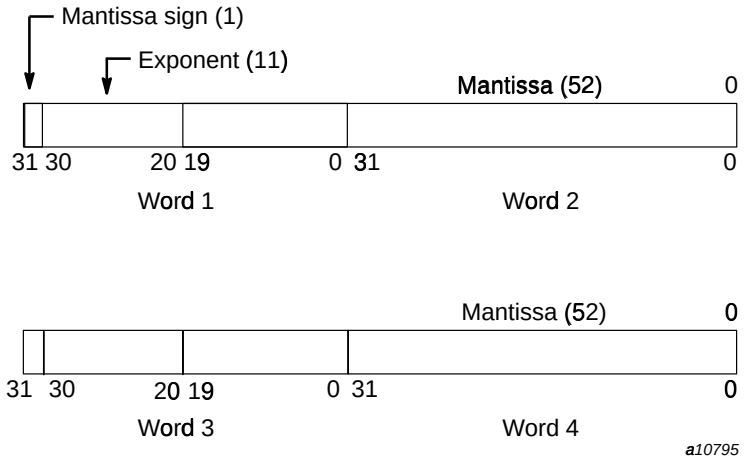

Figure 25. `COMPLEX(KIND=16)` on IRIX systems (imaginary portion)

To declare an entity to be of quad precision, complex type, specify one of the following:

- `KIND=16`.

- `KIND=KIND(`*kind_expr*`)`, where *kind_expr* is a scalar initialization expression with a kind type parameter that evaluates to 16.

5.2.4 Character Type

Characters are represented by 8-bit ASCII codes. On IRIX systems, the codes are stored in 1 byte.

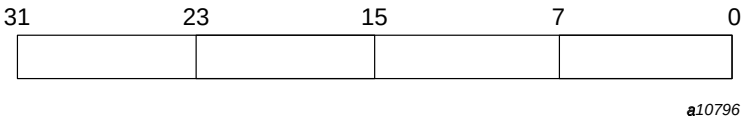

Figure 26. Character type

The MIPSpro 7 Fortran 90 compiler does not support a nondefault character type. The only kind value supported is 1.

5.2.5 Logical Type

Logical entities specified as `KIND=1`, `KIND=2`, and `KIND=4` occupy 32 bits on IRIX systems. Logical entities specified as `KIND=8` occupy 64 bits on IRIX systems. Its value is true if the numeric value in the word is one (1). Its value is false if the numeric value in the word is zero (0).

Note: SGI does not guarantee a particular internal representation of logical values on any machine or system; the MIPSpro 7 Fortran 90 compiler is designed on the assumption that logical values will be used only as described in the Fortran standard. Therefore, it is not good programming practice to use logical values as numbers or vice versa.

To declare an entity to be of logical type, you can specify one of the following:

- `KIND=1`, `KIND=2`, `KIND=4`, or `KIND=8`.
- `KIND=KIND(`*kind_expr*`)`, where *kind_expr* is a scalar initialization expression with a kind type parameter that evaluates to 1, 2, 4, or 8.

5.2.6 Cray Character Pointers (Deferred Implementation)

Cray character pointers include a byte address and a byte length field.

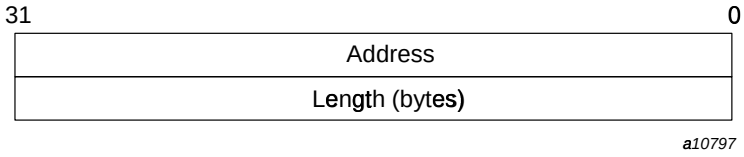

Figure 27. 32-bit addressing on IRIX systems

5.3 Data Representation for UNICOS/mk Systems

The following sections describe the representation of data on UNICOS/mk systems.

Note: On UNICOS/mk systems, `KIND=4` values are stored in 32 bits and can be packed two per word.

5.3.1 Integer Type

The following subsections describe integer data representation of `KIND=1`, `2`, `4`, and `8` on UNICOS/mk systems.

5.3.1.1 `KIND=1`, `KIND=2`, or `KIND=4`

Range: $-2^{31} < I < 2^{31}$ or approximately $-10^9 < I < 10^9$

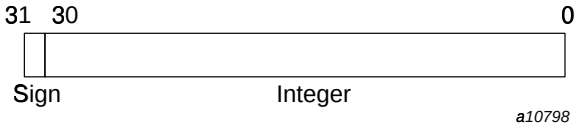

Figure 28. Integer `KIND=1`, `2`, or `4` on UNICOS/mk systems

To declare 32-bit integers, specify one of the following:

- `KIND=1`, `KIND=2`, or `KIND=4`.
- `KIND=KIND(`*kind_expr*`)`, where *kind_expr* is a scalar initialization expression with a kind type parameter that evaluates to 1, 2, or 4.

5.3.1.2 KIND=8

Range: $-2^{63} < I < 2^{63}$ or approximately $-10^{18} < I < 10^{18}$

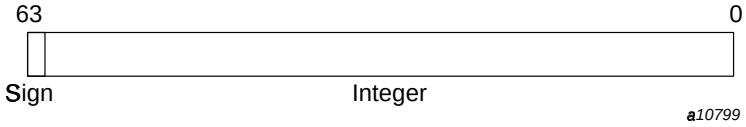

Figure 29. INTEGER(KIND=8) on UNICOS/mk systems

To declare 64-bit integers, specify one of the following:

- KIND=8.
- KIND=KIND(*kind_expr*), where *kind_expr* is a scalar initialization expression with a kind type parameter that evaluates to 8.

5.3.2 Real Type

The following sections describe real data representation of KIND=4 and 8. Real (floating-point) numbers are represented in a packed representation of a sign, an exponent (power of 2), and a binary mantissa.

5.3.2.1 KIND=4

Range: -2^{-125} .LE. $I < 2^{128}$ or approximately -10^{-38} .LE. $I < 10^{38}$

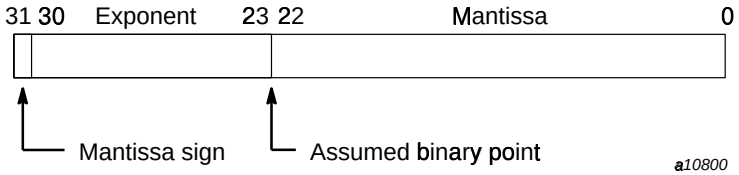

Figure 30. REAL(KIND=4) on UNICOS/mk systems

To declare 32-bit reals, specify one of the following:

- KIND=4.

- `KIND=KIND(`*kind_expr*`)`, where *kind_expr* is a scalar initialization expression with a kind type parameter that evaluates to 4.

Notes on real data type representation:

The exponent is a power of 2, represented by a number that is 177_8 higher than the actual value; this is called a *bias*. The effect of the bias is that the second bit in the word serves as the exponent's sign bit. This bit's usage is the inverse of the mantissa's sign bit, as follows:

Bit	Applies to	1 value indicates
31	Mantissa	Negative
30	Exponent	Positive (> 0)

The exponent is represented by the second through ninth digits in a binary printout; these digits have the range 01111111_2 through 11111110_2 for a positive exponent, and 00000000_2 through 01111110_2 for a negative exponent.

When the bias is accounted for, the range of all exponents is as follows:

- 2^{-177} to 2^{177} (octal)

  or

- 2^{-127} to 2^{127} (decimal)

The mantissa is a 24-bit fraction with an assumed leading 1; that is, the leading 1 is not stored. The only exception is for the value 0, which has an assumed leading 0. The sign of the mantissa is separated from the rest of the mantissa as shown in the preceding diagram. The mantissa is not complemented for negative values. That is, the mantissa for –10.0 is the same as for +10.0.

In terms of decimal values, the 32-bit floating-point format allows representation of numbers to about 7 significant decimal digits in the following approximate decimal range:

$1.18 \times 10^{-38} < R < 3.4 \times 10^{38}$

A zero value is not biased and is represented as a word of all zeros.

The following are some sample numbers as represented within memory:

Decimal	Octal	Hexadecimal
10.0	010110000000	41200000
–10.0	030110000000	C1200000
0.1	007563146315	3DCCCCCD
–0.1	027563146315	BDCCCCCD

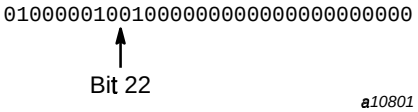

Figure 31. Binary version of 10.0

The leftmost bit, with a 0 value, indicates a positive mantissa; that is, the real value is positive. The next 8 bits (10000010, or decimal 130) are the exponent. Subtracting the bias of 127 yields an exponent of 3, meaning that the binary fraction in the mantissa is multiplied by 2^3; to express it another way, the binary point is moved 3 bits to the right from the mantissa's highest bit. Interpreted this way, the first 4 bits of the mantissa, [1]010, indicate the real decimal value 10.0; remember that there is an assumed 1 to the left of the mantissa in the IEEE floating-point format with a binary point to its immediate right. You can display other values by printing them with formats `O11`, `Z8`, or `B32`.

5.3.2.2 `KIND=8`

Range: -2^{-1021} .LE. I < 2^{1024} or approximately -10^{-308} .LE. I < 10^{308}

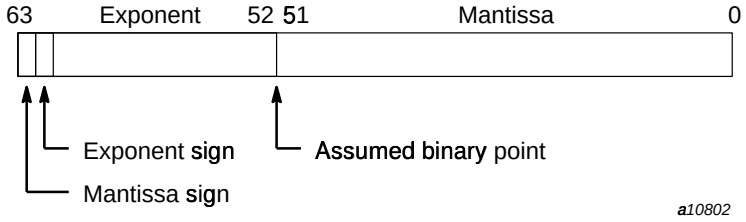

Figure 32. `REAL(KIND=8)` on UNICOS/mk systems

To declare 64-bit reals, specify one of the following:

- `KIND=8`.
- `KIND=KIND(`*kind_expr*`)`, where *kind_expr* is a scalar initialization expression with a kind type parameter that evaluates to 8.

5.3.3 Complex Type

The following sections describe complex data representation of `KIND=4` and `KIND=8` on UNICOS/mk systems. A complex value has two parts. The first part represents the real part, and the second represents the imaginary part. Each word has the same range as a real value.

5.3.3.1 `KIND=4`

A `KIND=4` complex value consists of 2 parts. The first part represents the real portion, and the second represents the imaginary portion. Each part has the same range as a 32-bit (or `KIND=4`) real value.

Range: -2^{-125} .LE. I < 2^{128} or approximately -10^{-38} .LE. I <10^{38}

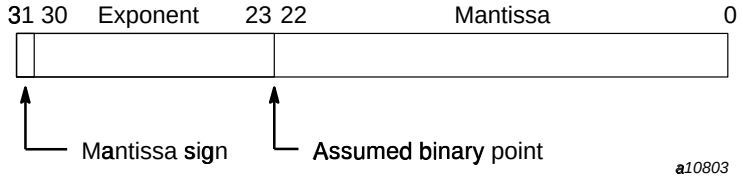

Figure 33. `COMPLEX(KIND=4)` on UNICOS/mk systems (real portion)

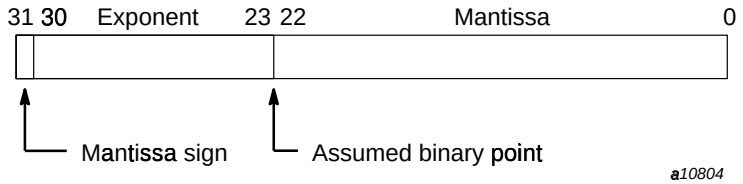

Figure 34. `COMPLEX(KIND=4)` on UNICOS/mk systems (imaginary portion)

To declare an entity to be of complex type with a total length of 64 bits, specify one of the following:

- `KIND=4`.
- `KIND=KIND(`*kind_expr*`)`, where *kind_expr* is a scalar initialization expression with a kind type parameter that evaluates to 4.

5.3.3.2 `KIND=8`

A single-precision, `KIND=8`, complex value is represented by 2 words. The first word represents the real part, and the second represents the imaginary part. Each word has the same range as a 64-bit (or `KIND=8`) real value.

Range: -2^{-1021} .LE. I < 2^{1024} or approximately -10^{-308} .LE. I < 10^{308}

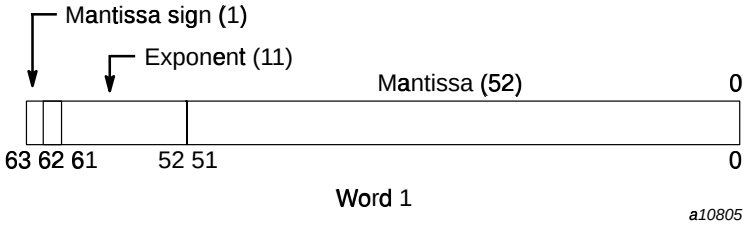

Figure 35. `COMPLEX(KIND=8)` on UNICOS/mk systems (real portion)

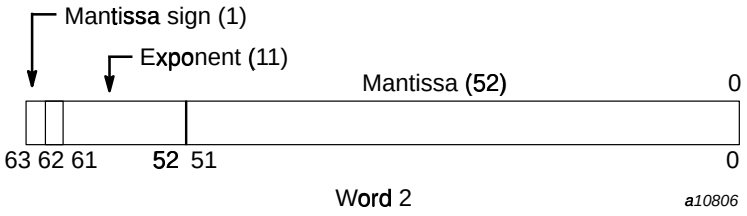

Figure 36. `COMPLEX(KIND=8)` on UNICOS/mk systems (imaginary portion)

To declare an entity to be of single-precision, complex type, specify one of the following:

- `KIND=8`.

- `KIND=KIND(`*kind_expr*`)`, where *kind_expr* is a scalar initialization expression with a kind type parameter that evaluates to 8.

5.3.4 Character Type

Characters are represented by 8-bit ASCII codes. On UNICOS/mk systems, the codes are packed 8 per word.

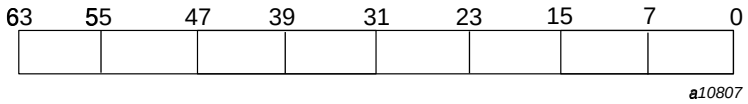

Figure 37. Character type

The CF90 compiler does not support a nondefault character type. The only kind value supported is 1.

5.3.5 Logical Type

A logical variable uses one word. Its value is true if the numeric value in the word is nonzero, and it is false if the numeric value in the word is zero.

Note: SGI does not guarantee a particular internal representation of logical values on any machine or system; the CF90 compiler is designed on the assumption that logical values will be used only as described in the Fortran standard. Therefore, it is not good programming practice to use logical values as numbers or vice versa.

To declare an entity to be of logical type, you can specify one of the following:

- `KIND=1`, `KIND=2`, `KIND=4`, or `KIND=8`.
- `KIND=KIND(`*kind_expr*`)`, where *kind_expr* is a scalar initialization expression with a kind type parameter that evaluates to 1, 2, 4, or 8.

On UNICOS/mk systems, all `KIND=1`, `2`, and `4` occupy 32 bits. The `KIND=8` specification occupies 64 bits.

5.3.6 Cray Character Pointers

Cray character pointers include a byte address and a byte length field. On UNICOS/mk systems, character pointers are 128-bit objects, as follows:

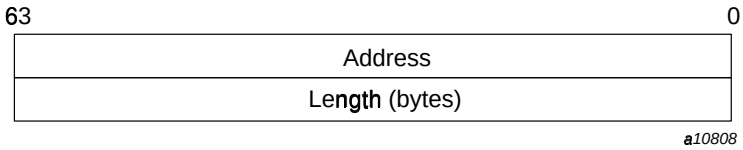

Figure 38. Cray character pointers on UNICOS/mk systems

5.4 Data Representation for Cray T90 Systems That Support IEEE Floating-point Arithmetic

The following sections describe the representation of data on Cray T90 systems that support IEEE floating-point arithmetic.

5.4.1 Integer Type

The following sections describe integer data representation of `KIND=1`, `2`, `4`, and `8` on Cray T90 systems that support IEEE floating-point arithmetic.

5.4.1.1 `KIND=1`, `KIND=2`, or `KIND=4`

Range: $-2^{31} < I < 2^{31}$ or approximately $-10^9 < I < 10^9$

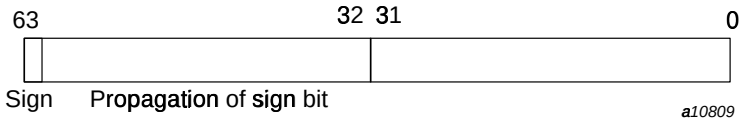

Figure 39. Integer `KIND=1`, `2`, or `4` on Cray T90 systems that support IEEE floating-point arithmetic

To declare 32-bit integers, specify one of the following:

- `KIND=1`, `KIND=2`, or `KIND=4`.

- `KIND=KIND(`*kind_expr*`)`, where *kind_expr* is a scalar initialization expression with a kind type parameter that evaluates to 1, 2, or 4.

5.4.1.2 `KIND=8`

By default, the range for `INTEGER(KIND=8)` operations is $-2^{63} < I < 2^{63}$ or approximately $-10^{18} < I < 10^{18}$. When fast integer operations are specified on the `f90`(1) command line, the range for `INTEGER(KIND=8)` operations is $-2^{50} < I < 2^{50}$ or approximately $-10^{15} < I < 10^{15}$.

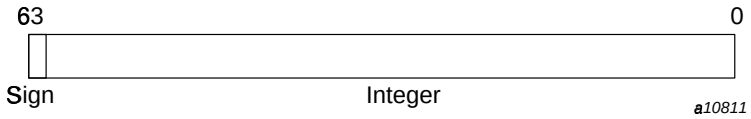

Figure 40. Default `INTEGER(KIND=8)` on Cray T90 systems that support IEEE floating-point arithmetic

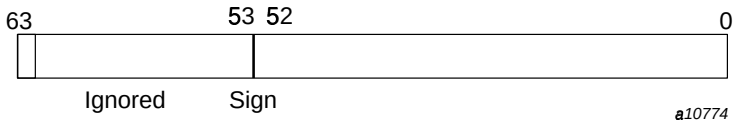

Figure 41. Fast operations with `INTEGER(KIND=8)` on Cray T90 systems that support IEEE floating-point arithmetic

To declare 64-bit integers, specify one of the following:

- `KIND=8`.
- `KIND=KIND(`*kind_expr*`)`, where *kind_expr* is a scalar initialization expression with a kind type parameter that evaluates to 8.

5.4.2 Real Type

The following sections describe real data representation of `KIND=4`, `8`, and `16` on Cray T90 systems that support IEEE floating-point arithmetic. Real (floating-point) numbers are represented in a packed representation of a sign, an exponent (power of 2), and a binary mantissa.

5.4.2.1 KIND=4 and KIND=8

Range: -2^{-1021} .LE. I < 2^{1024} or approximately -10^{-308} .LE. I < 10^{308}

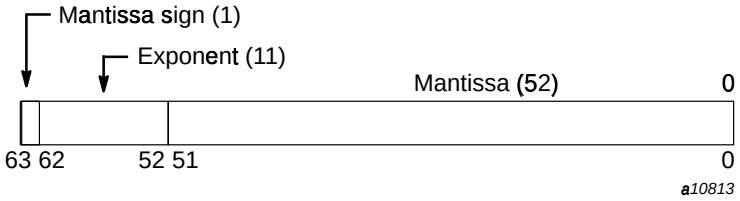

Figure 42. Real KIND=4 or 8 on Cray T90 systems that support IEEE floating-point arithmetic

To declare 64-bit reals, specify one of the following:

- KIND=4 or KIND=8.
- KIND=KIND(*kind_expr*), where *kind_expr* is a scalar initialization expression with a kind type parameter that evaluates to 4 or 8.

For additional information on how real data is represented on Cray T90 systems that support IEEE floating-point arithmetic, see "*Notes on real data type representation*" in Section 5.3.2.1, page 114. The information presented there for UNICOS/mk systems applies to Cray T90 systems that support IEEE floating-point arithmetic.

Note that a real data object with KIND=4 has the same internal representation as a real data object with KIND=8. Numeric inquiry functions on a real data object with KIND=4 return different values than on a real data object with KIND=8. A numeric operation on a real data object with KIND=4 returns the same result as the same numeric operation on a real data object with KIND=8.

5.4.2.2 KIND=16

Double precision, REAL(KIND=16), values are represented in 2 words on Cray T90 systems that support IEEE floating-point arithmetic.

Range: -2^{-16381} .LE. I < 2^{16384} or approximately -10^{-4932} .LE. I < 10^{4932}

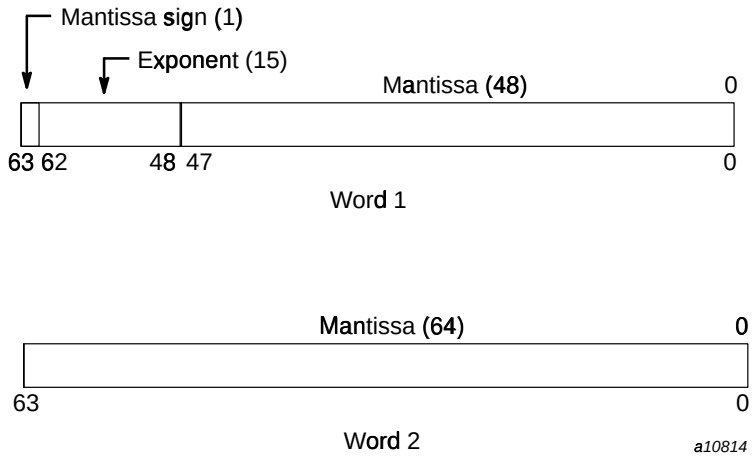

Figure 43. `REAL(KIND=16)` on Cray T90 systems that support IEEE floating-point arithmetic

To declare 64-bit reals, specify one of the following:

- `KIND=16`.
- `KIND=KIND(`*kind_expr*`)`, where *kind_expr* is a scalar initialization expression with a kind type parameter that evaluates to 16.

5.4.3 Complex Type

The following sections describe complex data representation of `KIND=4`, `8`, and `16` on Cray T90 systems that support IEEE floating-point arithmetic. A complex value has two parts. The first part represents the real part, and the second represents the imaginary part. Each word has the same range as a real value.

5.4.3.1 `KIND=4` and `KIND=8`

A single-precision, `KIND=4` or `KIND=8`, complex value is represented by 2 words. The first word represents the real part, and the second represents the imaginary part. Each word has the same range as a real value.

Range: -2^{-1021} .LE. I < 2^{1024} or approximately -10^{-308} .LE. I < 10^{308}

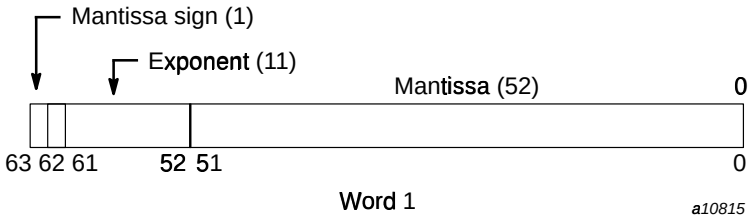

Figure 44. Complex KIND=8 or 4 on Cray T90 systems that support IEEE floating-point arithmetic (real portion)

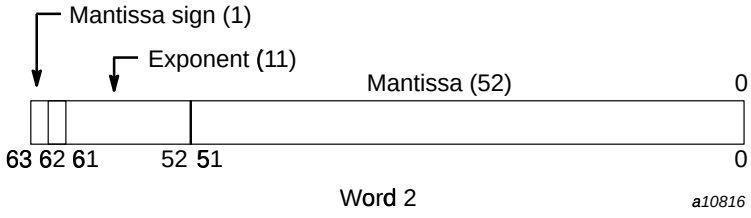

Figure 45. Complex KIND=8 or 4 on Cray T90 systems that support IEEE floating-point arithmetic (imaginary portion)

To declare an entity to be of single-precision, complex type, specify one of the following:

- KIND=4 or KIND=8.
- KIND=KIND(*kind_expr*), where *kind_expr* is a scalar initialization expression with a kind type parameter that evaluates to 4 or 8.

Note that a complex data object with KIND=4 has the same internal representation as a complex data object with KIND=8. Numeric inquiry functions on a complex data object with KIND=4 return different values than on a complex data object with KIND=8. A numeric operation on a complex data object with KIND=4 returns the same result as the same numeric operation on a complex data object with KIND=8.

5.4.3.2 KIND=16

A double-precision, KIND=16, complex value is represented by 4 words. The first two words represent the real part, and the second two words represent the imaginary part. Each word has the same range as a real value.

Range: -2^{-16381} .LE. I < 2^{16384} or approximately -10^{-4932} .LE. I < 10^{4932}

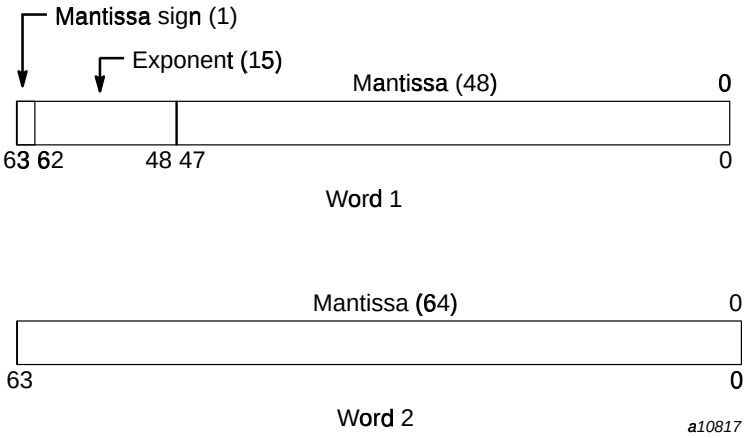

Figure 46. `COMPLEX(KIND=16)` on Cray T90 systems that support IEEE floating-point arithmetic (real portion)

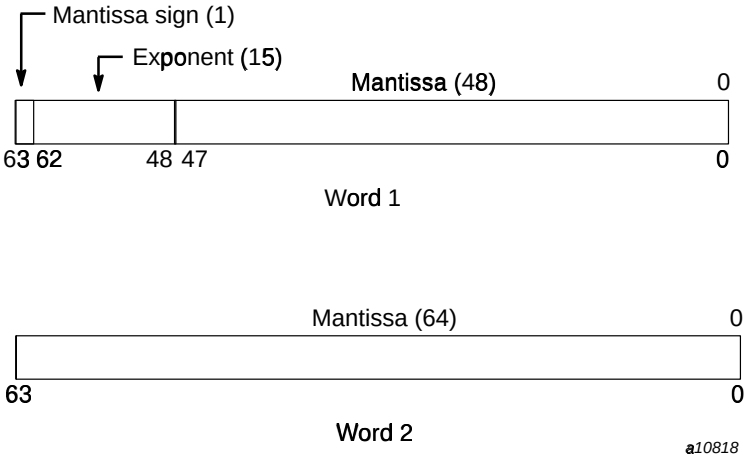

Figure 47. `COMPLEX(KIND=16)` on Cray T90 systems that support IEEE floating-point arithmetic (imaginary portion)

To declare an entity to be of double-precision, complex type, specify one of the following:

- `KIND=16`.
- `KIND=KIND(`*kind_expr*`)`, where *kind_expr* is a scalar initialization expression with a kind type parameter that evaluates to 16.

5.4.4 Character Type

Characters are represented by 8-bit ASCII codes. The codes are packed 8 per word.

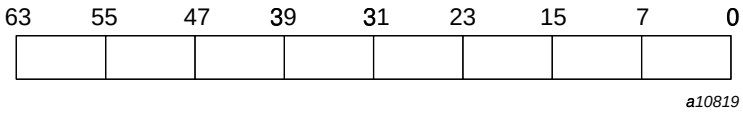

Figure 48. Character type

The CF90 compiler does not support a nondefault character type. The only kind value supported is 1.

5.4.5 Logical Type

A logical variable uses one word. Its value is true if the numeric value in the word is nonzero, and it is false if the numeric value in the word is zero.

Note: SGI does not guarantee a particular internal representation of logical values on any machine or system; the CF90 compiler is designed on the assumption that logical values will be used only as described in the Fortran standard. Therefore, it is not good programming practice to use logical values as numbers or vice versa.

To declare an entity to be of logical type, you can specify one of the following:

- `KIND=1`, `KIND=2`, `KIND=4`, or `KIND=8`.
- `KIND=KIND(`*kind_expr*`)`, where *kind_expr* is a scalar initialization expression with a kind type parameter that evaluates to 1, 2, 4, or 8.

On Cray T90 systems that support IEEE floating-point arithmetic, all `KIND=1`, `KIND=2`, and `KIND=4` specifications occupy 32 bits. The `KIND=8` specification occupies 64 bits.

5.4.6 Cray Character Pointers

Cray character pointers are two words in length. The first word includes an offset and an address. The second word includes the byte length field.

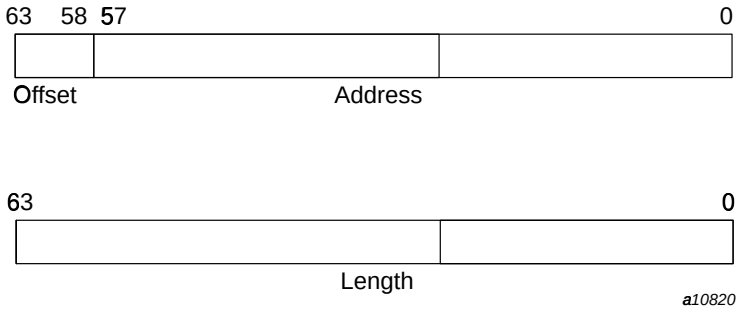

Figure 49. Cray character pointer for Cray T90 systems that support IEEE floating-point arithmetic

5.5 Storage Issues

This section describes how the CF90 and MIPSpro 7 Fortran 90 compilers use storage, including how these compilers accommodate programs that use overindexing.

Note: The information in this section assumes that you are using the default data representations.

On UNICOS/mk systems, specifying `-i 32` or `-s default32` on the `f90`(1) command line changes the storage and data representation of all noncharacter data types. This affects data that is storage sequence-associated. Mixing data types is not recommended when these command line options are used.

On IRIX systems, the following options to the `f90`(1) command affect storage and data representation:

- `-d16` changes default double precision and double complex to 128 bits
- `-i4` changes default integer and logical to 32 bits
- `-i8` changes default integer and logical to 64 bits
- `-n32` and `-64` change pointer sizes and the maximum amount of addressable memory
- `-r4` changes default real and complex to 32 bits/64 bits
- `-r8` changes default real and complex to 64 bits/128 bits

5.5.1 Storage Units and Sequences

A *numeric storage unit* can be one of the following:

- A word on UNICOS and UNICOS/mk systems of 64 bits.
- A word on IRIX systems of 32 bits.

A *character storage unit* is an 8-bit byte.

A *storage sequence* is a contiguous group of storage units with a consecutive series of addresses. Each array and each common block is stored in a storage sequence. The size of a storage sequence is the number of storage units it contains. Two storage sequences are *associated* if they share at least one storage unit.

All nondefault types have an unspecified storage unit. The `-s default32` option on the `f90`(1) command line changes the number of bits in a numeric storage unit for UNICOS/mk systems. There is no longer a relationship between storage units after this command line option is used.

The following list shows the storage units for the default types on UNICOS systems:

Type	Storage units
Integer	1
Real (single precision)	1
Real (double precision)	2
Complex	2
Logical	1

Complex values occupy twice the storage of real values. The real portion of the complex value occupies the first half of the total storage; the imaginary portion of the complex value occupies the second half of the total storage, as follows:

- On UNICOS and UNICOS/mk systems, a double precision or complex value (`KIND=4` or `KIND=8`) uses a storage sequence of two numeric storage units. The first storage unit contains the most significant bits of a double-precision value or the real part of a complex value. The second storage unit contains the least significant bits of a double-precision value or the imaginary part of a complex value. Double precision and double complex data types are not supported on UNICOS/mk systems.

  On IRIX systems, a double-precision value uses a storage sequence of 8 or 16 bytes. Depending on the `KIND=` specification, a complex value uses 8, 16, or 32 bytes. The first half of the bytes used contains the most significant bits of a double-precision value or the real part of a complex value. The last half of the bytes used contains the least significant bits of a double-precision value or the imaginary part of a complex value.

- On UNICOS and UNICOS/mk systems, a double-complex value occupies 4 words of storage; the first 2 words contain the real part of the complex value, and the second 2 words contain the imaginary part.

  On IRIX systems, a double-complex value occupies 16 bytes of storage; the first 8 bytes contain the real part of the complex value, and the second 8 bytes contain the imaginary part.

  On IRIX systems, a quad precision complex value occupies 32 bytes of storage; the first 16 bytes contain the real part of the complex value, and the second 16 bytes contain the imaginary part.

A character value is represented as an 8-bit ASCII code, packed 8 characters per word on UNICOS and UNICOS/mk systems; this value is packed 4 characters per byte on IRIX systems. The storage size depends on the length specification of the value.

ANSI/ISO: The Fortran standard does not specify the relationship between storage units and computer words, and it does not specify any relation between default numeric and character storage units.

5.5.2 Static and Stack Storage

With static storage, any variable that is allocated memory occupies the same address throughout program execution. Allocation is determined before program execution.

Code using static storage can be used with Autotasking, multitasking, and macrotasking if variables in static storage conform to the following guidelines:

- Loops are Autotasked regardless of the presence of variables in static or stack storage. Scoping is controlled by the presence of `PRIVATE` or `SHARED` parameters on the `DOALL` Autotasking directive. If a subroutine that contains static data is called from within an autotasked loop, static data is treated as shared data, which means that the static data must be protected by `GUARD` and `ENDGUARD` Autotasking directives.
- Variables in static storage can be read when loops are multitasked and macrotasked. If a loop modifies variables in static storage, you must use guards (`GUARD` and `ENDGUARD` Autotasking directives) or locks (`LOCKON()` and `LOCKOFF()` calls) to protect the variables.

For more information on Autotasking directives, see the *CF90 Commands and Directives Reference Manual*, or the *MIPSpro 7 Fortran 90 Commands and Directives Reference Manual*. For more information on locks, see the `LOCKON`(3F) or `LOCKOFF`(3F) man pages.

Stack storage is the default storage allocation for the CF90 compiler on UNICOS and UNICOS/mk systems. On IRIX systems, stack storage is the MIPSpro 7 Fortran 90 default for all subprograms, but static storage is the default for items that require 256 bits of storage in a main program. The stack is an area of memory where storage for variables is allocated when a subprogram or procedure begins execution. These variables are released when execution completes. The stack expands and contracts as procedures are entered and exited. Autotasking and recursion require a stack.

When stack storage is used, the value of a variable is not saved between invocations of a subprogram unless it is specified in a `SAVE` or `DATA` statement. When `f90 -e v` (UNICOS and UNICOS/mk systems) or `f90 -static` (IRIX systems) is specified, all user variables are treated as if they appeared in a `SAVE`

statement. When `-e v` or `-static` is in effect, compiler-generated temporary variables and the calling sequence are still allocated to the stack.

Note: If `f90 -e i` is specified, variables are reset for each invocation of a subprogram, even in static storage. Therefore, the `SAVE` or `DATA` statement is necessary to preserve the value of a variable between invocations. This information applies only to UNICOS and UNICOS/mk systems.

The way in which the amount of memory available for the stack is determined depends on your platform. On UNICOS and UNICOS/mk systems, it is determined by the `STACK` directive, available with the `segldr`(1) or `cld`(1) loaders; see the `segldr`(1) or `cld`(1) man pages for more information. On IRIX systems, you can use the `limit`(1) command to change the amount of stack space that a program is allowed; see the `limit`(1) man page for more information.

A *heap* is memory that, like a stack, is dynamically allocated; it is used internally.

The CF90 and MIPSpro 7 Fortran 90 compilers allocate variables to storage according to the following criteria:

- Variables in common blocks are always allocated in the order in which they appear in `COMMON` statements.
- Data in modules are statically allocated.
- User variables that are defined or referenced in a program unit, and that also appear in `SAVE` or `DATA` statements, are allocated to static storage, but not necessarily in the order shown in your source program.
- Other referenced user variables are assigned to the stack. If `-e v` (UNICOS and UNICOS/mk systems) or `-static` (IRIX systems) is specified on the `f90`(1) command line, referenced variables are allocated to static storage. This allocation does not necessarily depend on the order in which the variables appear in your source program.
- Compiler-generated variables are assigned to a register or to memory (to the stack or heap), depending on how the variable is used. Compiler-generated variables include `DO`-loop trip counts, dummy argument addresses, temporaries used in expression evaluation, argument lists, and variables storing adjustable dimension bounds at entries.
- Automatic objects may be allocated to either the stack or to the heap, depending on how much stack space is available when the objects are allocated.

- Heap or stack allocation can be used for `TASK COMMON` variables and some compiler-generated temporary data such as automatic arrays and array temporaries.

  Note: Unreferenced user variables not appearing in `COMMON` statements are not allocated.

5.5.3 Dynamic Memory Allocation (UNICOS Systems Only)

Many FORTRAN 77 programs contain a memory allocation scheme that expands an array in a common block located in central memory at the end of the program. This practice of expanding a blank common block or expanding a dynamic common block (sometimes referred to as *overindexing*) causes conflicts between user management of memory and the dynamic memory requirements of UNICOS libraries. It is recommended that you modify programs rather than expand blank common blocks, particularly when migrating from other environments.

Figure 50, page 133, shows the structure of a program under the UNICOS operating system in relation to expanding a blank common block. In both figures, the user area includes code, data, and common blocks.

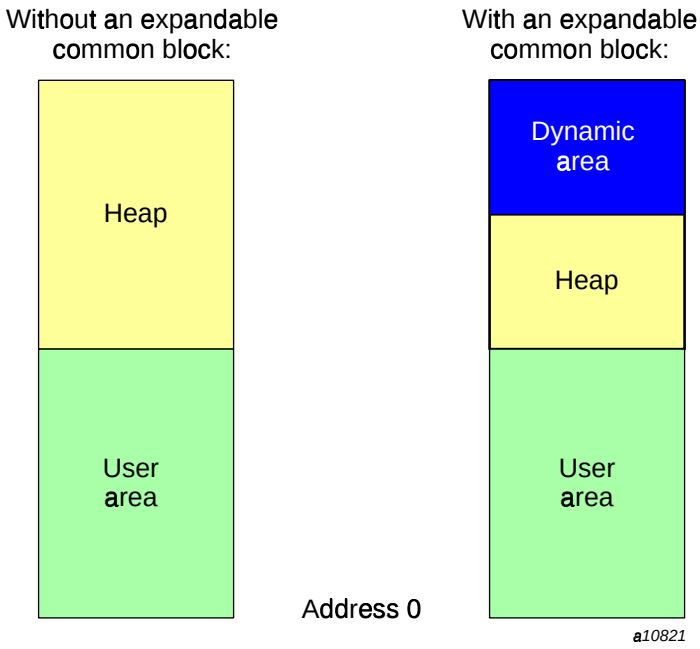

Figure 50. Memory use under UNICOS

There are two ways to change your code. The standard method, shown in Section 5.5.3.1 is preferred.

5.5.3.1 Changing Your Code: Standard Method

You can use the ALLOCATE statement to dynamically allocate an array. Use the following three-step process:

1. For arrays that expand in a common block, define Fortran allocatable arrays in a Fortran module.
2. Replace the common block definition in all source files that use the global array with a USE statement.
3. Use the ALLOCATE statement in place of any calls to the MEMORY routine.

Original code:

```
      PROGRAM TEST
C Puts array X in blank common:
```

```
      COMMON X(1)
...
C Adds 100000 words to blank common:
      CALL MEMORY ('UC',100000)
...
      DO 10, I=1,100000
        X(I) = RANF()
10    CONTINUE
...
```

Converted code (after steps 1 and 2):

```
      MODULE GLOBAL_DATA              ! STEP 1
      REAL, SAVE, ALLOCATABLE :: X(:)
      END MODULE
...
      PROGRAM TEST
      USE GLOBAL_DATA                 ! STEP 2
      LIMIT = 100000
      ALLOCATE (X(LIMIT))             ! STEP 3
...
      DO 10 I = 1,LIMIT
        X(I) = RANF()
10    CONTINUE
...
      END
```

5.5.3.2 Changing Your Code: Nonstandard Method

The nonstandard way to change your program is by using the following two-step process:

1. For arrays that expand in a common block, define Cray pointers that point to the first address in each array.
2. Change any calls to memory to calls to library routine HPALLOC(3).

Original code:

```
      PROGRAM TEST
C Puts array X in blank common:
      COMMON X(1)
...
C Adds 100000 words to blank common:
      CALL MEMORY ('UC',100000)
```

```
...
      DO 10, I=1,100000
        X(I) = RANF()
10    CONTINUE
...
```

Converted code (after steps 1 and 2):

```
      PROGRAM TEST
      COMMON /WORK/ IPTR
...
C Establish array location at runtime:
      POINTER (IPTR,X(1))
...
C Effective common block size:
      CALL HPALLOC (IPTR,100000,ERRCODE,0)
...
      DO 10 I=1,100000
        X(I) = RANF()
10    CONTINUE
...
```

Outmoded Features [6]

This chapter describes outmoded Fortran features that the CF90 and MIPSpro 7 Fortran 90 compilers support. These features have been replaced by alternatives that enhance the portability of CF90 and MIPSpro 7 Fortran 90 programs. None of the outmoded features described in this chapter were part of any Fortran standard; they were extensions supported in older Cray Research compilers. The outmoded features and their preferred alternatives are listed in Table 4.

Table 4. Outmoded features and preferred alternatives

Outmoded feature	Preferred alternative
Hollerith data	Character data.
`ENCODE` and `DECODE`	Internal files.
Asterisk character constant delimiters in formats	Apostrophe or quotation mark delimiters.
[-*b*]`X` edit descriptor	`TL` edit descriptor, `1X.`
`A` descriptor used for noncharacter data and `R` descriptor	Character type and other conventional matchings of data with descriptors.
`EOF`, `IEOF`, and `IOSTAT` functions	End-of-file specifier (`END=`) or status specifier (`IOSTAT=`).
Initialization using long strings	Replace the numeric target with a character item. Replace a Hollerith constant with a character constant.
`IMPLICIT UNDEFINED`	`IMPLICIT NONE`
Type statements with **n*	Standard type statements (`KIND=`).
Two-branch arithmetic `IF`	`IF` construct or `IF` statement.
`TASK COMMON` statement	`TASKCOMMON` compiler directive.
Indirect logical `IF`	`IF` construct or `IF` statement.
Nested loops ending with a single, labeled `END DO`	One `END DO` statement for each loop.
`DOUBLE COMPLEX` statement and related specific intrinsic function names	`COMPLEX (KIND=)` statement and standard intrinsic functions. See Section 6.9, page 152, for more information.

Outmoded feature	Preferred alternative
Bitwise intrinsic functions	Standard intrinsic functions. See Section 6.10, page 153, for more information.
CLOCK(3I), DATE(3I), and JDATE(3I) intrinsic functions	DATE_AND_TIME(3I) intrinsic subroutine.
DCOT(3M) intrinsic function	COT(3M) intrinsic function.
DFLOAT(3M) and DREAL(3M) intrinsic functions	REAL(3M) intrinsic function.
NUMARG(3I) intrinsic function	PRESENT(3I) intrinsic function for optional arguments.
RANF(3I) and RANGET(3I) intrinsic functions	RANDOM_NUMBER(3I) intrinsic subroutine.
RANSET(3I) intrinsic function	RANDOM_SEED(3I) intrinsic subroutine.
RTC(3I) intrinsic function	SYSTEM_CLOCK(3I) intrinsic subroutine.

6.1 Hollerith Type

Hollerith data is a sequence of any characters capable of internal representation as specified in Table 3, page 77. Its length is the number of characters in the sequence, including blank characters. Each character occupies a position within the storage sequence identified by one of the numbers 1, 2, 3, . . . indicating its placement from the left (position 1). Hollerith data must contain at least one character.

6.1.1 Hollerith Constants

A Hollerith constant is expressed in one of three forms. The first of these is specified as a nonzero integer constant followed by the letter H, L, or R and as many characters as equal the value of the integer constant. The second form of Hollerith constant specification delimits the character sequence between a pair of apostrophes followed by the letter H, L, or R. The third form is like the second, except that quotation marks replace apostrophes. For example:

```
Character sequence:     ABC 12
Form 1:                 6HABC 12
Form 2:                 'ABC 12'H
Form 3:                 "ABC 12"H
```

Two adjacent apostrophes or quotation marks appearing between delimiting apostrophes or quotation marks are interpreted and counted by the compiler as

a single apostrophe or quotation mark within the sequence. Thus, the sequence `DON'T USE "*"` would be specified with apostrophe delimiters as `'DON''T USE "*"'H`, and with quotation mark delimiters as `"DON'T USE ""*"""H`.

Each character of a Hollerith constant is represented internally by an 8-bit code, with up to 32 such codes allowed. This limit corresponds to the size of the largest numeric type, `COMPLEX(KIND = 16)`. The ultimate size and makeup of the Hollerith data depends on the context. If the Hollerith constant is larger than the size of the type implied by context, the constant is truncated to the appropriate size. If the Hollerith constant is smaller than the size of the type implied by context, the constant is padded with a character dependent on the Hollerith indicator. When an `H` Hollerith indicator is used, the truncation and padding is done on the right end of the constant. The pad character is the blank character code (20).

Null codes can be produced in place of blank codes by substituting the letter `L` for the letter `H` in the Hollerith forms described above. The truncation and padding is also done on the right end of the constant, with the null character code (00) as the pad character.

Using the letter `R` instead of the letter `H` as the Hollerith indicator means truncation and padding is done on the left end of the constant with the null character code (00) used as the pad character.

All of the following Hollerith constants yield the same Hollerith constant and differ only in specifying the content and placement of the unused portion of the single 64-bit entity containing the constant:

Hollerith	Internal byte, beginning on bit:							
constant	0	8	16	24	32	40	48	56
`6HABCDEF`	A	B	C	D	E	F	20_{16}	20_{16}
`'ABCDEF'H`	A	B	C	D	E	F	20_{16}	20_{16}
`"ABCDEF" H`	A	B	C	D	E	F	20_{16}	20_{16}
`6LABCDEF`	A	B	C	D	E	F	00	00
`'ABCDEF'L`	A	B	C	D	E	F	00	00
`"ABCDEF"L`	A	B	C	D	E	F	00	00
`6RABCDEF`	00	00	A	B	C	D	E	F

Hollerith	Internal byte, beginning on bit:							
constant	0	8	16	24	32	40	48	56
'ABCDEF'R	00	00	A	B	C	D	E	F
"ABCDEF"R	00	00	A	B	C	D	E	F

A Hollerith constant is limited to 32 characters except when specified in a CALL statement, a function argument list, or a DATA statement. An all-zero computer word follows the last word containing a Hollerith constant specified as an actual argument in an argument list.

A character constant of 32 or fewer characters is treated as if it were a Hollerith constant in situations where a character constant is not allowed by the standard but a Hollerith constant is allowed by the CF90 and MIPSpro 7 Fortran 90 compilers. If the character constant appears in a DATA statement value list, it can be longer than 32 characters.

6.1.2 Hollerith Values

A *Hollerith value* is a Hollerith constant or a variable that contains Hollerith data. A Hollerith value is limited to 32 characters.

A Hollerith value can be used in any operation in which a numeric constant can be used. It can also appear on the right-hand side of an assignment statement in which a numeric constant can be used. It is truncated or padded to be the correct size for the type implied by the context.

6.1.3 Hollerith Relational Expressions

Used with a relational operator, the Hollerith value e_1 is less than e_2 if its value precedes the value of e_2 in the collating sequence and is greater if its value follows the value of e_2 in the collating sequence.

The following examples are evaluated as true if the integer variable LOCK contains the Hollerith characters K, E, and Y in that order and left-justified with five trailing blank character codes:

```
3HKEY.EQ.LOCK
'KEY'.EQ.LOCK
LOCK.EQ.LOCK
'KEY1'.GT.LOCK
'KEY0'H.GT.LOCK
```

6.2 Formatted I/O and Internal Files

A formatted I/O operation defines entities by transferring data between I/O list items and records of a file. The file can be on an external media or in internal storage.

The Fortran standard provides `READ` and `WRITE` statements for both formatted external and internal file I/O. This is the preferred method for formatted internal file I/O. It is the only method for list-directed internal file I/O.

The `ENCODE` and `DECODE` statements are an alternative to standard Fortran `READ` and `WRITE` statements for formatted internal file I/O.

An internal file in standard Fortran I/O must be declared as character, while the internal file in `ENCODE` and `DECODE` statements can be any data type. A record in an internal file in standard Fortran I/O is either a scalar character variable or an array element of a character array. The record size in an internal file in an `ENCODE` or `DECODE` statement is independent of the storage size of the variable used as the internal file. If the internal file is a character array in standard Fortran I/O, multiple records can be read or written with internal file I/O. The alternative form does not provide the multiple record capability.

6.2.1 `ENCODE` Statement

The `ENCODE` statement provides a method of converting or encoding the internal representation of the entities in the output list to a character representation. The format of the `ENCODE` statement is as follows:

```
ENCODE ( n, f, dest ) [ elist ]
```

n Number of characters to be processed. Nonzero integer expression not to exceed the maximum record length for formatted records. This is the record size for the internal file.

f Format identifier. It cannot be an asterisk.

dest Name of internal file. It can be a variable or array of any data type. It cannot be an array section, a zero-sized array, or a zero-sized character variable.

elist Output list to be converted to character during the ENCODE statement.

The output list items are converted using format *f* to produce a sequence of *n* characters that are stored in the internal file *dest*. The *n* characters are packed 8 characters per word on UNICOS and UNICOS/mk systems. The *n* characters are packed 4 characters per word on IRIX systems.

An ENCODE statement transfers one record of length *n* to the internal file *dest*. If format *f* attempts to write a second record, ENCODE processing repositions the current record position to the beginning of the internal file and begins writing at that position.

An error is issued when the ENCODE statement attempts to write more than *n* characters to the record of the internal file. If *dest* is a noncharacter entity and *n* is not a multiple of 8 (for UNICOS and UNICOS/mk systems) or 4 (for IRIX systems), the last word of the record is padded with blanks to a word boundary. If *dest* is a character entity, the last word of the record is not padded with blanks to a word boundary.

Example 1: The following example assumes a machine word length of 64 bits and uses the underscore character (_) as a blank:

```
      INTEGER ZD(5), ZE(3)
      ZD(1)='THIS____'
      ZD(2)='MUST____'
      ZD(3)='HAVE____'
      ZD(4)='FOUR____'
      ZD(5)='CHAR____'
1     FORMAT(5A4)
      ENCODE(20,1,ZE)ZD
      DO 10 I=1,3
        PRINT 2,'ZE(',I,')="',ZE(I),'"'
10    CONTINUE
2     FORMAT(A,I2,A,A8,A)
      END
```

On UNICOS systems, the output is as follows:

```
>ZE( 1)="THISMUST"
>ZE( 2)="HAVEFOUR"
>ZE( 3)="CHAR____"
```

Example 2: On IRIX systems, the comparable example would be as follows:

```
      INTEGER ZD(5), ZE(3)
      ZD(1)='TH__'
      ZD(2)='IS__'
      ZD(3)='=4__'
      ZD(4)='CH__'
      ZD(5)='AR__'
1     FORMAT(5A2)
      ENCODE(10,1,ZE)ZD
      DO 10 I=1,3
      PRINT 2,'ZE(',I,')="',ZE(I),'"'
10    CONTINUE
2     FORMAT(A,I2,A,A4,A)
      END
```

The output is as follows:

```
>ZE( 1)="THIS"
>ZE( 2)="=4CH"
>ZE( 3)="AR__"
```

6.2.2 DECODE Statement

The DECODE statement provides a method of converting or decoding from a character representation to the internal representation of the entities in the input list. The format of the DECODE statement is as follows:

```
DECODE ( n, f, source ) [ dlist ]
```

n Number of characters to be processed. Nonzero integer expression not to exceed the maximum record length for formatted records. This is the record size for the internal file.

f Format identifier. It cannot be an asterisk.

source Name of internal file. It can be a variable or array of any data type. It cannot be an array section or a zero-sized array or a zero-sized character variable.

dlist Input list to be converted from character during the DECODE statement.

The input list items are converted using format *f* from a sequence of *n* characters in the internal file *source* to an internal representation and stored in the input list entities. If the internal file *source* is noncharacter, the internal file is assumed to be a multiple of 8 characters (for UNICOS and UNICOS/mk systems) or 4 characters (for IRIX systems).

Example 1: On UNICOS systems, an example of a DECODE statement is as follows:

```
      INTEGER ZD(4), ZE(3)
      ZE(1)='WHILETHI'
      ZE(2)='S HAS  F'
      ZE(3)='IVE     '
3     FORMAT(4A5)
      DECODE(20,3,ZE)ZD
      DO 10 I=1,4
         PRINT 2,'ZD(',I,')="',ZD(I),'"'
10    CONTINUE
2     FORMAT(A,I2,A,A8,A)
      END
```

The output is as follows:

```
>ZD( 1)="WHILE   "
>ZD( 2)="THIS    "
>ZD( 3)="HAS     "
>ZD( 4)="FIVE    "
```

Example 2: On IRIX systems, an example of a DECODE statement is as follows:

```
      INTEGER ZD(5), ZE(4)
      ZE(1)='WHIL'
      ZE(2)='E_IT'
      ZE(3)='=4CH'
      ZE(4)='ARS_'
      ZE(5)='RS.+'
3     FORMAT(5A3)
      DECODE(16,3,ZE)ZD
      DO 10 I=1,4
```

```
         PRINT 2,'ZD(',I,')="',ZD(I),'"'
10    CONTINUE
2     FORMAT(A,I2,A,A4,A)
      END
```

The output is as follows:

```
>ZD( 1)="WHI_"
>ZD( 2)="LE__"
>ZD( 3)="IT=_"
>ZD( 4)="4CH_"
>ZD( 5)="ARS_"
```

6.3 Edit Descriptors

The following sections show obsolete edit descriptors and outmoded uses of current descriptors.

6.3.1 Asterisk Delimiters

The asterisk was allowed to delimit a literal character constant. It has been replaced by the apostrophe and quotation mark.

h_1 h_2 ... h_n

*	Delimiter for a literal character string
h	Any ASCII character indicated by a C in Table 3, page 77 (that is, capable of internal representation)

Example:

```
*AN ASTERISK EDIT DESCRIPTOR*
```

6.3.2 Negative-valued X Descriptor

A negative value could be used with the X descriptor to indicate a move to the left. This has been replaced by the TL descriptor.

[-*b*]X

b	Any nonzero, unsigned integer constant
X	Indicates a move of as many positions as indicated by *b*

Example:

```
-55X   ! Moves current position 55 spaces left
```

6.3.3 A and R Descriptors for Noncharacter Types

The R*w* descriptor and the use of the A*w* descriptor for noncharacter data are available primarily for programs that were written before a true character type was available. Other uses include adding labels to binary files and the transfer of data whose type is not known in advance.

List items can be of type real, integer, complex, or logical. For character use, the binary form of the data is converted to or from ASCII codes. The numeric list item is assumed to contain ASCII characters when used with these edit descriptors.

Complex items use two storage units and require two A descriptors, for the first and second storage units respectively.

The A*w* descriptor works with noncharacter list items containing character data in essentially the same way as described in the *Fortran Language Reference Manual, Volume 1*. The R*w* descriptor works like A*w* with the following exceptions:

- Characters in an incompletely filled input list item are right-justified with the remainder of that list item containing binary zeros.
- Partial output of an output list item is from its rightmost character positions.

The following example shows the A*w* and R*w* edit descriptors for noncharacter data types:

```
      INTEGER IA
      LOGICAL LA
      REAL RA
      DOUBLE PRECISION DA
      COMPLEX CA
      CHARACTER*52 CHC
      CHC='ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz'
      READ(CHC,3) IA, LA, RA, DA, CA
3     FORMAT(A4,A8,A10,A17,A7,A6)
      PRINT 4, IA, LA, RA, DA, CA
```

```
4     FORMAT(1x,3(A8,'-'),A16,'-',2A8)
      READ(CHC,5) IA, LA, RA
5     FORMAT(R2,R8,R9)
      PRINT 4, IA, LA, RA
      END
```

On UNICOS and UNICOS/mk systems, the output of this program would be as follows:

```
> ABCD    -EFGHIJKL-OPQRSTUV-XYZabcdefghijklm-nopqrst uvwxyz
      ^^^^
> oooooooAB-CDEFGHIJ-LMNOPQRS-
```

The carat (^) indicates leading blanks in the use of the `A` edit descriptor. The lowercase letter `o` is used to indicate where binary zeros have been written with the `R` edit descriptor.

On IRIX systems, the output of this program would be as follows:

```
>     ABCD-    IJKL-    STUV-        fghijklm-    qrst    wxyz
 ^^^^^     ^^^^     ^^^^     ^^^^^^^^         ^^^^    ^^^^
>     AB-    GHIJ-    PQRS-
 ^^^^^         ^^^^     ^^^^
```

The binary zeros are not printable characters, so the printed output simply contains the characters without the binary zeros.

6.4 Type Declaration with Data Length

Data type declarations that include the data length are outmoded. The CF90 and MIPSpro 7 Fortran 90 compilers recognize this usage in type statements, `IMPLICIT` statements, and `FUNCTION` statements, mapping these numbers onto lengths appropriate for the target machine.

Format:

type [**n*] *v* [, *v*] ...

IMPLICIT *type* [**n*] (a_1 [-a_2] [, a_1 [-a_2]] ...)
[, *type* ...] ...

[*type* [**n*]] **FUNCTION** *fun* ([*d* [, *d*] ...])

type	Can be `INTEGER`, `REAL`, `DOUBLE PRECISION`, `COMPLEX`, or `LOGICAL`.
**n*	Data length as shown in Table 5, Table 6, page 149, and Table 7, page 149. Any other data length generates an error.
v	Name of a constant, variable, or array declarator.
a_n	A letter. A range of letters is denoted by the first and last letters of the range separated by a hyphen. A range ($a_1 - a_n$) has the same effect as a list of the letters (a_1, a_2, ... a_n).
fun	Name of the function subprogram.
d	Dummy argument representing a variable, array, or dummy procedure name.

The following tables show the data lengths for UNICOS, UNICOS/mk, and IRIX systems.

Note: On UNICOS systems, a 32-bit item or a 46-bit item is contained in a 64-bit word.

Table 5. Data length (UNICOS systems)

type *n*:	*1	*2	*4	*8	*16	*32
`INTEGER`	64-bit	64-bit	64-bit	64-bit	Error	Error
`REAL`	Error	Error	64-bit single precision	64-bit single precision	128-bit double precision	Error
`COMPLEX`	Error	Error	Error	128-bit single precision	128-bit single precision	256-bit double precision
`LOGICAL`	64-bit	64-bit	64-bit	64-bit	Error	Error
`DOUBLE PRECISION`	Error	Error	Error	Error	128-bit double precision	Error

Table 6. Data length (UNICOS/mk systems)

type *n*:	*1	*2	*4	*8	*16	*32
INTEGER	32-bit	32-bit	32-bit	64-bit	Error	Error
REAL	Error	Error	32-bit single precision[1]	64-bit double precision[2]	64-bit double precision[3]	Error
COMPLEX	Error	Error	Error	64-bit single precision[4]	64-bit single precision[5]	64-bit single precision[6]
LOGICAL	32-bit	32-bit	32-bit	64-bit	Error	Error
DOUBLE PRECISION	Error	Error	Error	Error	64-bit single precision[7]	Error

Table 7. Data length (IRIX systems)

type *n*:	*1	*2	*4	*8	*16	*32
INTEGER	8-bit	16-bit	32-bit	64-bit	Error	Error
LOGICAL	8-bit	16-bit	32-bit	64-bit	Error	Error
REAL	Error	Error	32-bit	64-bit	128-bit	Error
COMPLEX	Error	Error	32-bit	64-bit	128-bit	Error
DOUBLE PRECISION	Error	Error	Error	64-bit	Error	Error

1 This is an additional precision on a UNICOS/mk system.
2 This is a single precision on a UNICOS/mk system.
3 128-bit precision is not supported on UNICOS/mk systems.
4 This is an additional precision on a UNICOS/mk system.
5 128-bit precision is not supported on UNICOS/mk systems.
6 128-bit precision is not supported on UNICOS/mk systems.
7 128-bit precision is not supported on UNICOS/mk systems.

6.5 DATA Statement Features

The DATA statement has the following outmoded features:

- A constant need not exist for each element of a whole array named in a *data_stmt_object_list* if the array is the last item in the list.
- A Hollerith or character constant can initialize more than one element of an integer or single-precision real array if the array is specified without subscripts.

  Example 1: On a machine with 64-bit words, if an array is declared by `INTEGER A(2)`, the following DATA statements have the same effect:

```
DATA A /'1234567890123456'/
DATA A /'12345678','90123456'/
```

  Example 2: On a machine with 32-bit words, if an array is declared by `INTEGER A(2)`, the following DATA statements have the same effect:

```
DATA A /'12345678'/
DATA A /'1234','5678'/
```

  An integer or single-precision real array can be defined in the same way in a DATA implied-DO statement.

6.6 IF Statements

Outmoded IF statements are the two-branch arithmetic IF and the indirect logical IF.

6.6.1 Two-branch Arithmetic IF

A two-branch arithmetic IF statement transfers control to statement s_1 if expression *e* is evaluated as nonzero or to statement s_2 if *e* is zero. The arithmetic expression should be replaced with a relational expression, and the statement should be changed to an IF statement or an IF construct. This format is as follows:

```
IF ( e ) s1, s2
```

e Integer, real, or double-precision expression

s Label of an executable statement in the same program unit

Example:

```
IF (I+J*K) 100,101
```

6.6.2 Indirect Logical IF

An indirect logical IF statement transfers control to statement s_t if logical expression *le* is true and to statement s_f if *le* is false. An IF construct or an IF statement should be used in place of this outmoded statement. This format is as follows:

```
IF ( le ) s_t, s_f
```

le Logical expression

s_t, s_f Labels of executable statements in the same program unit

Example:

```
IF(X.GE.Y)148,9999
```

6.7 TASK COMMON Statement (UNICOS Systems Only)

When multitasking is used, some common blocks might need to be local to a task. The TASK COMMON statement declares all variables in a common block to be local to a task. If multiple tasks execute code containing the same task common block, each task will have a separate copy of the block. A common block cannot be declared both local common and task common. If a common block is declared local common in one routine and task common in another routine, the loader will generate an error.

A task common block can also be declared by the use of a COMMON statement with the TASKCOMMON compiler directive. The compiler directives are described in *CF90 Commands and Directives Reference Manual*. The directive is recommended over the TASK COMMON statement for better portability.

The keyword TASK must precede the keyword COMMON to establish a task common block. Task common blocks must be named. A task common block is allocated at task invocation.

The TASK COMMON statement has the following format:

```
TASK COMMON / cb / member_list [ , / cb / member_list ] ...
```

cb	Task common block name.
member_list	A variable name, array name, or array declarator. A member name must not be a subprogram dummy argument name.

Variables in *member_list* may appear in a DATA statement.

For information on using the -a *alloc* option to allocate storage from the f90(1) command line, see the f90(1) man page or the *CF90 Commands and Directives Reference Manual*.

6.8 Nested Loop Termination

Older Cray Research Fortran compilers allowed nested DO loops to terminate on a single END DO statement if the END DO statement had a statement label. The END DO statement is included in the Fortran standard. The Fortran standard specifies that a separate END DO statement must be used to terminate each DO loop, so allowing nested DO loops to end on a single, labeled END DO statement is an outmoded feature.

6.9 DOUBLE COMPLEX Statement (UNICOS Systems Only)

The DOUBLE COMPLEX statement is used to declare an item to be of type double complex. The format for the DOUBLE COMPLEX statement is as follows:

```
DOUBLE COMPLEX [ , attribute_list :: ] entity_list
```

Items declared as DOUBLE COMPLEX contain two double-precision entities.

When the -d p option is in effect, double complex entities are affected as follows:

- The nonstandard DOUBLE COMPLEX declaration is treated as a single-precision complex type.
- Double-precision intrinsic procedures are changed to the corresponding single-precision intrinsic procedures.

The `-e p` or `-d p` specification is used for all source files compiled with a single invocation of the `f90`(1) command. If a module is compiled separately from a program unit that uses the module, they both must be compiled with the same `-e p` or `-d p` specification.

Table 8 shows the CF90 double complex intrinsic functions and the preferred standard alternatives:

Table 8. Standard alternatives to CF90 double-complex functions

Double complex function	Fortran 90 standard alternative
`CDABS`	`ABS`(3)
`CDCOS`	`COS`(3)
`CDEXP`	`EXP`(3)
`CDLOG`	`LOG`(3)
`CDSIN`	`SIN`(3)
`CDSQRT`	`SQRT`(3)

6.10 Bitwise Logical Expressions

A *bitwise logical expression* (also called a masking expression) is an expression in which a logical operator operates on individual bits within integer, real, Cray pointer, or Boolean operands, giving a result of type Boolean. Each operand is treated as a single storage unit. This storage unit is a 64-bit word on UNICOS and UNICOS/mk systems; it is a 32-bit word on IRIX systems. The result is a single storage unit. Boolean values and bitwise logical expressions are contrasted to logical values and expressions.

Bitwise logical operators can also be written as functions; for example `A.AND.B` can be written as `AND(A,B)` and `.NOT.A` can be written as `COMPL(A)`.

The CF90 and MIPSpro 7 Fortran 90 compiler intrinsic functions that operate on Boolean values in bitwise fashion, such as shifting, parity count, and tallying 1s or leading 0s, are extensions to the Fortran standard. Generally, these bitwise functions have equivalent Fortran standard intrinsic procedures. Table 9 shows the bitwise functions and, where possible, their equivalent Fortran standard intrinsic procedures:

Table 9. Standard alternatives to CF90 and MIPSpro 7 Fortran 90 bitwise functions

Bitwise function	Fortran standard alternative
AND(3M)	IAND(3I)
COMPL(3I)	NOT(3I)
CSMG(3I)	MERGE(3I)
EQV(3M)	IEOR(3I)
MASK(3I)	IBSET(3I)
OR(3M)	IOR(3I)
NEQV(3M)	IEOR(3I)
SHIFT(3I)	ISHFT(3I), ISHFTC(3I)
SHIFTL(3I), LSHIFT(3I)	ISHFT(3I), ISHFTC(3I)
SHIFTR(3I), , RSHIFT(3I)	ISHFT(3I), ISHFTC(3I)
XOR(3M)	IEOR(3I)

If one operand is of type logical, then both operands must be of type logical; the operation performed, then, is a logical operation (not a masking operation). In a logical or masking operation, neither operand can be of type double precision or of type double complex.

Table 10, page 155, shows which data types can be used together in bitwise logical operations.

Table 10. Data types in bitwise logical operations

x_1 x_2	Integer	Real	Boolean	Pointer	Logical	Character
Integer	Masking operation, Boolean result.	Masking operation, Boolean result.	Masking operation, Boolean result.	Masking operation, Boolean result.	Not valid	Not valid[1]
Real	Masking operation, Boolean result.	Masking operation, Boolean result.	Masking operation, Boolean result.	Masking operation, Boolean result.	Not valid	Not valid[1]
Boolean	Masking operation, Boolean result.	Masking operation, Boolean result.	Masking operation, Boolean result.	Masking operation, Boolean result.	Not valid	Not valid[1]
Pointer	Masking operation, Boolean result.	Masking operation, Boolean result.	Masking operation, Boolean result.	Masking operation, Boolean result.	Not valid	Not valid[1]
Logical	Not valid	Not valid	Not valid	Not valid	Logical operation logical result	Not valid
Character	Not valid[1]	Not valid[1]	Not valid[1]	Not valid[1]	Not valid	Not valid

Notes:

1. x_1 and x_2 represent operands for a logical or bitwise expression, using operators `.NOT.`, `.AND.`, `.OR.`, `.XOR.`, `.NEQV.`, and `.EQV.`.
2. The entry "Not valid[1]" indicates that if the operand is a character operand of 32 or fewer characters, the operand is treated as a Hollerith constant and is allowed.

Bitwise logical expressions can be combined with expressions of Boolean or other types by using arithmetic, relational, and logical operators. Evaluation of an arithmetic or relational operator processes a bitwise logical expression with no type conversion. Boolean data is never automatically converted to another type.

A bitwise logical expression performs the indicated logical operation separately on each bit. The interpretation of individual bits in bitwise *multiplication_expr*s,

*summation_expr*s, and general expressions is the same as for logical expressions. The results of binary 1 and 0 correspond to the logical results TRUE and FALSE, respectively, in each of the bit positions. These values are summarized as follows:

```
.NOT. 1100          1100          1100          1100          1100
     =0011     .AND. 1010     .OR. 1010    .XOR. 1010    .EQV. 1010
                     ----          ----          ----          ----
                     1000          1110          0110          1001
```

CF90 Defined Externals [7]

This chapter describes global variables used by the CF90 compiler on UNICOS and UNICOS/mk systems

7.1 Conformance Checks

Additional `segldr`(1) and `cld`(1) directives for load time optimization and activating library features are described in the *Application Programmer's I/O Guide*.

Several `segldr`(1) directives are used to provide strict, intermediate, and minimal error checking of edit descriptors with input/output (I/O) list items during formatted `READ` and `WRITE` statements. The `NOCHK` versions provide the least error checking.

The version of `NOCHK` for formatted output is as follows:

```
% segldr -D EQUIV=$WNOCHK($WCHK)
```

The version of `NOCHK` for a formatted input is as follows:

```
% segldr -D EQUIV=$RNOCHK($RCHK)
```

For strict conformance to editing in FORTRAN 77, use the `CHK77` versions, which are as follows:

```
% segldr -D EQUIV=$WCHK77($WCHK)
% segldr -D EQUIV=$RCHK77($RCHK)
```

For strict conformance to editing in Fortran 90, use the `CHK90` versions, which are as follows:

```
% segldr -D EQUIV=$WCHK90($WCHK)
% segldr -D EQUIV=$RCHK90($RCHK)
```

The default checking is somewhat stricter than the `NOCHK` versions but is not as strict as the `CHK77` and `CHK90` versions.

7.2 Record Length

The `RECL` specifier in an `OPEN` statement can be used to specify the maximum record size for a file declared with sequential access. An alternate method is

provided through segldr(1) directives. If RECL is present, the values provided by these directives are ignored. The use of RECL for sequential access files is recommended.

To set the maximum output record length X for a file opened as a sequential formatted file, use the following specification:

```
SET=$WBUFLN:X
COMMONS=$WFDCOM:X+9
```

The default size is 267.

To set the maximum input record length X for a file opened as a sequential formatted file, use the following specification:

```
SET=$RBUFLN:X
COMMONS=$RFDCOM:X+9
```

The default size is 267.

Glossary

argument keyword

The name of a dummy (or formal) argument. This name is used in the subprogram definition; it also may be used when the subprogram is invoked to associate an actual argument with a dummy argument. Using argument keywords allows the actual arguments to appear in any order. The Fortran 90 standard specifies argument keywords for all intrinsic procedures. Argument keywords for user-supplied external procedures may be specified in a procedure interface block.

array

(1) A data structure that contains a series of related data items arranged in rows and columns for convenient access. The C shell and the `awk`(1) command can store and process arrays. (2) In Fortran 90, an object with the `DIMENSION` attribute. It is a set of scalar data, all of the same type and type parameters. The rank of an array is at least 1, and at most 7. Arrays may be used as expression operands, procedure arguments, and function results, and they may appear in input/output (I/O) lists.

association

An association permits an entity to be referenced by different names in a scoping unit or by the same or different names in different scoping units. Several kinds of association exist. The principal kinds of association are pointer association, argument association, host association, use association, and storage association.

automatic variable

A variable that is not a dummy argument but whose declaration depends on a nonconstant expression (array bounds and/or character length).

Autotasking

A trademarked process of Cray Research that automatically divides a program into individual tasks and organizes them to make the most efficient use of the computer hardware.

bottom loading

An optimization technique used on some scalar loops in which operands are prefetched during each loop iteration for use in the next iteration. The operand is available as soon as the first loop instruction executes. A prefetch is performed even during the final loop iteration, before the loop's final jump test has been performed.

cache

In a processing unit, a high-speed buffer storage that is continually updated to contain recently accessed contents of main storage. Its purpose is to reduce access time. In disk subsystems, a method the channel buffers use to buffer disk data during transfer between the devices and memory.

cache line

On Cray MPP systems, a cache line consists of four quad words, which is the maximum size of a hardware message.

CIV

A constant increment variable is a variable that is incremented only by a loop invariant value (for example, in a loop with index J, the statement J = J + K, in which K can be equal to 0, J is a CIV).

constant

A data object whose value cannot be changed. A named entity with the `PARAMETER` attribute is called a named constant. A constant without a name is called a literal constant.

construct

A sequence of statements that starts with a `SELECT CASE`, `DO`, `IF`, or `WHERE` statement and ends with the corresponding terminal statement.

control construct

An action statement that can change the normal execution sequence (such as a `GO TO`, `STOP`, or `RETURN` statement) or a `CASE`, `DO`, or `IF` construct.

critical region

On Cray MPP systems, a synchronization mechanism that enforces serial access to a piece of code. Only one PE may execute in a critical region at a time.

data entity

A data object, the result of the evaluation of an expression, or the result of the execution of a function reference (also called the function result). A data entity always has a type.

data object

A constant, a variable, or a part of a constant or variable.

declaration

A nonexecutable statement that specifies the attributes of a data object (for example, it may be used to specify the type of a variable or function result or the shape of an array).

definition

This term is used in two ways. (1) A data object is said to be defined when it has a valid or predictable value; otherwise, it is undefined. It may be given a valid value by execution of statements such as assignment or input. Under certain circumstances, it may subsequently become undefined. (2) Procedures and derived types are said to be defined when their descriptions have been supplied by the programmer and are available in a program unit.

derived type

A type that is not intrinsic (a user-defined type); it requires a type definition to name the type and specify its components. The components may be of intrinsic or user-defined types. An object of derived type is called a structure. For each derived type, a structure constructor is available to specify values. Operations on objects of derived type must be defined by a function with an interface and the generic specifier `OPERATOR`. Assignment for derived type objects is defined intrinsically, but it may be redefined by a subroutine with the `ASSIGNMENT` generic specifier. Data objects of derived type may be used as procedure arguments and function results, and they may appear in input/output (I/O) lists.

designator

Sometimes it is convenient to reference only part of an object, such as an element or section of an array, a substring of a character string, or a component of a structure. This requires the use of the name of the object followed by a selector that selects a part of the object. A name followed by a selector is called a **designator**.

entity

(1) In Open Systems Interconnection (OSI) terminology, a layered protocol machine. An entity in a layer performs the functions of the layer in one computer system, accessing the layer entity below and providing services to the layer entity above at local service access points. (2) In Fortran 90, a general term used to refer to any Fortran 90 concept (for example, a program unit, a common block, a variable, an expression value, a constant, a statement label, a construct, an operator, an interface block, a derived type, an input/output (I/O) unit, a name list group, and so on).

executable construct

A statement (such as a GO TO statement) or a construct (such as a DO or CASE construct).

expression

A set of operands, which may be function invocations, and operators that produce a value.

extent

A structure that defines a starting block and number of blocks for an element of file data.

function

Usually a type of operating-system-related function written outside a program and called in to do a specific function. Smaller and more limited in capability than a utility. In a programming language, a function is usually defined as a closed subroutine that performs some defined task and returns with an answer, or identifiable return value.

The word "function" has a more specific meaning in Fortran than it has in C. In C, it is refers to any called code; in Fortran, it refers to a subprogram that returns a value.

generic specifier

An optional component of the `INTERFACE` statement. It can take the form of an identifier, an `OPERATOR (`defined_operator`)` clause, or an `ASSIGNMENT (=)` clause.

heap

A section of memory within the user job area that provides a capability for dynamic allocation. See the `HEAP` directive in SR-0066.

inlining

The process of replacing a user subroutine or function call with the definition itself. This saves subprogram call overhead and may allow better optimization of the inlined code. If all calls within a loop are inlined, the loop becomes a candidate for vectorization and/or tasking.

intrinsic

Anything that the language defines is intrinsic. There are intrinsic data types, procedures, and operators. You may use these freely in any scoping unit. Fortran programmers may define types, procedures, and operators; these entities are not intrinsic.

local

(1) A type of scope in which variables are accessible only to a particular part of a program (usually one module). (2) The system initiating the request for service. This term is relative to the perspective of the user.

multitasking

(1) The parallel execution of two or more parts of a program on different CPUs; these parts share an area of memory. (2) A method in multiuser systems that incorporates multiple interconnected CPUs; these CPUs run their programs simultaneously (in parallel) and shares resources such as memory, storage devices, and printers. This term can often be used interchangeably with **`parallel processing`**.

name

A term that identifies many different entities of a program such as a program unit, a variable, a common block, a construct, a formal argument of a

subprogram (dummy argument), or a user-defined type (derived type). A name may be associated with a specific constant (named constant).

operator

(1) A symbolic expression that indicates the action to be performed in an expression; operator types include arithmetic, relational, and logical. (2) In Fortran 90, an operator indicates a computation that involves one or two operands. Fortran 90 defines several intrinsic operators (for example, +, -, *, /, ** are numeric operators, and .NOT., .AND., and .OR. are logical operators). Users also may define operators for use with operands of intrinsic or derived types.

overindexing

The nonstandard practice of referencing an array with a subscript not contained between the declared lower and upper bounds of the corresponding dimension for that array. This practice sometimes, but not necessarily, leads to referencing a storage location outside of the entire array.

parallel processing

Processing in which multiple processors work on a single application simultaneously.

pointer

(1) A data item that consists of the address of a desired item. (2) A symbol that moves around a computer screen under the control of the user.

procedure

(1) A named sequence of control statements and/or data that is saved in a library for processing at a later time, when a calling statement activates it; it provides the capability to replace values within the procedure. (2) In Fortran 90, procedure is defined by a sequence of statements that expresses a computation that may be invoked as a subroutine or function during program execution. It may be an intrinsic procedure, an external procedure, an internal procedure, a module procedure, a dummy procedure, or a statement function. If a subprogram contains an ENTRY statement, it defines more than one procedure.

procedure interface

In Fortran 90, a sequence of statements that specifies the name and characteristics of one or more procedures, the name and attributes of each

dummy argument, and the generic specifier by which it may be referenced if any. See **generic specifier**.

In FORTRAN 77 and Fortran 90, a **generic function** is one whose output value data type is determined by the data type of its input arguments. In FORTRAN 77, the only generic functions allowed are those that the standard defines. In Fortran 90, programmers may construct their own generic function by creating "generic interface," which is like a regular procedure interface, except that it has a "generic specifier" (the name of the generic function) after the keyword `INTERFACE`.

reduction loop

A loop that contains at least one statement that reduces an array to a scalar value by doing a cumulative operation on many of the array elements. This involves including the result of the previous iteration in the expression of the current iteration.

reference

A data object reference is the appearance of a name, designator, or associated pointer in an executable statement that requires the value of the object. A procedure reference is the appearance of the procedure name, operator symbol, or assignment symbol in an executable program that requires execution of the procedure. A module reference is the appearance of the module name in a `USE` statement.

scalar

(1) In Fortran 90, a single object of any intrinsic or derived type. A structure is scalar even if it has a component that is an array. The rank of a scalar is 0. (2) A nonvectorized, single numerical value that represents one aspect of a physical quantity and may be represented on a scale as a point. This term often refers to a floating-point or integer computation that is not vectorized; more generally, it also refers to logical and conditional (jump) computation.

scope

The region of a program in which a variable is defined and can be referenced.

scoping unit

Part of a program in which a name has a fixed meaning. A program unit or subprogram generally defines a scoping unit. Type definitions and procedure

interface bodies also constitute scoping units. Scoping units do not overlap, although one scoping unit may contain another in the sense that it surrounds it. If a scoping unit contains another scoping unit, the outer scoping unit is referred to as the host scoping unit of the inner scoping unit.

search loop

A loop that can be exited by means of an IF statement.

sequence

A set ordered by a one-to-one correspondence with the numbers 1, 2, through **n**. The number of elements in the sequence is **n**. A sequence may be empty, in which case, it contains no elements.

shared

Accessible by multiple parts of a program. Shared is a type of scope.

shell variable

A name representing a string value. Variables that are usually set only on a command line are called **parameters** (positional parameters and keyword parameters). Other variables are simply names to which a user (user-defined variables) or the shell itself may assign string values. The shell has predefined shell variables (for example, `HOME`). Variables are referenced by prefixing the variable name by a `$` (for example, `$HOME`).

software pipelining

Software pipelining is a compiler code generation technique in which operations from various loop iterations are overlapped in order to exploit instruction-level parallelism, increase instruction issue rate, and better hide memory and instruction latency. As an optimization technique, software pipelining is similar to bottom loading, but it includes additional, and more efficient, scheduling optimizations.

Cray compilers perform safe bottom loading by default. Under these conditions, code generated for a loop contains operations and stores associated with the present loop iteration and contains loads associated with the next loop iteration. Loads for the first iteration are generated in the loop preamble.

When software pipelining is performed, code generated for the loop contains loads, operations, and stores associated with various iterations of the loop. Loads and operations for first iterations are generated in the preamble to the

loop. Operations and stores for last iterations of loop are generated in the postamble to the loop.

statement keyword

A keyword that is part of the syntax of a statement. Each statement, other than an assignment statement and a statement function definition, begins with a statement keyword. Examples of these keywords are `IF`, `READ`, and `INTEGER`. Statement keywords are not reserved words; you may use them as names to identify program elements.

stripmining

A single-processor optimization technique in which arrays, and the program loops that reference them, are split into optimally-sized blocks, termed strips. The original loop is transformed into two nested loops. The inner loop references all data elements within a single strip, and the outer loop selects the strip to be addressed in the inner loop. This technique is often performed by the compiler to maximize the usage of cache memory or as part of vector code generation.

structure

A language construct that declares a collection of one or more variables grouped together under one name for convenient handling. In C and C++, a structure is defined with the `struct` keyword. In Fortran 90, a derived type is defined first and various structures of that type are subsequently declared.

subobject

Parts of a data object may be referenced and defined separately from other parts of the object. Portions of arrays are array elements and array sections. Portions of character strings are substrings. Portions of structures are structure components. Subobjects are referenced by designators and are considered to be data objects themselves.

subroutine

A series of instructions that accomplishes a specific task for many other routines. (A subsection of a user-written program of varying size and, therefore, function. It is written within the program. It is not a subsection of a routine.) It differs from a main routine in that one of its parameters must specify the location to which to return in the main program after the function has been accomplished.

TKR

An acronym that represents attributes for argument association. It represents the data type, kind type parameter, and rank of the argument.

type parameter

Two type parameters exist for intrinsic types: kind and length. The kind type parameter `KIND` indicates the decimal range for the integer type, the decimal precision and exponent range for the real and complex types, and the machine representation method for the character and logical types. The length type parameter `LEN` indicates the length of a character string.

variable

(1) A name that represents a string value. Variables that usually are set only on a command line are called parameters. Other variables are simply names to which the user or the shell may assign string values. (2) In Fortran 90, data object whose value can be defined and redefined. A variable may be a scalar or an array. (3) In the shell command language, a named parameter. See also **`shell variable`**.

Index