

Fortran Language Reference Manual,
Volume 3

SR-3905 3.0.1

Copyright © 1993, 1997 Cray Research, Inc. All Rights Reserved. This manual or parts thereof may not be reproduced in any form unless permitted by contract or by written permission of Cray Research, Inc.

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

Portions of this product may still be in development. The existence of those portions still in development is not a commitment of actual release or support by Cray Research, Inc. Cray Research, Inc. assumes no liability for any damages resulting from attempts to use any functionality or documentation not officially released and supported. If it is released, the final form and the time of official release and start of support is at the discretion of Cray Research, Inc.

Autotasking, CF77, CRAY, Cray Ada, CraySoft, CRAY Y-MP, CRAY-1, CRInform, CRI/*TurboKiva*, HSX, LibSci, MPP Apprentice, SSD, SUPERCLUSTER, UNICOS, and X-MP EA are federally registered trademarks and Because no workstation is an island, CCI, CCMT, CF90, CFT, CFT2, CFT77, ConCurrent Maintenance Tools, COS, Cray Animation Theater, CRAY APP, CRAY C90, CRAY C90D, Cray C++ Compiling System, CrayDoc, CRAY EL, CRAY J90, CRAY J90se, CrayLink, Cray NQS, Cray/REELibrarian, CRAY S-MP, CRAY SSD-T90, CRAY T90, CRAY T3D, CRAY T3E, CrayTutor, CRAY X-MP, CRAY XMS, CRAY-2, CSIM, CVT, Delivering the power . . ., DGauss, Docview, EMDS, GigaRing, HEXAR, IOS, ND Series Network Disk Array, Network Queuing Environment, Network Queuing Tools, OLNETH, RQS, SEGLDR, SMARTE, SUPERLINK, System Maintenance and Remote Testing Environment, Trusted UNICOS, UNICOS MAX, and UNICOS/mk are trademarks of Cray Research, Inc.

Adapted with permission of McGraw-Hill, Inc. from the FORTRAN 90 HANDBOOK, Copyright © 1992 by Walter S. Brainerd, Jeanne C. Adams, Jeanne T. Martin, Brian T. Smith, and Jerrold L. Wagener. All rights reserved. Cray Research, Inc. is solely responsible for the content of this work.

DynaWeb is a trademark of Electronic Book Technologies, Inc. PostScript is a trademark of Adobe Systems, Inc. MIPS is a registered trademark and MIPSpro is a trademark of MIPS Technologies, Inc. Silicon Graphics, the Silicon Graphics logo, and IRIS are registered trademarks and IRIX is a trademark of Silicon Graphics, Inc. SPARC is a trademark of SPARC International, Inc. Sun Microsystem is a trademark of Sun Microsystems, Inc. TotalView is a trademark of Bolt Baranek and Newman Inc. UNIX is a registered trademark in the United States and other countries, licensed exclusively to X/Open Limited. UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited. X/Open is a registered trademark, and the X device is a trademark, of X/Open Company Ltd.

The UNICOS operating system is derived from UNIX® System V. The UNICOS operating system is also based in part on the Fourth Berkeley Software Distribution (BSD) under license from The Regents of the University of California.

New Features

Fortran Language Reference Manual, Volume 3

SR-3905 3.0.1

This manual describes the Fortran 90 language as implemented by the Cray Research CF90 compiler, releases 3.0 and 3.0.1, and the MIPSpro 7 Fortran 90 compiler, revision 7.2.

Revision 3.0 of the Fortran language reference manual set describes the following features, all of which are extensions to the Fortran 90 standard:

- **AUTOMATIC statement and attribute.** The `AUTOMATIC` statement and attribute allow you to declare that an array or variable is based on the stack. Items declared as `AUTOMATIC` become undefined when the procedure that contains them is exited.
- **The `<>`, or `.LG.`, operator.** This logical operator is suggested by the IEEE standard for floating-point arithmetic.
- **PURE and ELEMENTAL keywords for subroutines and functions.** These features assist programmers interested in parallel processing. `PURE` declares that the procedure has no side effects. `ELEMENTAL` declares that the procedure contains scalar dummy arguments but that they may correspond to array actual arguments. When the actual arguments are array-valued, the function result is the same as the result that would have been obtained by operating on each element of the actual argument array independently.
- **An optional `[DIM=]dim` argument has been added to the `MAXLOC(3I)` and `MINLOC(3I)` intrinsic procedures.**
- **On UNICOS systems, the `RESHAPE(3I)` intrinsic procedure vectorizes.**
- **The following intrinsic procedures vectorize on UNICOS/mk systems: `LOG(3M)`, `EXP(3M)`, `SQRT(3M)`, `RANF(3I)`, `SIN(3M)`, `POPCNT(3I)`, and `COS(3M)`.** For more information on vectorization on UNICOS/mk systems, see *CF90 Commands and Directives Reference Manual*, publication SR-3901.

This manual is one of several manuals that describe the Fortran 90 language as implemented by the CF90 compiler and the MIPSpro 7 Fortran 90 compiler. If information on one of the preceding features cannot be found in this manual, please see the other manuals in the Fortran language reference manual set.

Revision 3.0.1, which is provided in online form only, contains corrections and features to support the CF90 3.0.1 release and the MIPSpro 7 Fortran 90 7.2 release. Changes to support this revision are noted with change indicators.

Record of Revision

<i>Version</i>	<i>Description</i>
2.0	November 1995 Original Printing. The sections in this manual previously appeared in the <i>CF90 Fortran Language Reference Manual</i> , revision 1.0, publication SR-3902, and the <i>CF90 Commands and Directives Reference Manual</i> , publication SR-3901.
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.

Contents

	<i>Page</i>
Preface	xiii
Related CF90 and MIPSpro 7 Fortran 90 compiler publications	xiv
CF90 and MIPSpro 7 Fortran 90 compiler messages	xiv
CF90 and MIPSpro 7 Fortran 90 compiler man pages	xv
Related Fortran publications	xv
Related publications	xvi
Ordering publications	xvi
Conventions	xvii
BNF conventions	xviii
Reader comments	xx
Fortran 90 Syntax [1]	1
Syntax form	1
Syntax rules expressed in BNF	1
Definition syntax symbol: is	2
Alternative syntax symbol: or	2
Optional symbol: []	3
Symbol for repeated items: [].	3
Syntax rule continuation	3
Assumed syntax rules	4
Example BNF syntax	4
Constraints	5
Identifying numbers	5
Syntax rules and constraints	5
Introduction	5
SR-3905 3.0.1	iii

	<i>Page</i>
Fortran terms and concepts	6
Characters, lexical tokens, and source form	10
Intrinsic and derived data types	12
Data object declarations and specifications	17
Use of data objects	28
Expressions and assignment	32
Execution control	37
Input/Output (I/O) statements	43
I/O editing	49
Program units	51
Procedures	53
Intrinsic procedures	59
Scope, association, and definition	59
Cross-references	59
Decremental Features [2]	79
Deleted features	79
Obsolescent features	79
Alternate return	80
PAUSE statement	80
ASSIGN and assigned GO TO statements	80
Assigned FORMAT specifiers	81
H editing	81
Character Set [3]	83
Extensions and Differences [4]	89
FORTRAN 77 and Fortran 90 differences	89
Fortran 90 and end-of-record action	90
Fortran 90 and new intrinsic procedures	90

	<i>Page</i>
Fortran 90 and G edit descriptor output differences	92
Fortran 90 and list-directed output differences	93
Incompatibilities with extensions	93
Namelist I/O	93
Differences between CF90 and CF77 namelist functionality	93
Similarities between CF90 and CF77 namelist input	97
Differences between CF90 and CF77 namelist output	97
Portability between CF77 and CF90 namelist	99
List-directed I/O	100
Delimited and undelimited character strings in list-directed I/O	100
List-directed I/O and internal files	101
List-directed I/O and Hollerith constants	101
List-directed I/O and floating-point zero	101
OPEN statement	101
INQUIRE statement	102
READ and WRITE statements	102
Differences in the G edit descriptor	102
Differences in the B, O, and Z edit descriptors	103
Implied-DO variables in an I/O list	103
Common blocks and I/O	103
CF90 restrictions on CF77 I/O extensions	104
CF90 and CF77 integrated environment differences	104
Loading CF77 and CF90 program units	104
The assign(1) command and the CF77 file attribute for the CF90 compiler	105
New I/O environment	105
CF90 extensions to Fortran 90	105
Source forms, character sets, and compiler directives	106
Data types and constants	106

	<i>Page</i>
Declaring attributes, COMMON, DATA, EQUIVALENCE, SAVE, FUNCTION, and SUBROUTINE statements	109
Expressions and assignments	110
I/O, including FORMAT statements	111
Flow control and other statements	113
Program units, functions, subroutines, and statement functions	113
Call by value	113
Intrinsic procedures	114
CF90 and CF77 implementation differences	114
BMM intrinsic function differences (UNICOS systems only)	114
Integer types	115
Integer constants	115
Vectorization	115
Miscellaneous differences	115
Data Representation and Storage [5]	119
Data representation for UNICOS systems	119
Integer type	119
Real type	120
Normalized floating-point numbers	123
Double-precision type	123
Single-precision complex type	124
Double-precision complex type	124
Character type	125
Logical type	126
Cray character pointers	126
Data representation for IRIX systems	127
Integer type	127
KIND=1	127

	<i>Page</i>
KIND=2	128
KIND=4	128
KIND=8	129
Real type	129
KIND=4	130
KIND=8	132
KIND=16	132
Complex type	133
KIND=4	133
KIND=8	134
KIND=16	135
Character type	137
Logical type	137
Cray character pointers (deferred implementation)	137
Data representation for UNICOS/mk systems	138
Integer type	138
KIND=1, KIND=2, or KIND=4	138
KIND=8	139
Real type	139
KIND=4	139
KIND=8	141
Complex type	142
KIND=4	142
KIND=8	143
Character type	144
Logical type	144
Cray character pointers	145
Data representation for CRAY T90 systems that support IEEE floating-point arithmetic	145
Integer type	145

	<i>Page</i>
KIND=1, KIND=2, or KIND=4	145
KIND=8	146
Real type	146
KIND=4 and KIND=8	147
KIND=16	147
Complex type	148
KIND=4 and KIND=8	148
KIND=16	149
Character type	151
Logical type	151
Cray character pointers	152
Storage issues	152
Storage units and sequences	153
Static and stack storage	155
Dynamic memory allocation (UNICOS systems only)	157
Changing your code: standard method	158
Changing your code: nonstandard method	159
Outmoded Features [6]	161
Hollerith type	162
Hollerith constants	162
Hollerith values	164
Hollerith relational expressions	164
Formatted I/O and internal files	165
ENCODE statement	165
DECODE statement	167
Edit descriptors	169
Asterisk delimiters	169
Negative-valued x descriptor	169

	<i>Page</i>
A and R descriptors for noncharacter types	170
Type declaration with data length	171
DATA statement features	174
IF statements	174
Two-branch arithmetic IF	174
Indirect logical IF	175
TASK COMMON statement (UNICOS systems only)	175
Nested loop termination	176
DOUBLE COMPLEX statement (UNICOS systems only)	176
Bitwise logical expressions	177
CF90 Defined Externals [7]	181
Conformance checks	181
Record length	181
Glossary	183
Index	193
Figures	
Figure 1. Default 64-bit integers	120
Figure 2. Fast integer operations with INTEGER(KIND=8), CRAY T90 systems	120
Figure 3. Fast integer operations with INTEGER(KIND=8), UNICOS systems (except CRAY T90 systems)	120
Figure 4. Real type	121
Figure 5. Binary version of 10.0	122
Figure 6. Double-precision type	123
Figure 7. Single-precision complex type	124
Figure 8. Double-precision complex type (real portion)	125
Figure 9. Double-precision complex type (imaginary portion)	125

	<i>Page</i>
Figure 10. Character type	126
Figure 11. 32-bit addressing for UNICOS systems (except CRAY T90 systems)	127
Figure 12. 32-bit addressing for CRAY T90 systems	127
Figure 13. INTEGER(KIND=1) on IRIX systems	128
Figure 14. INTEGER(KIND=2) on IRIX systems	128
Figure 15. INTEGER(KIND=4) on IRIX systems	129
Figure 16. INTEGER(KIND=8) on IRIX systems	129
Figure 17. REAL(KIND=4) on IRIX systems	130
Figure 18. Binary version of 10.0	131
Figure 19. REAL(KIND=8) on IRIX systems	132
Figure 20. REAL(KIND=16) on IRIX systems	133
Figure 21. COMPLEX(KIND=4) on IRIX systems (real portion)	134
Figure 22. COMPLEX(KIND=4) on IRIX systems (imaginary portion)	134
Figure 23. COMPLEX(KIND=8) on IRIX systems (real portion)	135
Figure 24. COMPLEX(KIND=8) on IRIX systems (imaginary portion)	135
Figure 25. COMPLEX(KIND=16) on IRIX systems (real portion)	136
Figure 26. COMPLEX(KIND=16) on IRIX systems (imaginary portion)	136
Figure 27. Character type	137
Figure 28. 32-bit addressing on IRIX systems	138
Figure 29. Integer KIND=1, 2, or 4 on UNICOS/mk systems	138
Figure 30. INTEGER(KIND=8) on UNICOS/mk systems	139
Figure 31. REAL(KIND=4) on UNICOS/mk systems	139
Figure 32. Binary version of 10.0	141
Figure 33. REAL(KIND=8) on UNICOS/mk systems	141
Figure 34. COMPLEX(KIND=4) on UNICOS/mk systems (real portion)	142
Figure 35. COMPLEX(KIND=4) on UNICOS/mk systems (imaginary portion)	142
Figure 36. COMPLEX(KIND=8) on UNICOS/mk systems (real portion)	143

	<i>Page</i>
Figure 37. <code>COMPLEX(KIND=8)</code> on UNICOS/mk systems (imaginary portion)	143
Figure 38. Character type	144
Figure 39. Cray character pointers on UNICOS/mk systems	145
Figure 40. Integer <code>KIND=1, 2, or 4</code> on CRAY T90 systems that support IEEE floating-point arithmetic	145
Figure 41. Default <code>INTEGER(KIND=8)</code> on CRAY T90 systems that support IEEE floating-point arithmetic	146
Figure 42. Fast operations with <code>INTEGER(KIND=8)</code> on CRAY T90 systems that support IEEE floating-point arithmetic	146
Figure 43. Real <code>KIND=4 or 8</code> on CRAY T90 systems that support IEEE floating-point arithmetic	147
Figure 44. <code>REAL(KIND=16)</code> on CRAY T90 systems that support IEEE floating-point arithmetic	148
Figure 45. Complex <code>KIND=8 or 4</code> on CRAY T90 systems that support IEEE floating-point arithmetic (real portion)	149
Figure 46. Complex <code>KIND=8 or 4</code> on CRAY T90 systems that support IEEE floating-point arithmetic (imaginary portion)	149
Figure 47. <code>COMPLEX(KIND=16)</code> on CRAY T90 systems that support IEEE floating-point arithmetic (real portion)	150
Figure 48. <code>COMPLEX(KIND=16)</code> on CRAY T90 systems that support IEEE floating-point arithmetic (imaginary portion)	150
Figure 49. Character type	151
Figure 50. Cray character pointer for CRAY T90 systems that support IEEE floating-point arithmetic	152
Figure 51. Memory use under UNICOS	158

Tables

Table 1. Syntax metalanguage abbreviations	2
Table 2. Fortran 90 standard nonterminal symbols defined through BNF rules	60
Table 3. Fortran 90 standard nonterminal symbols with no BNF definition	71
Table 4. Fortran 90 standard terminal symbols	72
Table 5. Character set	83
Table 6. Outmoded features and preferred alternatives	161
Table 7. Data length (UNICOS systems)	172

	<i>Page</i>
Table 8. Data length (UNICOS/mk systems)	173
Table 9. Data length (IRIX systems)	173
Table 10. Standard alternatives to CF90 double-complex functions	177
Table 11. Standard alternatives to CF90 and MIPSpro 7 Fortran 90 bitwise functions	178
Table 12. Data types in bitwise logical operations	179

Preface

This manual describes the Fortran 90 language as implemented by the Cray Research CF90 compiler, revision 3.0, and by the MIPSpro 7 Fortran 90 compiler, revision 7.2. The CF90 and MIPSpro 7 Fortran 90 compilers implement the Fortran 90 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 9.0, or later, on the following platforms: CRAY C90 systems, CRAY J90 systems, CRAY T90 systems, CRAY Y-MP E systems, CRAY Y-MP M90, and CRAY EL systems.
- The CF90 compiler runs under UNICOS/mk 1.3, or later, on CRAY T3E systems.
- The MIPSpro 7 Fortran 90 compiler runs under IRIX 6.2, or later, on Cray Research and Silicon Graphics IRIX systems.

Note: This manual describes how the CF90 and MIPSpro 7 Fortran 90 compilers work on Cray Research UNICOS, Cray Research UNICOS/mk, and Silicon Graphics IRIX systems. Implementation of the MIPSpro 7 Fortran 90 compiler on Silicon Graphics IRIX systems is deferred.

The CF90 and MIPSpro 7 Fortran 90 compilers were developed to support the Fortran standards adopted by the American National Standards Institute (ANSI) and the International Standards Organization (ISO). These standards, commonly referred to as *the Fortran 90 standard*, are ANSI X3.198-1992 and ISO/IEC 1539:1991-1. Because the ANSI Fortran 90 standard is a superset of the FORTRAN 77 standard, the CF90 and MIPSpro 7 Fortran 90 compilers will compile code written to the FORTRAN 77 standard.

Note: The Fortran 90 standard is a substantial revision to the FORTRAN 77 language standard. Because of the number and complexity of the features, the standards organizations are continuing to interpret the Fortran 90 standard for Cray Research, Silicon Graphics, and for other vendors. To maintain conformance to the Fortran 90 standard, Cray Research and Silicon Graphics may need to change the behavior of certain CF90 and MIPSpro 7 Fortran 90 compiler features in future releases based upon the outcomes of the outstanding 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*, publication SR-2138
- *Fortran Language Reference Manual, Volume 1*, publication SR-3902. Chapters 1 through 8 correspond to sections 1 through 8 of the Fortran 90 standard.
- *Fortran Language Reference Manual, Volume 2*, publication SR-3903. Chapters 1 through 6 of this manual correspond to sections 9 through 14 of the Fortran 90 standard.
- *Fortran Language Reference Manual, Volume 3*, publication SR-3905. This manual contains CF90 and MIPSpro 7 Fortran 90 compiler information that supplements the Fortran 90 standard. The standard is the complete, official description of the language. This manual also contains the complete Fortran 90 syntax in Backus-Naur form (BNF). The syntax rules are numbered exactly as they are in the Fortran standard. There is a cross reference that lists, for each nonterminal syntactic item, the number of the rule in which it is defined and all rules in which it is referenced.

The following publications contain information specific to the CF90 compiler:

- *CF90 Ready Reference*, publication SQ-3900
- *CF90 Commands and Directives Reference Manual*, publication SR-3901

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

- *MIPSpro 7 Fortran 90 Commands and Directives Reference Manual*, publication SR-3907

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 should consult for more information on the history of Fortran and the Fortran 90 language itself:

- Adams, J., W. Brainerd, J. Martin, B. Smith, and J. Wagener. *Fortran 90 Handbook — Complete ANSI/ISO Reference*. New York, NY: Intertext Publications/Multiscience Press, Inc., 1990.
- Metcalf, M. and J. Reid. *Fortran 90 Explained*. Oxford, UK: Oxford University Press, 1990.
- American National Standards Institute. *American National Standard Programming Language Fortran*, ANSI X3.198-1992. New York, 1992.
- International Standards Organization. *ISO/IEC 1539:1991, Information technology — Programming languages — Fortran*. Geneva, 1991.

Related publications

Certain other publications from Silicon Graphics and Cray Research 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*, publication SR-0066
- *UNICOS User Commands Reference Manual*, publication SR-2011
- *UNICOS Performance Utilities Reference Manual*, publication SR-2040
- *Scientific Libraries Reference Manual*, publication SR-2081
- *Introducing the Program Browser*, publication IN-2140
- *Application Programmer's Library Reference Manual*, publication SR-2165
- *Guide to Parallel Vector Applications*, publication SG-2182
- *Introducing the Cray TotalView Debugger*, publication IN-2502
- *Introducing the MPP Apprentice Tool*, publication IN-2511
- *Application Programmer's I/O Guide*, publication SG-2168
- *Optimizing Code on Cray PVP Systems*, publication SG-2192
- *Compiler Information File (CIF) Reference Manual*, publication SR-2401

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*, part number 007-2360-006
- *MIPSpro Fortran 77 Programmer's Guide*, part number 007-2361-004
- *MIPSpro 64-bit Porting and Transition Guide*, part number 007-2391-003
- *MIPSpro Assembly Language Programmer's Guide*, part number 007-2418-002

Ordering publications

The *User Publications Catalog*, publication CP-0099, describes the availability and content of all Cray Research hardware and software documents that are available to customers. Cray Research customers who subscribe to the Cray

Inform (CRInform) program can access this information on the CRInform system.

Silicon Graphics maintains information on available publications at the following URL:

<http://www.sgi.com/Technology/TechPubs>

The preceding website contains information that allows you to browse documents online, order documents, and send feedback to Silicon Graphics.

To order a Cray Research or Silicon Graphics document, either call the Distribution Center in Mendota Heights, Minnesota, at +1-612-683-5907, or send a facsimile of your request to fax number +1-612-452-0141.

Cray Research employees may send their orders via electronic mail to `orderdsk` (UNIX system users).

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:

<u>Convention</u>	<u>Meaning</u>
<code>command</code>	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.
<i>variable</i>	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.
EXT or EXTENSION	The EXT or EXTENSION notation indicates that the feature being described is an extension to the Fortran 90 standard.

<i>scalar_</i>	When <i>scalar_</i> is the first item in a syntax description, it indicates that the item is a scalar, not an array, value.
<i>_name</i>	When <i>_name</i> 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 <code>ARRAY</code> is a name, but <code>ARRAY(I)</code> is not.
(Rnnnn)	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 <i>Fortran Language Reference Manual, Volume 3</i> , publication SR-3905. The numbering of the rules in this manual matches the numbering of the rules in the standard. The forms of the rules in this manual and the BNF syntax class terms that are used may differ from the rules and terms used in the standard.
POINTER	The term <code>POINTER</code> refers to the Fortran 90 <code>POINTER</code> attribute.
Cray pointer	The term <i>Cray pointer</i> refers to the Cray pointer data type extension.

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:

<i>goto_stmt</i>	is	<code>GOTO label</code>
------------------	-----------	-------------------------

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*, publication SR-3905

The following abbreviations are commonly used in naming nonterminal keywords:

<u>Abbreviation</u>	<u>Term</u>
<i>arg</i>	argument
<i>attr</i>	attribute
<i>char</i>	character
<i>decl</i>	declaration
<i>def</i>	definition
<i>desc</i>	descriptor
<i>expr</i>	expression
<i>int</i>	integer
<i>op</i>	operator
<i>spec</i>	specifier or specification
<i>stmt</i>	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 (-):

<i>add_op</i>	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:

R525	<i>dimension_stmt</i>	is	DIMENSION [::] <i>array_name</i> (<i>array_spec</i>) [, <i>array_name</i> (<i>array_spec</i>)] . . .
------	-----------------------	-----------	---

Reader comments

If you have comments about the technical accuracy, content, or organization of this document, please tell us. You can contact us in any of the following ways:

- Send us electronic mail at the following address:

publications@cray.com

- Contact your customer service representative and ask that an SPR or PV be filed. Use PUBLICATIONS for the group name, PUBS for the command, and NO-LICENSE for the release name.
- Call our Software Publications Group in Eagan, Minnesota, through the Customer Service Call Center, using either of the following numbers:
1-800-950-2729 (toll free from the United States and Canada)
+1-612-683-5600
- Send a facsimile of your comments to the attention of “Software Publications Group” in Eagan, Minnesota, at fax number +1-612-683-5599.

We value your comments and will respond to them promptly.

Fortran 90 Syntax [1]

This chapter contains a complete description of the Fortran 90 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 90 standard. Section 1.3, page 59, provides a cross-reference of each syntax term, the rule in which it is defined, and the rules in which it is referenced. A high-level summary of the syntax appears in the *Fortran Language Reference Manual, Volume 1*, publication SR-3902.

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

For example, all class definitions that end with *_stmt* might be used to generate a complete list of the statements in Fortran 90.

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_constants*.

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:

<i>xyz_list</i>	means	<i>xyz</i> [, <i>xyz</i>] . . .
<i>xyz_name</i>	is	a name
<i>scalar_xyz</i>	is	an <i>xyz</i> that is a scalar

1.1.8 Example BNF syntax

Consider the following example:

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

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	<i>io_unit</i>	is	<i>external_file_unit</i>
		or	*
		or	<i>internal_file_unit</i>
EXT		or	<i>unit_name</i>

1.2 Syntax rules and constraints

Each of the following sections contains the syntax rules and constraints from a section of the Fortran 90 standard. The following sections use an underscore, rather than a hyphen, as a separator; this differs from the Fortran 90 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 90 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 90 standard.

1.2.2 Fortran terms and concepts

The following syntax rules are described in section 2, "Fortran terms and concepts," of the Fortran 90 standard.

R201	<i>executable_program</i>	is	<i>program_unit</i> [<i>program_unit</i>] . . .
R202	<i>program_unit</i>	is or or or	<i>main_program</i> <i>external_subprogram</i> <i>module</i> <i>block_data</i>
R1101	<i>main_program</i>	is	[<i>program_stmt</i>] [<i>specification_part</i>] [<i>execution_part</i>] [<i>internal_subprogram_part</i>] <i>end_program_stmt</i>

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

R203	<i>external_subprogram</i>	is or	<i>function_subprogram</i> <i>subroutine_subprogram</i>
R1215	<i>function_subprogram</i>	is	<i>function_stmt</i> [<i>specification_part</i>] [<i>execution_part</i>] [<i>internal_subprogram_part</i>] <i>end_function_stmt</i>
R1219	<i>subroutine_subprogram</i>	is	<i>subroutine_stmt</i> [<i>specification_part</i>] [<i>execution_part</i>] [<i>internal_subprogram_part</i>] <i>end_subroutine_stmt</i>
R1104	<i>module</i>	is	<i>module_stmt</i> [<i>specification_part</i>] [<i>module_subprogram_part</i>] <i>end_module_stmt</i>

R1110	<i>block_data</i>	is	<i>block_data_stmt</i> [<i>specification_part</i>] <i>end_block_data_stmt</i>
R204	<i>specification_part</i>	is	[<i>use_stmt</i>] . . . [<i>implicit_part</i>] [<i>declaration_construct</i>] . . .
R205	<i>implicit_part</i>	is	[<i>implicit_part_stmt</i>] . . . <i>implicit_stmt</i>
R206	<i>implicit_part_stmt</i>	is or or or	<i>implicit_stmt</i> <i>parameter_stmt</i> <i>format_stmt</i> <i>entry_stmt</i>
R207	<i>declaration_construct</i>	is or or or or or or or or	<i>derived_type_def</i> <i>interface_block</i> <i>type_declaration_stmt</i> <i>specification_stmt</i> <i>parameter_stmt</i> <i>format_stmt</i> <i>entry_stmt</i> <i>stmt_function_stmt</i>
R208	<i>execution_part</i>	is	<i>executable_construct</i> [<i>execution_part_construct</i>] . . .
R209	<i>execution_part_construct</i>	is or or or	<i>executable_construct</i> <i>format_stmt</i> <i>data_stmt</i> <i>entry_stmt</i>
R210	<i>internal_subprogram_part</i>	is	<i>contains_stmt</i> <i>internal_subprogram</i> [<i>internal_subprogram</i>] . . .
R211	<i>internal_subprogram</i>	is or	<i>function_subprogram</i> <i>subroutine_subprogram</i>
R212	<i>module_subprogram_part</i>	is	<i>contains_stmt</i> <i>module_subprogram</i> [<i>module_subprogram</i>] . . .
R213	<i>module_subprogram</i>	is or	<i>function_subprogram</i> <i>subroutine_subprogram</i>

R214	<i>specification_stmt</i>	is	<i>access_stmt</i>
		or	<i>allocatable_stmt</i>
		or	<i>automatic_stmt</i>
		or	<i>common_stmt</i>
		or	<i>data_stmt</i>
		or	<i>dimension_stmt</i>
		or	<i>equivalence_stmt</i>
		or	<i>external_stmt</i>
		or	<i>intent_stmt</i>
		or	<i>intrinsic_stmt</i>
		or	<i>namelist_stmt</i>
		or	<i>optional_stmt</i>
		or	<i>pointer_stmt</i>
		or	<i>save_stmt</i>
		or	<i>target_stmt</i>

R215	<i>executable_construct</i>	is	<i>action_stmt</i>
		or	<i>case_construct</i>
		or	<i>do_construct</i>
		or	<i>if_construct</i>
		or	<i>where_construct</i>
R216	<i>action_stmt</i>	is	<i>allocate_stmt</i>
		or	<i>arithmetic_if_stmt</i>
		or	<i>assign_stmt</i>
		or	<i>assigned_goto_stmt</i>
		or	<i>assignment_stmt</i>
		or	<i>backspace_stmt</i>
EXT		or	<i>buffer_in_stmt</i>
EXT		or	<i>buffer_out_stmt</i>
		or	<i>call_stmt</i>
		or	<i>close_stmt</i>
		or	<i>computed_goto_stmt</i>
		or	<i>continue_stmt</i>
		or	<i>cycle_stmt</i>
		or	<i>deallocate_stmt</i>
		or	<i>endfile_stmt</i>
		or	<i>end_function_stmt</i>
		or	<i>end_program_stmt</i>
		or	<i>end_subroutine_stmt</i>
		or	<i>exit_stmt</i>
		or	<i>goto_stmt</i>
		or	<i>if_stmt</i>
		or	<i>inquire_stmt</i>
		or	<i>nullify_stmt</i>
		or	<i>open_stmt</i>
		or	<i>pause_stmt</i>
		or	<i>pointer_assignment_stmt</i>
		or	<i>print_stmt</i>
		or	<i>read_stmt</i>
		or	<i>return_stmt</i>
		or	<i>rewind_stmt</i>
		or	<i>stop_stmt</i>
		or	<i>where_stmt</i>
		or	<i>write_stmt</i>

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 90 standard.

R301	<i>character</i>	is	<i>alphanumeric_character</i>
		or	<i>special_character</i>
R302	<i>alphanumeric_character</i>	is	<i>letter</i>
		or	<i>digit</i>
		or	<i>underscore</i>
EXT		or	<i>currency_symbol</i>
EXT		or	<i>at_sign</i>

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

R303	<i>underscore</i>	is	—
EXT	<i>currency_symbol</i>	is	\$
EXT	<i>at_sign</i>	is	@
R304	<i>name</i>	is	<i>letter</i> [<i>alphanumeric_character</i>] . . .

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

R305	<i>constant</i>	is	<i>literal_constant</i>
		or	<i>named_constant</i>
R306	<i>literal_constant</i>	is	<i>int_literal_constant</i>
		or	<i>real_literal_constant</i>
		or	<i>complex_literal_constant</i>
		or	<i>logical_literal_constant</i>
		or	<i>char_literal_constant</i>
		or	<i>boz_literal_constant</i>
R307	<i>named_constant</i>	is	<i>name</i>
R308	<i>int_constant</i>	is	<i>constant</i>

Constraint: *int_constant* must be of type integer.

R309	<i>char_constant</i>	is	<i>constant</i>
------	----------------------	-----------	-----------------

Constraint: *char_constant* must be of type character.

R310	<i>intrinsic_operator</i>	is	<i>power_op</i>
		or	<i>mult_op</i>
		or	<i>add_op</i>
		or	<i>concat_op</i>
		or	<i>rel_op</i>
		or	<i>not_op</i>
		or	<i>and_op</i>
		or	<i>or_op</i>
		or	<i>equiv_op</i>
R708	<i>power_op</i>	is	**
R709	<i>mult_op</i>	is	*
		or	/
R710	<i>add_op</i>	is	+
		or	-
R712	<i>concat_op</i>	is	//
R714	<i>rel_op</i>	is	.EQ.
		or	.NE.
		or	.LT.
		or	.LE.
		or	.GT.
		or	.GE.
EXT		or	.LG.
		or	==
		or	/=
		or	<
		or	<=
		or	>
		or	>=
EXT		or	<>

R719	<i>not_op</i>	is	.NOT.
EXT		or	.N.
R720	<i>and_op</i>	is	.AND.
EXT		or	.A.
R721	<i>or_op</i>	is	.OR.
EXT		or	.O.
R722	<i>equiv_op</i>	is	.EQV.
		or	.NEQV.
EXT	<i>exclusive_disjunct_op</i>	is	.XOR.
EXT		or	.X.
R311	<i>defined_operator</i>	is	<i>defined_unary_op</i>
		or	<i>defined_binary_op</i>
		or	<i>extended_intrinsic</i>
R704	<i>defined_unary_op</i>	is	. letter [letter]
R724	<i>defined_binary_op</i>	is	. letter [letter]
R312	<i>extended_intrinsic_op</i>	is	<i>intrinsic_operator</i>

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	<i>label</i>	is	<i>digit</i> [<i>digit</i> [<i>digit</i> [<i>digit</i> [<i>digit</i>]]]]
------	--------------	-----------	---

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 90 standard.

R401	<i>signed_digit_string</i>	is	[<i>sign</i>] <i>digit_string</i>
R402	<i>digit_string</i>	is	<i>digit</i> [<i>digit</i>] . . .
R403	<i>signed_int_literal_constant</i>	is	[<i>sign</i>] <i>int_literal_constant</i>

R404	<i>int_literal_constant</i>	is	<i>digit_string</i> [<i>_kind_param</i>]
R405	<i>kind_param</i>	is	<i>digit_string</i>
		or	<i>scalar_int_constant_name</i>

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

Constraint: The value of *kind_param* must specify a representation method that the compiler allows.

R406	<i>sign</i>	is	+
		or	-
R407	<i>boz_literal_constant</i>	is	<i>binary_constant</i>
		or	<i>octal_constant</i>
		or	<i>hex_constant</i>

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

R408	<i>binary_constant</i>	is	B ' <i>digit</i> [<i>digit</i>] ... '
		or	B " <i>digit</i> [<i>digit</i>] ... "

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

R409	<i>octal_constant</i>	is	O ' <i>digit</i> [<i>digit</i>] ... '
		or	O " <i>digit</i> [<i>digit</i>] ... "

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

R410	<i>hex_constant</i>	is	Z ' <i>hex_digit</i> [<i>hex_digit</i>] . . . '
		or	Z " <i>hex_digit</i> [<i>hex_digit</i>] . . . "
R411	<i>hex_digit</i>	is	<i>digit</i>
		or	A
		or	B
		or	C
		or	D
		or	E
		or	F
R412	<i>signed_real_literal_constant</i>	is	[<i>sign</i>] <i>real_literal_constant</i>
R413	<i>real_literal_constant</i>	is	<i>significand</i> [<i>exponent_letter</i> <i>exponent</i>] [<i>_kind_param</i>]
		or	<i>digit_string</i> <i>exponent_letter</i> <i>exponent</i> [<i>_kind_param</i>]
R414	<i>significand</i>	is	<i>digit_string</i> . [<i>digit_string</i>]
		or	. <i>digit_string</i>
R415	<i>exponent_letter</i>	is	E
		or	D
		or	Q
R416	<i>exponent</i>	is	<i>signed_digit_string</i>

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 the compiler allows.

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

Constraint: The value of *kind_param* must specify a representation method that the compiler allows.

R421	<i>logical_literal_constant</i>	is	.TRUE. [<i>_kind_param</i>]
		or	.FALSE. [<i>_kind_param</i>]

Constraint: The value of *kind_param* must specify a representation method that the compiler allows.

R422	<i>derived_type_def</i>	is	<i>derived_type_stmt</i> [<i>private_sequence_stmt</i>] ... <i>component_def_stmt</i> [<i>component_def_stmt</i>] ... <i>end_type_stmt</i>
R423	<i>private_sequence_stmt</i>	is	PRIVATE
		or	SEQUENCE
R424	<i>derived_type_stmt</i>	is	TYPE [[, <i>access_spec</i>] ::] <i>type_name</i>

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: 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: 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	<i>end_type_stmt</i>	is	END TYPE [<i>type_name</i>]
------	----------------------	-----------	-------------------------------

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*.

R426	<i>component_def_stmt</i>	is	<i>type_spec</i> [[, <i>component_attr_spec_list</i>] ::] <i>component_decl_list</i>
R427	<i>component_attr_spec</i>	is or	POINTER DIMENSION (<i>component_array_spec</i>)

Constraint: No *component_attr_spec* may 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.

R428	<i>component_array_spec</i>	is or	<i>explicit_shape_spec_list</i> <i>deferred_shape_spec_list</i>
R429	<i>component_decl</i>	is	<i>component_name</i> [(<i>component_array_spec</i>)] [* <i>char_length</i>]

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*.

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

Constraint: 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* (R428) must be a constant specification expression.

R430	<i>structure_constructor</i>	is	<i>type_name</i> (<i>expr_list</i>)
R431	<i>array_constructor</i>	is	(/ <i>ac_value_list</i> /)
R432	<i>ac_value</i>	is or	<i>expr</i> <i>ac_implied_do</i>

R433	<i>ac_implied_do</i>	is	(<i>ac_value_list</i> , <i>ac_implied_do_control</i>)
R434	<i>ac_implied_do_control</i>	is	<i>ac_do_variable</i> = <i>scalar_int_expr</i> , <i>scalar_int_expr</i> [, <i>scalar_int_expr</i>]
R435	<i>ac_do_variable</i>	is	<i>scalar_int_variable</i>

Constraint: *ac_do_variable* must be a named variable.

Constraint: Each *ac_value* expression in the *array_constructor* must have the same type and 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 90 standard.

R501	<i>type_declaration_stmt</i>	is	<i>type_spec</i> [[, <i>attr_spec</i>] ... ::] <i>entity_decl_list</i>
R502	<i>type_spec</i>	is	INTEGER [<i>kind_selector</i>]
EXT		or	INTEGER* <i>length_value</i>
		or	REAL [<i>kind_selector</i>]
EXT		or	REAL* <i>length_value</i>
		or	DOUBLE PRECISION
EXT		or	DOUBLE PRECISION* <i>length_value</i>
		or	COMPLEX [<i>kind_selector</i>]
EXT		or	COMPLEX* <i>length_value</i>
		or	CHARACTER [<i>char_selector</i>]
		or	LOGICAL [<i>kind_selector</i>]
EXT		or	LOGICAL* <i>length_value</i>
		or	TYPE (<i>type_name</i>)
EXT		or	POINTER (<i>pointer_name</i> , <i>pointee_name</i> [(<i>array_spec</i>)])
			[, (<i>pointer_name</i> , <i>pointee_name</i> [(<i>array_spec</i>)])] ...
R503	<i>attr_spec</i>	is	PARAMETER
		or	<i>access_spec</i>
		or	ALLOCATABLE
EXT		or	AUTOMATIC
		or	DIMENSION (<i>array_spec</i>)
		or	EXTERNAL
		or	INTENT (<i>intent_spec</i>)
		or	INTRINSIC
		or	OPTIONAL
		or	POINTER
		or	SAVE
		or	TARGET
R504	<i>entity_decl</i>	is	<i>object_name</i> [(<i>array_spec</i>)] [* <i>char_length</i>] [= <i>initialization_expr</i>]
		or	<i>function_name</i> [* <i>char_length</i>]
R505	<i>kind_selector</i>	is	([KIND =] <i>scalar_int_initialization_expr</i>)

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, 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, a pointer, 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 a *function_name* that does not have the POINTER attribute must be an *explicit_shape_spec_list*.

Constraint: An *array_spec* for a *function_name* that does have 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, or an automatic data object.

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

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

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 the compiler allows.

R506	<i>char_selector</i>	is	<i>length_selector</i>
		or	(<code>LEN = type_param_value</code> , <code>KIND = kind_value</code>)
		or	(<i>type_param_value</i> , [<code>KIND =</code>] <i>kind_value</i>)
		or	(<code>KIND = kind_value</code> [, <code>LEN = type_param_value</code>])
R507	<i>length_selector</i>	is	([<code>LEN =</code>] <i>type_param_value</i>)
		or	* <i>char_length</i> [,]
R508	<i>char_length</i>	is	(<i>type_param_value</i>)
		or	<i>scalar_int_literal_constant</i>

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

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 the compiler allows.

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

R509	<i>type_param_value</i>	is	<i>specification_expr</i>
		or	*

Constraint: A function name must not be declared with an **type_param_value* if the function is an internal or module function, array-valued, pointer-valued, or recursive.

R510	<i>access_spec</i>	is	PUBLIC
		or	PRIVATE

Constraint: An *access_spec* attribute may appear only in the scoping unit of a module.

R511	<i>intent_spec</i>	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.

R512	<i>array_spec</i>	is	<i>explicit_shape_spec_list</i>
		or	<i>assumed_shape_spec_list</i>
		or	<i>deferred_shape_spec_list</i>
		or	<i>assumed_size_spec</i>

Constraint: The maximum rank is seven.

R513	<i>explicit_shape_spec</i>	is	[<i>lower_bound</i> :] <i>upper_bound</i>
R514	<i>lower_bound</i>	is	<i>specification_expr</i>
R515	<i>upper_bound</i>	is	<i>specification_expr</i>

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.

R516	<i>assumed_shape_spec</i>	is	[<i>lower_bound</i>] :
R517	<i>deferred_shape_spec</i>	is	:
R518	<i>assumed_size_spec</i>	is	[<i>explicit_shape_spec_list</i> ,] [<i>lower_bound</i> :] *

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

R519	<i>intent_stmt</i>	is	INTENT (<i>intent_spec</i>) [::] <i>dummy_arg_name_list</i>
------	--------------------	-----------	---

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.

R520	<i>optional_stmt</i>	is	OPTIONAL [::] <i>dummy_arg_name_list</i>
------	----------------------	-----------	--

Constraint: An *optional_stmt* may occur only in the scoping unit of a subprogram or an interface body.

R521	<i>access_stmt</i>	is	<i>access_spec</i> [[::] <i>access_id_list</i>]
R522	<i>access_id</i>	is or	<i>use_name</i> <i>generic_spec</i>

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

R523	<i>save_stmt</i>	is	SAVE [[::] <i>saved_entity_list</i>]
R524	<i>saved_entity</i>	is or	<i>object_name</i> / <i>common_block_name</i> /

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

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.

R525	<i>dimension_stmt</i>	is	DIMENSION [::] <i>array_name</i> (<i>array_spec</i>) [, <i>array_name</i> (<i>array_spec</i>)] ...
R526	<i>allocatable_stmt</i>	is	ALLOCATABLE [::] <i>array_name</i> [(<i>deferred_shape_spec_list</i>)] [, <i>array_name</i> [(<i>deferred_shape_spec_list</i>)]] ...

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*.

R527	<i>pointer_stmt</i>	is	POINTER [::] <i>object_name</i> [(<i>deferred_shape_spec_list</i>)] [, <i>object_name</i> [(<i>deferred_shape_spec_list</i>)]] ...
------	---------------------	-----------	---

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*.

R528	<i>target_stmt</i>	is	TARGET [::] <i>object_name</i> [(<i>array_spec</i>)] [, <i>object_name</i> [(<i>array_spec</i>)]] ...
------	--------------------	-----------	--

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

R529	<i>data_stmt</i>	is	DATA <i>data_stmt_set</i> [[,] <i>data_stmt_set</i>] ...
R530	<i>data_stmt_set</i>	is	<i>data_stmt_object_list</i> / <i>data_stmt_value_list</i> / [[,] <i>data_stmt_object_list</i> / <i>data_stmt_value_list</i> /] ...
R531	<i>data_stmt_object</i>	is or	<i>variable</i> <i>data_implied_do</i>
R532	<i>data_stmt_value</i>	is	[<i>data_stmt_repeat</i> *] <i>data_stmt_constant</i>
R533	<i>data_stmt_constant</i>	is or or or or	<i>scalar_constant</i> <i>signed_int_literal_constant</i> <i>signed_real_literal_constant</i> <i>structure_constructor</i> <i>boz_literal_constant</i>
EXT		or	<i>typeless_constant</i>
R534	<i>data_stmt_repeat</i>	is	<i>scalar_int_constant</i>
R535	<i>data_implied_do</i>	is	(<i>data_i_do_object_list</i> , <i>data_i_do_variable</i> = <i>scalar_int_expr</i> , <i>scalar_int_expr</i> [, <i>scalar_int_expr</i>])
R536	<i>data_i_do_object</i>	is or or	<i>array_element</i> <i>scalar_structure_component</i> <i>data_implied_do</i>

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

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

R537	<i>data_i_do_variable</i>	is	<i>scalar_int_variable</i>
------	---------------------------	-----------	----------------------------

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: 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, a pointer, 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 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 or DO variables of the containing *data_implied_dos*, and each operation must be intrinsic.

EXT	<i>typeless_constant</i>	is	<i>octal_typeless_constant</i>
		or	<i>hexadecimal_typeless_constant</i>
		or	<i>binary_typeless_constant</i>
EXT	<i>octal_typeless_constant</i>	is	<i>digit</i> [<i>digit</i>] ... B
		or	o" <i>digit</i> [<i>digit</i>] ... "
		or	o' <i>digit</i> [<i>digit</i>] ... '
		or	" <i>digit</i> [<i>digit</i>] ... "o
		or	' <i>digit</i> [<i>digit</i>] ... 'o
EXT	<i>hexadecimal_typeless_constant</i>	is	x' <i>hex_digit</i> [<i>hex_digit</i>] ... '
		or	x" <i>hex_digit</i> [<i>hex_digit</i>] ... "
		or	' <i>hex_digit</i> [<i>hex_digit</i>] ... 'x
		or	" <i>hex_digit</i> [<i>hex_digit</i>] ... "x
		or	z' <i>hex_digit</i> [<i>hex_digit</i>] ... '
		or	z" <i>hex_digit</i> [<i>hex_digit</i>] ... "
EXT	<i>binary_typeless_constant</i>	is	B' <i>bin_digit</i> [<i>bin_digit</i>] ... '
		or	B" <i>bin_digit</i> [<i>bin_digit</i>] ... "

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.

R538	<i>parameter_stmt</i>	is	PARAMETER (<i>named_constant_def_list</i>)
R539	<i>named_constant_def</i>	is	<i>named_constant</i> = <i>initialization_expr</i>
R540	<i>implicit_stmt</i>	is	IMPLICIT <i>implicit_spec_list</i>
		or	IMPLICIT NONE
R541	<i>implicit_spec</i>	is	<i>type_spec</i> (<i>letter_spec_list</i>)
R542	<i>letter_spec</i>	is	<i>letter</i> [- <i>letter</i>]

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.

R543	<i>namelist_stmt</i>	is	NAMELIST / <i>namelist_group_name</i> / <i>namelist_group_object_list</i> [[,] / <i>namelist_group_name</i> / <i>namelist_group_object_list</i>] . . .
R544	<i>namelist_group_object</i>	is	<i>variable_name</i>

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* may have the PRIVATE attribute.

R545	<i>equivalence_stmt</i>	is	EQUIVALENCE <i>equivalence_set_list</i>
R546	<i>equivalence_set</i>	is	(<i>equivalence_object</i> , <i>equivalence_object_list</i>)
R547	<i>equivalence_object</i>	is	<i>variable_name</i>
		or	<i>array_element</i>
		or	<i>substring</i>

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: 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.

R548	<i>common_stmt</i>	is	COMMON [/ [<i>common_block_name</i>] /] <i>common_block_object_list</i> [[,] / [<i>common_block_name</i>] / <i>common_block_object_list</i>] . . .
R549	<i>common_block_object</i>	is	<i>variable_name</i> [(<i>explicit_shape_spec_list</i>)]

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

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

1.2.6 Use of data objects

The following syntax rules are described in section 6, "Use of data objects," of the Fortran 90 standard.

R601	<i>variable</i>	is	<i>scalar_variable_name</i>
		or	<i>array_variable_name</i>
		or	<i>subobject</i>

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	<i>subobject</i>	is	<i>array_element</i>
		or	<i>array_section</i>
		or	<i>structure_component</i>
		or	<i>substring</i>
R603	<i>logical_variable</i>	is	<i>variable</i>

Constraint: *logical_variable* must be of type logical.

R604	<i>default_logical_variable</i>	is	<i>variable</i>
------	---------------------------------	-----------	-----------------

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

R605	<i>char_variable</i>	is	<i>variable</i>
------	----------------------	-----------	-----------------

Constraint: *char_variable* must be of type character.

R606	<i>default_char_variable</i>	is	<i>variable</i>
------	------------------------------	-----------	-----------------

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

R607	<i>int_variable</i>	is	<i>variable</i>
------	---------------------	-----------	-----------------

Constraint: *int_variable* must be of type integer.

R608	<i>default_int_variable</i>	is	<i>variable</i>
------	-----------------------------	-----------	-----------------

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

R609	<i>substring</i>	is	<i>parent_string</i> (<i>substring_range</i>)
R610	<i>parent_string</i>	is	<i>scalar_variable_name</i>
		or	<i>array_element</i>
		or	<i>scalar_structure_component</i>
		or	<i>scalar_constant</i>
R611	<i>substring_range</i>	is	[<i>scalar_int_expr</i>] : [<i>scalar_int_expr</i>]

Constraint: *parent_string* must be of type character.

R612	<i>data_ref</i>	is	<i>part_ref</i> [% <i>part_ref</i>] . . .
R613	<i>part_ref</i>	is	<i>part_name</i> [(<i>section_subscript_list</i>)]

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.

Constraint: A *part_name* to the right of a *part_ref* with nonzero rank must not have the `POINTER` attribute.

R614	<i>structure_component</i>	is	<i>data_ref</i>
------	----------------------------	-----------	-----------------

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	<i>array_element</i>	is	<i>data_ref</i>
------	----------------------	-----------	-----------------

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

R616	<i>array_section</i>	is	<i>data_ref</i> [(<i>substring_range</i>)]
------	----------------------	-----------	--

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* has nonzero rank.

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

R617	<i>subscript</i>	is	<i>scalar_int_expr</i>
R618	<i>section_subscript</i>	is or or	<i>subscript</i> <i>subscript_triplet</i> <i>vector_subscript</i>
R619	<i>subscript_triplet</i>	is	[<i>subscript</i>] : [<i>subscript</i>] [: <i>stride</i>]
R620	<i>stride</i>	is	<i>scalar_int_expr</i>
R621	<i>vector_subscript</i>	is	<i>int_expr</i>

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

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

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

R629	<i>nullify_stmt</i>	is	NULLIFY (<i>pointer_object_list</i>)
R630	<i>pointer_object</i>	is or	<i>variable_name</i> <i>structure_component</i>

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

R631	<i>deallocate_stmt</i>	is	<code>DEALLOCATE (<i>allocate_object_list</i> [, <code>STAT = stat_variable</code>])</code>
------	------------------------	-----------	--

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 90 standard.

Note: The language of the Fortran 90 standard is presented in this subsection in its original form. Chapter 7 of the *Fortran Language Reference Manual, Volume 1*, publication SR-3902, 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 90 standard.

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

<i>or_operand</i>	<i>conjunct_expr</i>
<i>and_op</i>	.AND.
<i>or_op</i>	.OR.
<i>equiv_operand</i>	<i>inclusive_disjunct_expr</i>
<i>level_5_expr</i>	<i>equivalence_expr</i>
<i>mask_expr</i>	<i>logical_expr</i>

R701	<i>primary</i>	is	<i>constant</i> or <i>constant_subobject</i> or <i>variable</i> or <i>array_constructor</i> or <i>structure_constructor</i> or <i>function_reference</i> or (<i>expr</i>)
R702	<i>constant_subobject</i>	is	<i>subobject</i>

Constraint: *subobject* must be a subobject designator whose parent is a constant.

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

R703	<i>level_1_expr</i>	is	[<i>defined_unary_op</i>] <i>primary</i>
R704	<i>defined_unary_op</i>	is	. <i>letter</i> [<i>letter</i>]

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	<i>mult_operand</i>	is	<i>level_1_expr</i> [<i>power_op mult_operand</i>]
R706	<i>add_operand</i>	is	[<i>add_operand mult_op</i>] <i>mult_operand</i>
R707	<i>level_2_expr</i>	is	[[<i>level_2_expr</i>] <i>add_op</i>] <i>add_operand</i>
R708	<i>power_op</i>	is	**

R709	<i>mult_op</i>	is	*
		or	/
R710	<i>add_op</i>	is	+
		or	-
R711	<i>level_3_expr</i>	is	[<i>level_3_expr</i> <i>concat_op</i>] <i>level_2_expr</i>
R712	<i>concat_op</i>	is	//
R713	<i>level_4_expr</i>	is	[<i>level_3_expr</i> <i>rel_op</i>] <i>level_3_expr</i>
R714	<i>rel_op</i>	is	.EQ.
		or	.NE.
		or	.LT.
		or	.LE.
		or	.GT.
		or	.GE.
EXT		or	.LG.
		or	==
		or	/=
		or	<
		or	<=
		or	>
		or	>=
EXT		or	<>
R715	<i>and_operand</i>	is	[<i>not_op</i>] <i>level_4_expr</i>
R716	<i>or_operand</i>	is	[<i>or_operand</i> <i>and_op</i>] <i>and_operand</i>
R717	<i>equiv_operand</i>	is	[<i>equiv_operand</i> <i>or_op</i>] <i>or_operand</i>
R718	<i>level_5_expr</i>	is	[<i>level_5_expr</i> <i>equiv_op</i>] <i>equiv_operand</i>
R719	<i>not_op</i>	is	.NOT.
R720	<i>and_op</i>	is	.AND.
R721	<i>or_op</i>	is	.OR.
R722	<i>equiv_op</i>	is	.EQV.
		or	.NEQV.
R723	<i>expr</i>	is	[<i>expr</i> <i>defined_binary_op</i>] <i>level_5_expr</i>
R724	<i>defined_binary_op</i>	is	. <i>letter</i> [<i>letter</i>]

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	<i>logical_expr</i>	is	<i>expr</i>
------	---------------------	-----------	-------------

Constraint: *logical_expr* must be type logical.

R726	<i>char_expr</i>	is	<i>expr</i>
------	------------------	-----------	-------------

Constraint: *char_expr* must be type character.

R727	<i>default_char_expr</i>	is	<i>expr</i>
------	--------------------------	-----------	-------------

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

R728	<i>int_expr</i>	is	<i>expr</i>
------	-----------------	-----------	-------------

Constraint: *int_expr* must be type integer.

R729	<i>numeric_expr</i>	is	<i>expr</i>
------	---------------------	-----------	-------------

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

R730	<i>initialization_expr</i>	is	<i>expr</i>
------	----------------------------	-----------	-------------

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

R731	<i>char_initialization_expr</i>	is	<i>char_expr</i>
------	---------------------------------	-----------	------------------

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

R732	<i>int_initialization_expr</i>	is	<i>int_expr</i>
------	--------------------------------	-----------	-----------------

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

R733	<i>logical_initialization_expr</i>	is	<i>logical_expr</i>
------	------------------------------------	-----------	---------------------

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

R734	<i>specification_expr</i>	is	<i>scalar_int_expr</i>
------	---------------------------	-----------	------------------------

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

R735	<i>assignment_stmt</i>	is	<i>variable = expr</i>
------	------------------------	-----------	------------------------

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

R736	<i>pointer_assignment_stmt</i>	is	<i>pointer_object => target</i>
------	--------------------------------	-----------	------------------------------------

R737	<i>target</i>	is	<i>variable</i>
		or	<i>expr</i>

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, 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	<i>where_stmt</i>	is	WHERE (<i>mask_expr</i>) <i>assignment_stmt</i>
R739	<i>where_construct</i>	is	<i>where_construct_stmt</i> [<i>assignment_stmt</i>] . . . [<i>elsewhere_stmt</i> [<i>assignment_stmt</i>] . . .] <i>end_where_stmt</i>
R740	<i>where_construct_stmt</i>	is	WHERE (<i>mask_expr</i>)
R741	<i>mask_expr</i>	is	<i>logical_expr</i>
R742	<i>elsewhere_stmt</i>	is	ELSEWHERE
R743	<i>end_where_stmt</i>	is	END WHERE

Constraint: In each *assignment_stmt*, the *mask_expr* and the variable being defined must be arrays of the same shape.

Constraint: The *assignment_stmt* must not be a defined assignment.

1.2.8 Execution control

The following syntax rules are described in section 8, "Execution control," of the Fortran 90 standard.

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

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	<i>if_stmt</i>	is	IF (<i>scalar_logical_expr</i>) <i>action_stmt</i>
------	----------------	-----------	--

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	<i>case_construct</i>	is	<i>select_case_stmt</i> [<i>case_stmt</i> <i>block</i>] . . . <i>end_select_stmt</i>
R809	<i>select_case_stmt</i>	is	[<i>case_construct_name</i>] : SELECT CASE (<i>case_expr</i>)
R810	<i>case_stmt</i>	is	CASE <i>case_selector</i> [<i>case_construct_name</i>]
R811	<i>end_select_stmt</i>	is	END SELECT [<i>case_construct_name</i>]

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	<i>case_expr</i>	is	<i>scalar_int_expr</i> or <i>scalar_char_expr</i> or <i>scalar_logical_expr</i>
R813	<i>case_selector</i>	is	(<i>case_value_range_list</i>) or DEFAULT

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

R814	<i>case_value_range</i>	is	<i>case_value</i>
		or	<i>case_value</i> :
		or	: <i>case_value</i>
		or	<i>case_value</i> : <i>case_value</i>
R815	<i>case_value</i>	is	<i>scalar_int_initialization_expr</i>
		or	<i>scalar_char_initialization_expr</i>
		or	<i>scalar_logical_initialization_expr</i>

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_ranges* must not overlap; that is, there must be no possible value of the *case_expr* that matches more than one *case_value_range*.

R816	<i>do_construct</i>	is	<i>block_do_construct</i>
		or	<i>nonblock_do_construct</i>
R817	<i>block_do_construct</i>	is	<i>do_stmt</i>
			<i>do_block</i>
			<i>end_do</i>
R818	<i>do_stmt</i>	is	<i>label_do_stmt</i>
		or	<i>nonlabel_do_stmt</i>
R819	<i>label_do_stmt</i>	is	[<i>do_construct_name</i> :] DO <i>label</i> [<i>loop_control</i>]
R820	<i>nonlabel_do_stmt</i>	is	[<i>do_construct_name</i> :] DO [<i>loop_control</i>]
R821	<i>loop_control</i>	is	[,] <i>do_variable</i> = <i>scalar_numeric_expr</i> ,
		or	<i>scalar_numeric_expr</i> [, <i>scalar_numeric_expr</i>]
			[,] WHILE (<i>scalar_logical_expr</i>)
R822	<i>do_variable</i>	is	<i>scalar_variable</i>

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

R823	<i>do_block</i>	is	<i>block</i>
R824	<i>end_do</i>	is or	<i>end_do_stmt</i> <i>continue_stmt</i>
R825	<i>end_do_stmt</i>	is	END DO [<i>do_construct_name</i>]

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.

R826	<i>nonblock_do_construct</i>	is or	<i>action_term_do_construct</i> <i>outer_shared_do_construct</i>
R827	<i>action_term_do_construct</i>	is	<i>label_do_stmt</i> <i>do_body</i> <i>do_term_action_stmt</i>
R828	<i>do_body</i>	is	[<i>execution_part_construct</i>] . . .
R829	<i>do_term_action_stmt</i>	is	<i>action_stmt</i>

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*, an *arithmetic_if_stmt*, or an *assigned_goto_stmt*.

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

R830	<i>outer_shared_do_construct</i>	is	<i>label_do_stmt</i> <i>do_body</i> <i>shared_term_do_construct</i>
R831	<i>shared_term_do_construct</i>	is or	<i>outer_shared_do_construct</i> <i>inner_shared_do_construct</i>
R832	<i>inner_shared_do_construct</i>	is	<i>label_do_stmt</i> <i>do_body</i> <i>do_term_shared_stmt</i>
R833	<i>do_term_shared_stmt</i>	is	<i>action_stmt</i>

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*, an *arithmetic_if_stmt*, or an *assigned_goto_stmt*.

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	<i>cycle_stmt</i>	is	CYCLE [<i>do_construct_name</i>]
------	-------------------	-----------	------------------------------------

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	<i>exit_stmt</i>	is	EXIT [<i>do_construct_name</i>]
------	------------------	-----------	-----------------------------------

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	<i>goto_stmt</i>	is	GO TO <i>label</i>
------	------------------	-----------	--------------------

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

R837	<i>computed_goto_stmt</i>	is	GO TO (<i>label_list</i>) [,] <i>scalar_int_expr</i>
------	---------------------------	-----------	--

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*.

R838	<i>assign_stmt</i>	is	ASSIGN <i>label</i> TO <i>scalar_int_variable</i>
------	--------------------	-----------	---

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*.

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

R839	<i>assigned_goto_stmt</i>	is	GO TO <i>scalar_int_variable</i> [[,] (<i>label_list</i>)]
------	---------------------------	-----------	--

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*.

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

R840	<i>arithmetic_if_stmt</i>	is	IF (<i>scalar_numeric_expr</i>) <i>label</i> , <i>label</i> , <i>label</i>
------	---------------------------	-----------	--

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

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

R841	<i>continue_stmt</i>	is	CONTINUE
R842	<i>stop_stmt</i>	is	STOP [<i>stop_code</i>]
R843	<i>stop_code</i>	is	<i>scalar_char_constant</i>
EXT		or	<i>digit</i> [<i>digit</i>] . . .

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

R844	<i>pause_stmt</i>	is	PAUSE [<i>stop_code</i>]
------	-------------------	-----------	----------------------------

1.2.9 Input/Output (I/O) statements

The following syntax rules are described in section 9, "Input/Output statements," of the Fortran 90 standard.

R901	<i>io_unit</i>	is	<i>external_file_unit</i>
		or	*
		or	<i>internal_file_unit</i>
EXT		or	<i>unit_name</i>
R902	<i>external_file_unit</i>	is	<i>scalar_int_expr</i>
R903	<i>internal_file_unit</i>	is	<i>char_variable</i>

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

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

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	<i>close_stmt</i>	is	CLOSE (<i>close_spec_list</i>)
R908	<i>close_spec</i>	is	[UNIT =] <i>external_file_unit</i>
		or	IOSTAT = <i>scalar_default_int_variable</i>
		or	ERR = <i>label</i>
		or	STATUS = <i>scalar_char_expr</i>

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

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.

R913	<i>format</i>	is	<i>default_char_expr</i>
		or	<i>label</i>
		or	*
		or	<i>scalar_default_int_variable</i>

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	<i>input_item</i>	is	<i>variable</i>
		or	<i>io_implied_do</i>
R915	<i>output_item</i>	is	<i>expr</i>
		or	<i>io_implied_do</i>
R916	<i>io_implied_do</i>	is	(<i>io_implied_do_object_list</i> , <i>io_implied_do_control</i>)
R917	<i>io_implied_do_object</i>	is	<i>input_item</i>
		or	<i>output_item</i>
R918	<i>io_implied_do_control</i>	is	<i>do_variable</i> = <i>scalar_numeric_expr</i> , <i>scalar_numeric_expr</i> [, <i>scalar_numeric_expr</i>]

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

Constraint: The *do_variable* must be a scalar of type integer, default real, or double-precision real.

Constraint: Each *scalar_numeric_expr* in an *io_implied_do_control* must be of type integer, default real, or double-precision real.

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	<i>buffer_in_stmt</i>	is	BUFFER IN (<i>io_unit</i> , <i>mode</i>) (<i>start_loc</i> , <i>end_loc</i>)
EXT	<i>buffer_out_stmt</i>	is	BUFFER OUT (<i>io_unit</i> , <i>mode</i>) (<i>start_loc</i> , <i>end_loc</i>)
EXT	<i>io_unit</i>	is or	<i>external_file_unit</i> <i>file_name_expr</i>
EXT	<i>mode</i>	is	<i>scalar_integer_expr</i>
EXT	<i>start_loc</i>	is	<i>variable</i>
EXT	<i>end_loc</i>	is	<i>variable</i>

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	<i>backspace_stmt</i>	is or	BACKSPACE <i>external_file_unit</i> BACKSPACE (<i>position_spec_list</i>)
R920	<i>endfile_stmt</i>	is or	ENDFILE <i>external_file_unit</i> ENDFILE (<i>position_spec_list</i>)
R921	<i>rewind_stmt</i>	is or	REWIND <i>external_file_unit</i> REWIND (<i>position_spec_list</i>)
R922	<i>position_spec</i>	is or or	[UNIT =] <i>scalar_int_expr</i> IOSTAT = <i>scalar_default_int_variable</i> ERR = <i>label</i>

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	<i>inquire_stmt</i>	is	INQUIRE (<i>inquire_spec_list</i>)
		or	INQUIRE (IOLENGTH = <i>scalar_default_int_variable</i>) <i>output_item_list</i>
R924	<i>inquire_spec</i>	is	[UNIT =] <i>external_file_unit</i>
		or	FILE = <i>file_name_expr</i>
		or	IOSTAT = <i>scalar_default_int_variable</i>
		or	ERR = <i>label</i>
		or	EXIST = <i>scalar_default_logical_variable</i>
		or	OPENED = <i>scalar_default_logical_variable</i>
		or	NUMBER = <i>scalar_default_int_variable</i>
		or	NAMED = <i>scalar_default_logical_variable</i>
		or	NAME = <i>scalar_char_variable</i>
		or	ACCESS = <i>scalar_char_variable</i>
		or	SEQUENTIAL = <i>scalar_char_variable</i>
		or	DIRECT = <i>scalar_char_variable</i>
		or	FORM = <i>scalar_char_variable</i>
		or	FORMATTED = <i>scalar_char_variable</i>
		or	UNFORMATTED = <i>scalar_char_variable</i>
		or	RECL = <i>scalar_default_int_variable</i>
		or	NEXTREC = <i>scalar_default_int_variable</i>
		or	BLANK = <i>scalar_char_variable</i>
		or	POSITION = <i>scalar_char_variable</i>
		or	ACTION = <i>scalar_char_variable</i>
		or	READ = <i>scalar_char_variable</i>
		or	WRITE = <i>scalar_char_variable</i>
		or	READWRITE = <i>scalar_char_variable</i>
		or	DELIM = <i>scalar_char_variable</i>
		or	PAD = <i>scalar_char_variable</i>

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 90 standard.

R1001	<i>format_stmt</i>	is	FORMAT <i>format_specification</i>
R1002	<i>format_specification</i>	is	([<i>format_item_list</i>])

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	<i>format_item</i>	is	[<i>r</i>] <i>data_edit_desc</i>
		or	<i>control_edit_desc</i>
		or	<i>char_string_edit_desc</i>
		or	[<i>r</i>] (<i>format_item_list</i>)
R1004	<i>r</i>	is	<i>int_literal_constant</i>

Constraint: *r* must be positive.

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

R1005	<i>data_edit_desc</i>	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
R1006	<i>w</i>	is	<i>int_literal_constant</i>
R1007	<i>m</i>	is	<i>int_literal_constant</i>
R1008	<i>d</i>	is	<i>int_literal_constant</i>
R1009	<i>e</i>	is	<i>int_literal_constant</i>

Constraint: *w* and *e* must be positive.

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

R1010	<i>control_edit_desc</i>	is	<i>position_edit_desc</i>
		or	[<i>r</i>] /
		or	:
		or	<i>sign_edit_desc</i>
		or	<i>k</i> P
		or	<i>blank_interp_edit_desc</i>
R1011	<i>k</i>	is	<i>signed_int_literal_constant</i>

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

R1012	<i>position_edit_desc</i>	is	T <i>n</i>
		or	TL <i>n</i>
		or	TR <i>n</i>
		or	<i>n</i> X
EXT		or	\$
R1013	<i>n</i>	is	<i>int_literal_constant</i>

Constraint: *n* must be positive.

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

R1014	<i>sign_edit_desc</i>	is	S
		or	SP
		or	SS
R1015	<i>blank_interp_edit_desc</i>	is	BN
		or	BZ
R1016	<i>char_string_edit_desc</i>	is	<i>char_literal_constant</i>
		or	<i>c</i> H <i>rep_char</i> [<i>rep_char</i>] ...
R1017	<i>c</i>	is	<i>int_literal_constant</i>

Constraint: *c* must be positive.

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

Constraint: The *rep_char* in the *c* H form must be of default character type.

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 90 standard.

R1101	<i>main_program</i>	is	[<i>program_stmt</i>] [<i>specification_part</i>] [<i>execution_part</i>] [<i>internal_subprogram_part</i>] <i>end_program_stmt</i>
R1102	<i>program_stmt</i>	is	PROGRAM <i>program_name</i> [(<i>args</i>)]
EXT	<i>args</i>	is	Any character in the CF90 character set. The CF90 compiler ignores any <i>args</i> specified after <i>program_name</i> .
R1103	<i>end_program_stmt</i>	is	END [PROGRAM [<i>program_name</i>]]

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* (R204) of a main program.

R1104	<i>module</i>	is	<i>module_stmt</i> [<i>specification_part</i>] [<i>module_subprogram_part</i>] <i>end_module_stmt</i>
R1105	<i>module_stmt</i>	is	MODULE <i>module_name</i>
R1106	<i>end_module_stmt</i>	is	END [MODULE [<i>module_name</i>]]

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* (R204) of a module.

R1107	<i>use_stmt</i>	is	USE <i>module_name</i> [, <i>rename_list</i>]
		or	USE <i>module_name</i> , ONLY : [<i>only_list</i>]
R1108	<i>rename</i>	is	<i>local_name</i> => <i>use_name</i>
R1109	<i>only</i>	is	<i>access_id</i>
		or	[<i>local_name</i> =>] <i>use_name</i>

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

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

R1110	<i>block_data</i>	is	<i>block_data_stmt</i> [<i>specification_part</i>] <i>end_block_data_stmt</i>
R1111	<i>block_data_stmt</i>	is	BLOCK DATA [<i>block_data_name</i>]
R1112	<i>end_block_data_stmt</i>	is	END [BLOCK DATA [<i>block_data_name</i>]]

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 90 standard.

R1201	<i>interface_block</i>	is	<i>interface_stmt</i> [<i>interface_body</i>] . . . [<i>module_procedure_stmt</i>] . . . <i>end_interface_stmt</i>
R1202	<i>interface_stmt</i>	is	INTERFACE [<i>generic_spec</i>]
R1203	<i>end_interface_stmt</i>	is	END INTERFACE
R1204	<i>interface_body</i>	is	<i>function_stmt</i> [<i>specification_part</i>] <i>end_function_stmt</i> or <i>subroutine_stmt</i> [<i>specification_part</i>] <i>end_subroutine_stmt</i>
R1205	<i>module_procedure_stmt</i>	is	MODULE PROCEDURE <i>procedure_name_list</i>
R1206	<i>generic_spec</i>	is	<i>generic_name</i> or OPERATOR (<i>defined_operator</i>) or ASSIGNMENT (=)

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

Constraint: The MODULE PROCEDURE specification is allowed only if the *interface_block* has a *generic_spec* and has a host that is a module or accesses a module by use association; each *procedure_name* must be the name of a module procedure that is accessible in the host.

Constraint: An *interface_block* must not appear in a BLOCK DATA program unit.

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.

R1207	<i>external_stmt</i>	is	EXTERNAL <i>external_name_list</i>
R1208	<i>intrinsic_stmt</i>	is	INTRINSIC <i>intrinsic_procedure_name_list</i>

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

R1209	<i>function_reference</i>	is	<i>function_name</i> ([<i>actual_arg_spec_list</i>])
-------	---------------------------	-----------	--

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

R1210	<i>call_stmt</i>	is	CALL <i>subroutine_name</i> [([<i>actual_arg_spec_list</i>])]
R1211	<i>actual_arg_spec</i>	is	[<i>keyword</i> =] <i>actual_arg</i>
R1212	<i>keyword</i>	is	<i>dummy_arg_name</i>
R1213	<i>actual_arg</i>	is	<i>expr</i> or <i>variable</i> or <i>procedure_name</i>
R1214	<i>alt_return_spec</i>	is	* <i>label</i>

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 (see subsections 12.3.2.1 and 13.1 of the Fortran 90 standard).

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*.

R1215	<i>function_subprogram</i>	is	<i>function_stmt</i> [<i>specification_part</i>] [<i>execution_part</i>] [<i>internal_subprogram_part</i>] <i>end_function_stmt</i>
R1216	<i>function_stmt</i>	is	[<i>prefix</i>] FUNCTION <i>function_name</i> ([<i>dummy_arg_name_list</i>]) [RESULT (<i>result_name</i>)]

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

R1217	<i>prefix</i>	is	<i>prefix_spec</i> [<i>prefix_spec</i>] . . .
EXT	<i>prefix_spec</i>	is	<i>type_spec</i>
		or	RECURSIVE
		or	PURE
		or	ELEMENTAL
R1218	<i>end_function_stmt</i>	is	END [FUNCTION [<i>function_name</i>]]

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 must not contain an ENTRY statement.

Constraint: An internal function 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*.

R1219	<i>subroutine_subprogram</i>	is	<i>subroutine_stmt</i> [<i>specification_part</i>] [<i>execution_part</i>] [<i>internal_subprogram_part</i>] <i>end_subroutine_stmt</i>
R1220	<i>subroutine_stmt</i>	is	[<i>prefix</i>] SUBROUTINE <i>subroutine_name</i> [([<i>dummy_arg_list</i>])]
EXT	<i>prefix</i>	is	<i>prefix_spec</i> [<i>prefix_spec</i>] . . .
EXT	<i>prefix_spec</i>	is	RECURSIVE or PURE or ELEMENTAL
R1221	<i>dummy_arg</i>	is	<i>dummy_arg_name</i> or *
R1222	<i>end_subroutine_stmt</i>	is	END [SUBROUTINE [<i>subroutine_name</i>]]

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*.

R1223	<i>entry_stmt</i>	is	ENTRY <i>entry_name</i> [([<i>dummy_arg_list</i>]) [RESULT (<i>result_name</i>)]]
-------	-------------------	-----------	--

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*.

Constraint: 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* may be an alternate return indicator only if the ENTRY statement is contained in a subroutine subprogram.

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

R1224	<i>return_stmt</i>	is	RETURN [<i>scalar_int_expr</i>]
-------	--------------------	-----------	-----------------------------------

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

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

R1225	<i>contains_stmt</i>	is	CONTAINS
R1226	<i>stmt_function_stmt</i>	is	<i>function_name</i> ([<i>dummy_arg_name_list</i>]) = <i>scalar_expr</i>

Constraint: The *scalar_expr* may be composed only of constants (literal and named), references to scalar variables and array elements, references to functions and function dummy procedures, and intrinsic operators. 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.

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 have been declared as an array earlier in the scoping unit or made accessible by use or host association. If a scalar variable, array element, 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.

1.2.13 Intrinsic procedures

There are no syntax rules described in section 13, "Intrinsic procedures," of the Fortran 90 standard.

1.2.14 Scope, association, and definition

There are no syntax rules described in section 14, "Scope, association, and definition," of the Fortran 90 standard.

1.3 Cross-references

Table 2 provides a cross-reference of all Fortran 90 standard syntactic symbols used in the BNF, showing the rule in which they are defined and all rules in which they are referenced.

The symbols are sorted alphabetically within three categories: nonterminal symbols that are defined, nonterminal symbols that are not defined in the BNF (but are defined by other means), and terminal symbols. Except for those ending with *_name*, the only undefined nonterminal symbols are *letter*, *digit*, *special_character*, and *rep_char*; these nonterminal symbols are defined in the *Fortran Language Reference Manual, Volume 1*, publication SR-3902. Symbols ending with *_name* are defined by the following rule:

xyz_name **is** a name

All occurrences of *_list* and *scalar_* in the symbol names have been removed.

Table 2. Fortran 90 standard nonterminal symbols defined through BNF rules

Symbol	Definition	References
<i>ac_do_variable</i>	R435	R434
<i>ac_implied_do</i>	R433	R432
<i>ac_implied_do_control</i>	R434	R433
<i>ac_value</i>	R432	R431, R433
<i>access_id</i>	R522	R521, R1109
<i>access_spec</i>	R510	R424, R503, R521
<i>access_stmt</i>	R521	R214
<i>action_stmt</i>	R216	R215, R807, R829, R833
<i>action_term_do_construct</i>	R827	R826
<i>actual_arg</i>	R1213	R1211
<i>actual_arg_spec</i>	R1211	R1209, R1210
<i>add_op</i>	R710	R310, R707
<i>add_operand</i>	R706	R706, R707
<i>allocatable_stmt</i>	R526	R214
<i>allocate_lower_bound</i>	R627	R626
<i>allocate_object</i>	R625	R624, R631
<i>allocate_shape_spec</i>	R626	R624
<i>allocate_stmt</i>	R622	R216
<i>allocate_upper_bound</i>	R628	R626
<i>allocation</i>	R624	R622
<i>alphanumeric_character</i>	R302	R301, R304
<i>alt_return_spec</i>	R1214	R1213
<i>and_op</i>	R720	R310, R716
<i>and_operand</i>	R715	R716
<i>arithmetic_if_stmt</i>	R840	R216
<i>array_constructor</i>	R431	R701

Symbol	Definition	References
<i>array_element</i>	R615	R536, R547, R602, R610
<i>array_section</i>	R616	R602
<i>array_spec</i>	R512	R503, R504, R525, R528
<i>assign_stmt</i>	R838	R216
<i>assigned_goto_stmt</i>	R839	R216
<i>assignment_stmt</i>	R735	R216, R738, R739
<i>assumed_shape_spec</i>	R516	R512
<i>assumed_size_spec</i>	R518	R512
<i>attr_spec</i>	R503	R501
<i>backspace_stmt</i>	R919	R216
<i>binary_constant</i>	R408	R407
<i>blank_interp_edit_desc</i>	R1015	R1010
<i>block</i>	R801	R802, R808, R823
<i>block_data</i>	R1110	R202
<i>block_data_stmt</i>	R1111	R1110
<i>block_do_construct</i>	R817	R816
<i>boz_literal_constant</i>	R407	R306, R533
<i>c</i>	R1017	R1016
<i>call_stmt</i>	R1210	R216
<i>case_construct</i>	R808	R215
<i>case_expr</i>	R812	R809
<i>case_selector</i>	R813	R810
<i>case_stmt</i>	R810	R808
<i>case_value</i>	R815	R814
<i>case_value_range</i>	R814	R813
<i>char_constant</i>	R309	R843
<i>char_expr</i>	R726	R731, R812
<i>char_initialization_expr</i>	R731	R815

Symbol	Definition	References
<i>char_length</i>	R508	R429, R504, R507
<i>char_literal_constant</i>	R420	R306, R1016
<i>char_selector</i>	R506	R502
<i>char_string_edit_desc</i>	R1016	R1003
<i>char_variable</i>	R605	R216
<i>character</i>	R301	
<i>close_spec</i>	R908	R907
<i>close_stmt</i>	R907	R216
<i>common_block_object</i>	R549	R548
<i>common_stmt</i>	R548	R214
<i>complex_literal_constant</i>	R417	R306
<i>component_array_spec</i>	R428	R427, R429
<i>component_attr_spec</i>	R427	R426
<i>component_decl</i>	R429	R426
<i>component_def_stmt</i>	R426	R422
<i>computed_goto_stmt</i>	R837	
<i>concat_op</i>	R712	R310, R711
<i>connect_spec</i>	R905	R904
<i>constant</i>	R305	R308, R309, R533, R610, R701
<i>constant_subobject</i>	R702	R701
<i>contains_stmt</i>	R1225	R210, R212
<i>continue_stmt</i>	R841	R216, R824
<i>control_edit_desc</i>	R1010	R1003
<i>cycle_stmt</i>	R834	R216
<i>d</i>	R1008	R1005
<i>data_edit_desc</i>	R1005	R1003
<i>data_i_do_object</i>	R536	R535
<i>data_i_do_variable</i>	R537	R535

Symbol	Definition	References
<i>data_implied_do</i>	R535	R531, R536
<i>data_ref</i>	R612	R614, R615, R616
<i>data_stmt</i>	R529	R209, R214
<i>data_stmt_constant</i>	R533	R532
<i>data_stmt_object</i>	R531	R530
<i>data_stmt_repeat</i>	R534	R532
<i>data_stmt_set</i>	R530	R529
<i>data_stmt_value</i>	R532	R530
<i>deallocate_stmt</i>	R631	R216
<i>declaration_construct</i>	R207	R204
<i>default_char_expr</i>	R727	R905, R906, R908, R912, R913
<i>default_char_variable</i>	R606	R903, R924
<i>default_int_variable</i>	R608	R905, R908, R912, R913, R922, R923, R924
<i>default_logical_variable</i>	R604	R924
<i>deferred_shape_spec</i>	R517	R428, R512, R526, R527
<i>defined_binary_op</i>	R724	R311, R723
<i>defined_operator</i>	R311	R1206
<i>defined_unary_op</i>	R704	R311, R703
<i>derived_type_def</i>	R422	R207
<i>derived_type_stmt</i>	R424	R422
<i>digit_string</i>	R402	R401, R404, R405, R413, R414
<i>dimension_stmt</i>	R525	R214
<i>do_block</i>	R823	R817
<i>do_body</i>	R828	R827, R830, R832
<i>do_construct</i>	R816	R215
<i>do_stmt</i>	R818	R817
<i>do_term_action_stmt</i>	R829	R827
<i>do_term_shared_stmt</i>	R833	R832

Symbol	Definition	References
<i>do_variable</i>	R822	R821, R918
<i>dummy_arg</i>	R1221	R1220, R1223
<i>e</i>	R1009	R1005
<i>else_if_stmt</i>	R804	R802
<i>else_stmt</i>	R805	R802
<i>elsewhere_stmt</i>	R742	R739
<i>end_block_data_stmt</i>	R1112	R1110
<i>end_do</i>	R824	R817
<i>end_do_stmt</i>	R825	R824
<i>end_function_stmt</i>	R1218	R216, R1204, R1215
<i>end_if_stmt</i>	R806	R802
<i>end_interface_stmt</i>	R1203	R1201
<i>end_module_stmt</i>	R1106	R1104
<i>end_program_stmt</i>	R1103	R216, R1101
<i>end_select_stmt</i>	R811	R808
<i>end_subroutine_stmt</i>	R1222	R216, R1204, R1219
<i>end_type_stmt</i>	R425	R422
<i>end_where_stmt</i>	R743	R739
<i>endfile_stmt</i>	R920	R216
<i>entity_decl</i>	R504	R501
<i>entry_stmt</i>	R1223	R206, R207, R209
<i>equiv_op</i>	R722	R310, R718
<i>equiv_operand</i>	R717	R717, R718
<i>equivalence_object</i>	R547	R546
<i>equivalence_set</i>	R546	R545
<i>equivalence_stmt</i>	R545	R214
<i>executable_construct</i>	R215	R208, R209
<i>executable_program</i>	R201	

Symbol	Definition	References
<i>execution_part</i>	R208	R1101, R1215, R1219
<i>execution_part_construct</i>	R209	R208, R801, R828
<i>exit_stmt</i>	R835	R216
<i>explicit_shape_spec</i>	R513	R428, R512, R518, R549
<i>exponent</i>	R416	R413
<i>exponent_letter</i>	R415	R413
<i>expr</i>	R723	R430, R432, R701, R723, R725, R726, R727, R728, R729, R730, R735, R737, R915, R1213, R1226
<i>extended_intrinsic_op</i>	R312	R311
<i>external_file_unit</i>	R902	R901, R905, R908, R919, R920, R921, R922, R924
<i>external_stmt</i>	R1207	R214
<i>external_subprogram</i>	R203	R202
<i>file_name_expr</i>	R906	R905, R924
<i>format</i>	R913	R909, R911, R912
<i>format_item</i>	R1003	R1002, R1003
<i>format_specification</i>	R1002	R1001
<i>format_stmt</i>	R1001	R206, R207, R209
<i>function_reference</i>	R1209	R701
<i>function_stmt</i>	R1216	R1204, R1215
<i>function_subprogram</i>	R1215	R203, R211, R213
<i>generic_spec</i>	R1206	R522, R1202
<i>goto_stmt</i>	R836	R216
<i>hex_constant</i>	R410	R407
<i>hex_digit</i>	R411	R410
<i>if_construct</i>	R802	R215
<i>if_stmt</i>	R807	R216
<i>if_then_stmt</i>	R803	R802
<i>imag_part</i>	R419	R417

Symbol	Definition	References
<i>implicit_part</i>	R205	R204
<i>implicit_part_stmt</i>	R206	R205
<i>implicit_spec</i>	R541	R540
<i>implicit_stmt</i>	R540	R205, R206
<i>initialization_expr</i>	R730	R504, R539
<i>inner_shared_do_construct</i>	R832	R831
<i>input_item</i>	R914	R909, R917
<i>inquire_spec</i>	R924	R923
<i>inquire_stmt</i>	R923	R216
<i>int_constant</i>	R308	R534
<i>int_expr</i>	R728	R434, R535, R611, R617, R620, R621, R627, R628, R732, R734, R812, R837, R902, R905, R912, R1224
<i>int_initialization_expr</i>	R732	R505, R506, R815
<i>int_literal_constant</i>	R404	R306, R403, R508, R1004, R1006, R1007, R1008, R1009, R1013, R1017
<i>int_variable</i>	R607	R435, R537, R623, R838, R839
<i>intent_spec</i>	R511	R503, R519
<i>intent_stmt</i>	R519	R214
<i>interface_block</i>	R1201	R207
<i>interface_body</i>	R1204	R1201
<i>interface_stmt</i>	R1202	R1201
<i>internal_file_unit</i>	R903	R901
<i>internal_subprogram</i>	R211	R210
<i>internal_subprogram_part</i>	R210	R1101, R1215, R1219
<i>intrinsic_operator</i>	R310	R312
<i>intrinsic_stmt</i>	R1208	R214
<i>io_control_spec</i>	R912	R909, R910
<i>io_implied_do</i>	R916	R914, R915

Symbol	Definition	References
<i>io_implied_do_control</i>	R918	R916
<i>io_implied_do_object</i>	R917	R916
<i>io_unit</i>	R901	R912
<i>k</i>	R1011	R1010
<i>keyword</i>	R1212	R1211
<i>kind_param</i>	R405	R404, R413, R420, R421
<i>kind_selector</i>	R505	R502
<i>label</i>	R313	R819, R836, R837, R838, R839, R840, R905, R908, R912, R913, R922, R924, R1214
<i>label_do_stmt</i>	R819	R818, R827, R830, R832
<i>length_selector</i>	R507	R506
<i>letter_spec</i>	R542	R541
<i>defined_unary_expr</i>	R703	R705
<i>summation_expr</i>	R707	R707, R711
<i>concatenation_expr</i>	R711	R711, R713
<i>comparison_expr</i>	R713	R715
<i>equivalence_expr</i>	R718	R718, R723
<i>literal_constant</i>	R306	R305
<i>logical_expr</i>	R725	R733, R741, R803, R804, R807, R812, R821
<i>logical_initialization_expr</i>	R733	R815
<i>logical_literal_constant</i>	R421	R306
<i>logical_variable</i>	R603	
<i>loop_control</i>	R821	R819, R820
<i>lower_bound</i>	R514	R513, R516, R518
<i>m</i>	R1007	R1005
<i>main_program</i>	R1101	R202
<i>mask_expr</i>	R741	R738, R740
<i>module</i>	R1104	R202

Symbol	Definition	References
<i>module_procedure_stmt</i>	R1205	R1201
<i>module_stmt</i>	R1105	R1104
<i>module_subprogram</i>	R213	R212
<i>module_subprogram_part</i>	R212	R1104
<i>mult_op</i>	R709	R310, R706
<i>mult_operand</i>	R705	R705, R706
<i>n</i>	R1013	R1012
<i>name</i>	R304	R307
<i>named_constant</i>	R307	R305, R539
<i>named_constant_def</i>	R539	R538
<i>namelist_group_object</i>	R544	R543
<i>namelist_stmt</i>	R543	R214
<i>nonblock_do_construct</i>	R826	R816
<i>nonlabel_do_stmt</i>	R820	R818
<i>not_op</i>	R719	R310, R715
<i>nullify_stmt</i>	R629	R216
<i>numeric_expr</i>	R729	R821, R840, R918
<i>octal_constant</i>	R409	R407
<i>only</i>	R1109	R1107
<i>open_stmt</i>	R904	R216
<i>optional_stmt</i>	R520	R214
<i>or_op</i>	R721	R310, R717
<i>or_operand</i>	R716	R716, R717
<i>outer_shared_do_construct</i>	R830	R826, R831
<i>output_item</i>	R915	R910, R911, R917, R923
<i>parameter_stmt</i>	R538	R206, R207
<i>parent_string</i>	R610	R609
<i>part_ref</i>	R613	R612

Symbol	Definition	References
<i>pause_stmt</i>	R844	R216
<i>pointer_assignment_stmt</i>	R736	R216
<i>pointer_object</i>	R630	R629, R736
<i>pointer_stmt</i>	R527	R214
<i>position_edit_desc</i>	R1012	R1010
<i>position_spec</i>	R922	R919, R920, R921
<i>power_op</i>	R708	R310, R705
<i>prefix</i>	R1217	R1216
<i>primary</i>	R701	R703
<i>print_stmt</i>	R911	R216
<i>private_sequence_stmt</i>	R423	R422
<i>program_stmt</i>	R1102	R1101
<i>program_unit</i>	R202	R201
<i>r</i>	R1004	R1003, R1010
<i>read_stmt</i>	R909	R216
<i>real_literal_constant</i>	R413	R306, R412
<i>real_part</i>	R418	R417
<i>rel_op</i>	R714	R310, R713
<i>rename</i>	R1108	R1107
<i>return_stmt</i>	R1224	R216
<i>rewind_stmt</i>	R921	R216
<i>save_stmt</i>	R523	R214
<i>saved_entity</i>	R524	R523
<i>section_subscript</i>	R618	R613
<i>select_case_stmt</i>	R809	R808
<i>shared_term_do_construct</i>	R831	R830
<i>sign</i>	R406	R401, R403, R412
<i>sign_edit_desc</i>	R1014	R1010

Symbol	Definition	References
<i>signed_digit_string</i>	R401	R416
<i>signed_int_literal_constant</i>	R403	R418, R419, R533, R1011
<i>signed_real_literal_constant</i>	R412	R418, R419, R533
<i>significand</i>	R414	R413
<i>specification_expr</i>	R734	R509, R514, R515
<i>specification_part</i>	R204	R1101, R1104, R1110, R1204, R1215, R1219
<i>specification_stmt</i>	R214	R207
<i>stat_variable</i>	R623	R622, R631
<i>stmt_function_stmt</i>	R1226	R207
<i>stop_code</i>	R843	R842, R844
<i>stop_stmt</i>	R842	R216
<i>stride</i>	R620	R619
<i>structure_component</i>	R614	R536, R602, R610, R625, R630
<i>structure_constructor</i>	R430	R533, R701, R430, R533, R701
<i>subobject</i>	R602	R601, R702
<i>subroutine_stmt</i>	R1220	R1204, R1219
<i>subroutine_subprogram</i>	R1219	R203, R211, R213
<i>subscript</i>	R617	R618, R619
<i>subscript_triplet</i>	R619	R618
<i>substring</i>	R609	R547, R602
<i>substring_range</i>	R611	R609, R616
<i>target</i>	R737	R736
<i>target_stmt</i>	R528	R214
<i>type_declaration_stmt</i>	R501	R207
<i>type_param_value</i>	R509	R506, R507, R508
<i>type_spec</i>	R502	R426, R501, R541, R1217
<i>underscore</i>	R303	R302
<i>upper_bound</i>	R515	R513

Symbol	Definition	References
<i>use_stmt</i>	R1107	R204
<i>variable</i>	R601	R531, R603, R604, R605, R606, R607, R608, R701, R735, R737, R822, R914, R1213
<i>vector_subscript</i>	R621	R618
<i>w</i>	R1006	R1005
<i>where_construct</i>	R739	R215
<i>where_construct_stmt</i>	R740	R739
<i>where_stmt</i>	R738	R216
<i>write_stmt</i>	R910	R216

Table 3 shows terms that are referenced in BNF definitions.

Table 3. Fortran 90 standard nonterminal symbols with no BNF definition

Symbol	References
<i>array_name</i>	R525, R526
<i>array_variable_name</i>	R601
<i>block_data_name</i>	R1111, R1112
<i>case_construct_name</i>	R809, R810, R811
<i>common_block_name</i>	R524, R548
<i>component_name</i>	R429
<i>digit</i>	R302, R313, R402, R408, R409, R411, R843
<i>do_construct_name</i>	R819, R820, R825, R834, R835
<i>dummy_arg_name</i>	R519, R520, R1212, R1216, R1221, R1226
<i>entry_name</i>	R1223
<i>external_name</i>	R1207
<i>function_name</i>	R504, R1209, R1216, R1218, R1226
<i>generic_name</i>	R1206
<i>if_construct_name</i>	R803, R804, R805, R806

Symbol	References
<i>int_constant_name</i>	R405
<i>intrinsic_procedure_name</i>	R1208
<i>letter</i>	R302, R304, R542, R704, R724
<i>local_name</i>	R1108, R1109
<i>module_name</i>	R1105, R1106, R1107
<i>namelist_group_name</i>	R543, R912
<i>object_name</i>	R504, R524, R527, R528
<i>part_name</i>	R613
<i>procedure_name</i>	R1205, R1213
<i>program_name</i>	R1102, R1103
<i>rep_char</i>	R420, R1016
<i>result_name</i>	R1216, R1223
<i>special_character</i>	R301
<i>subroutine_name</i>	R1210, R1220, R1222
<i>type_name</i>	R424, R425, R430, R502
<i>use_name</i>	R522, R1108, R1109
<i>variable_name</i>	R544, R547, R549, R601, R610, R625, R630

Table 4 shows symbols that are referenced in BNF definitions.

Table 4. Fortran 90 standard terminal symbols

Symbol	References
%	R612
*	R429, R504, R507, R509, R518, R532, R709, R901, R913, R1214, R1221
**	R708
+	R406, R710
- (hyphen)	R406, R542, R710

Symbol	References
. (period)	R414, R704, R724, R1005
.AND.	R720
.EQ.	R714
.EQV.	R722
.FALSE.	R421
.GE.	R714
.GT.	R714
.LE.	R714
.LT.	R714
.NE.	R714
.NEQV.	R722
.NOT.	R719
.OR.	R721
.TRUE.	R421
/	R524, R530, R543, R548, R709, R1010
//	R712
/=	R714
<	R714
<=	R714
=	R434, R504, R505, R506, R507, R535, R539, R622, R631, R735, R821, R905, R908, R912, R918, R922, R923, R924, R1206, R1211, R1226
==	R714
=>	R736, R1108, R1109
>	R714
>=	R714
A	R1005
ACCESS	R905, R924
ACTION	R905, R924

Symbol	References
ADVANCE	R912
ALLOCATABLE	R503, R526
ALLOCATE	R622
ASSIGN	R838
ASSIGNMENT	R1206
AUTOMATIC	R503
B	R408, R1005
BACKSPACE	R919
BLANK	R905, R924
BLOCK	R1111, R1112
BN	R1015
BZ	R1015
CALL	R1210
CASE	R809, R810
CHARACTER	R502
CLOSE	R907
COMMON	R548
COMPLEX	R502
CONTAINS	R1225
CONTINUE	R841
CYCLE	R834
D	R415, R1005
DATA	R529, R1111, R1112
DEALLOCATE	R631
DEFAULT	R813
DELIM	R905, R924
DIMENSION	R427, R503, R525
DIRECT	R924

Symbol	References
DO	R819, R820, R825
DOUBLE	R502
E	R415, R1005
ELSE	R804, R805
ELSEWHERE	R742
EN	R1005
END	R425, R743, R806, R811, R825, R912, R1103, R1106, R1112, R1203, R1218, R1222
ENDFILE	R920
ENTRY	R1223
EOR	R912
EQUIVALENCE	R545
ERR	R905, R908, R912, R922, R924
ES	R1005
EXIST	R924
EXIT	R835
EXTERNAL	R503, R1207
F	R1005
FILE	R905, R924
FMT	R912
FORM	R905, R924
FORMAT	R1001
FORMATTED	R924
FUNCTION	R1216, R1218
G	R1005
GO	R836, R837, R839
H	R1016
I	R1005
IF	R803, R804, R806, R807, R840

Symbol	References
IMPLICIT	R540
IN	R511
INOUT	R511
INQUIRE	R923
INTEGER	R502
INTENT	R503, R519
INTERFACE	R1202, R1203
INTRINSIC	R503, R1208
IOLength	R923
IOSTAT	R905, R908, R912, R922, R924
KIND	R505, R506
L	R1005
LEN	R506, R507
LOGICAL	R502
MODULE	R1105, R1106, R1205
NAME	R924
NAMED	R924
NAMELIST	R543
NEXTREC	R924
NML	R912
NONE	R540
NULLIFY	R629
NUMBER	R924
O	R409, R1005
ONLY	R1107
OPEN	R904
OPENED	R924
OPERATOR	R1206

Symbol	References
OPTIONAL	R503, R520
OUT	R511
P	R1010
PAD	R905, R924
PARAMETER	R503, R538
PAUSE	R844
POINTER	R427, R503, R527
POSITION	R905, R924
PRECISION	R502
PRINT	R911
PRIVATE	R423, R510
PROCEDURE	R1205
PROGRAM	R1102, R1103
PUBLIC	R510
READ	R909, R924
READWRITE	R924
REAL	R502
REC	R912
RECL	R905, R924
RECURSIVE	R1217, R1220
RESULT	R1216, R1223
RETURN	R1224
REWIND	R921
S	R1014
SAVE	R503, R523
SELECT	R809, R811
SEQUENCE	R423
SEQUENTIAL	R924

Symbol	References
SIZE	R912
SP	R1014
SS	R1014
STAT	R622, R631
STATUS	R905, R908
STOP	R842
SUBROUTINE	R1220, R1222
T	R1012
TARGET	R503, R528
THEN	R803, R804
TL	R1012
TO	R836, R837, R838, R839
TR	R1012
TYPE	R424, R425, R502
UNFORMATTED	R924
UNIT	R905, R908, R912, R922, R924
USE	R1107
WHERE	R738, R740, R743
WHILE	R821
WRITE	R910, R924
X	R1012
Z	R410, R1005

Decremental Features [2]

This chapter describes deleted and obsolescent Fortran features.

2.1 Deleted features

The deleted features are those features of FORTRAN 77 that are considered by the Fortran 90 standard to be redundant and considered largely unused. The list of deleted features for Fortran 90 is empty; there are none.

2.2 Obsolescent features

The obsolescent features are those features of FORTRAN 77 that are considered by the Fortran 90 standard to be redundant. The Fortran 90 standard states that these features are obsolescent and provides preferred alternatives.

The obsolescent features and the preferred alternatives are as follows:

<u>Obsolescent feature</u>	<u>Preferred alternative</u>
Arithmetic IF	IF statement or IF construct
Real and double precision type DO control variables and DO loop control expressions	Integer type
Shared DO termination and termination on a statement other than END DO or CONTINUE statements	An END DO or a CONTINUE statement for each DO statement
Branching to an END IF statement from outside its IF construct	Branch to the statement following the END IF
Alternate return	See topic discussed in Section 2.2.1, page 80
PAUSE statement	See topic discussed in Section 2.2.2, page 80
ASSIGN and assigned GO TO statements	See topic discussed in Section 2.2.3, page 80
Assigned FORMAT specifiers	See topic discussed in Section 2.2.4, page 81

H edit descriptor

See topic discussed in Section 2.2.5, page 81

2.2.1 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 same effect can be achieved with a return code that is used in a computed `GO TO` statement or `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
```

2.2.2 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.

2.2.3 ASSIGN and assigned GO TO statements

The `ASSIGN` statement allows a label to be dynamically assigned to an integer variable, and the assigned `GO TO` statement allows indirect branching through this variable. This hinders the readability of the program flow, especially if the integer variable also is used in arithmetic operations. The two totally different usages of the integer variable can be an obscure source of error.

These statements have been commonly used to simulate internal procedures, which now can be coded directly.

2.2.4 Assigned `FORMAT` specifiers

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.

2.2.5 `H` editing

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.

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 5. 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 Cray Research 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 between the FORTRAN 77 and Fortran 90 languages. It also describes the differences between the Cray Research CF77 compiling system, the Cray Research CF90 compiler, and the run-time libraries that support them. Many extensions that exist in the CF77 compiling system are retained in the CF90 compiler.

Note: The Cray Research CF77 compiling system is in maintenance mode. Information pertaining to that compiler is included in this chapter for transitional purposes.

The terms *FORTRAN 77* and *Fortran 90* are used to designate the ANSI and ISO standards for Fortran. The terms *CF77 compiling system* and *CF90 compiler* apply to the Cray Research products that implement the standards.

This chapter is divided into the following sections:

- FORTRAN 77 and Fortran 90 differences
- Fortran 90 standard differences and incompatibilities with the Cray Research extensions to the CF77 compiling system
- CF90 restrictions on CF77 input/output (I/O) extensions
- CF90 and CF77 integrated environment differences
- CF90 extensions to Fortran 90
- CF90 and CF77 implementation differences

Note: The information in this chapter that describes the CF90 and CF77 compilers pertains only to programs being run on a UNICOS or UNICOS/mk system. Information describing differences that pertain to IRIX systems and the MIPSpro 7 Fortran 90 compiler is under development and will appear in the *MIPSpro 7 Fortran 90 Commands and Directives Reference Manual*, publication SR-3907.

4.1 FORTRAN 77 and Fortran 90 differences

The following sections describe the main areas of differences between Fortran 90 and FORTRAN 77.

4.1.1 Fortran 90 and end-of-record action

FORTRAN 77 requires that the number of characters required by the input list be less than or equal to the number of characters in the record during formatted input. Fortran 90 specifies that unless the `PAD=NO` option is specified in an appropriate `OPEN` statement, the input record is logically padded with blanks if there are not enough characters in the record.

FORTRAN 77 effectively defaulted to `PAD=NO`. Fortran 90 defaults to `PAD=YES`.

4.1.2 Fortran 90 and new intrinsic procedures

Fortran 90 has more intrinsic functions than FORTRAN 77, and it adds intrinsic subroutines. Therefore, a program that conforms to the FORTRAN 77 standard can have a different interpretation under Fortran 90 if it invokes a procedure that has the same name as one of the new standard intrinsic procedures. This problem is avoided if the procedure is specified in an `EXTERNAL` statement.

The CF90 compiler accepts the following intrinsic functions, which are included in the Fortran 90 standard:

- `ADJUSTL(3I)`
- `ADJUSTR(3I)`
- `ALL(3I)`
- `ALLOCATED(3I)`
- `ANY(3I)`
- `ASSOCIATED(3I)`
- `BIT_SIZE(3I)`
- `COUNT(3I)`
- `CSHIFT(3I)`
- `DIGITS(3I)`
- `DOT_PRODUCT(3I)`
- `EOSHIFT(3I)`
- `EPSILON(3I)`
- `EXPONENT(3I)`

- FRACTION(3I)
- HUGE(3I)
- KIND(3I)
- LBOUND(3I)
- LEN_TRIM(3I)
- MATMUL(3I)
- MAXEXPONENT(3I)
- MAXLOC(3I)
- MAXVAL(3I)
- MERGE(3I)
- MINEXPONENT(3I)
- MINLOC(3I)
- MINVAL(3I)
- NEAREST(3I)
- PACK(3I)
- PRECISION(3I)
- PRESENT(3I)
- PRODUCT(3I)
- RADIX(3I)
- RANGE(3I)
- REPEAT(3I)
- RESHAPE(3I)
- RRSPPACING(3I)
- SCALE(3I)
- SCAN(3I)
- SELECTED_INT_KIND(3I)

- `SELECTED_REAL_KIND(3I)`
- `SET_EXPONENT(3I)`
- `SHAPE(3I)`
- `SIZE(3I)`
- `SPACING(3I)`
- `SPREAD(3I)`
- `SUM(3I)`
- `TINY(3I)`
- `TRANSFER(3I)`
- `TRANSPOSE(3I)`
- `TRIM(3I)`
- `UBOUND(3I)`
- `UNPACK(3I)`
- `VERIFY(3I)`

The CF77 compiling system does not accept the intrinsic subroutines introduced in the Fortran 90 standard. These are as follows:

- `DATE_AND_TIME(3I)`
- `RANDOM_NUMBER(3I)`
- `RANDOM_SEED(3I)`
- `SYSTEM_CLOCK(3I)`

4.1.3 Fortran 90 and G edit descriptor output differences

The format of a floating-point zero written with a `G` edit descriptor is different in Fortran 90. The floating-point zero was written with an `EW.d` edit descriptor in FORTRAN 77, but it is written with an `FW.d` edit descriptor in the CF90 compiler.

For more information on this topic, see Section 4.2.5.1, page 102.

4.1.4 Fortran 90 and list-directed output differences

Fortran 90 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 90 output:

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

- FORTRAN 77 output:

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

4.2 Incompatibilities with extensions

Some Fortran 90 features differ in syntax and defaults in comparison to CF77 extensions. The following sections describe these differences.

4.2.1 Namelist I/O

The CF77 compiling system provided a namelist extension to handle the creation or acceptance of a wide variety of namelist records. Namelist I/O was not part of the FORTRAN 77 standard, but it is included in the Fortran 90 standard.

4.2.1.1 Differences between CF90 and CF77 namelist functionality

The CF77 namelist extension is different from the namelist feature provided in the Fortran 90 standard. Some of these extensions conflict with the new standard.

4.2.1.1.1 Format of the namelist data transfer statement

The formats of the namelist data transfer statement for the CF77 compiling system and the CF90 compiler differ. The CF77 compiling system uses the following formats:

```
READ ( unit , group [ , ERR=sn ] [ , END=sn ] )  
  
READ group  
  
PRINT group
```

Fortran 90 permits the following format:

```
READ ( unit , [ NML= ] group [ , ERR=sn ] [ , END= sn ] [ , IOSTAT=ios ] )
```

The CF90 compiler accepts all these formats.

4.2.1.1.2 Format of the namelist name and namelist termination specification

The CF77 namelist and Fortran 90 specify that the format of the namelist group name in a namelist record be as follows:

```
&namelist_group_name
```

The CF77 extension accepts a dollar sign (\$) in place of the ampersand (&) on input, but it writes only the ampersand (&) by default as the prefix to the namelist group name. As described later in this section, these defaults can be changed.

The Fortran 90 language uses the slash to terminate the namelist record. The CF77 extension uses the following form to terminate the namelist record:

```
&END
```

The CF90 extension recognizes both forms of termination of a namelist record.

The CF77 extension allows you to use other characters in place of the ampersand (&) or dollar sign (\$) on input, and the ampersand on output, as a prefix for the namelist group name and the END termination sequence. You can use the WNLDELM(3) subroutine to specify the replacement character on output, and you can use the RNLDELM(3) subroutine to specify the replacement character on input. This replacement feature is allowed in the CF90 implementation only if the file was opened as a CF77 file. A file is interpreted as a CF77 file if a program compiled with the CF77 compiling system is used to

open a file or if the file is assigned as a CF77 file through the `assign(1)` command. For more information, see the `assign(1)` man page.

The CF77 extension recognizes the slash as a termination of the namelist input but does not replace the `&END` form on output with the slash.

4.2.1.1.3 Echoing and carriage control characters in the first namelist record

The CF77 compiling system allows you to echo input records to the standard output file, `stdout`, during a namelist `READ` statement. The `RNLFLAG(3F)` subroutine enables this feature before executing the namelist `READ` statement. If this feature is not enabled, the `READ` statement skips the first character. The `E` is the default character for the echoing feature.

The `WNLFLAG(3F)` subroutine can be invoked before a namelist `WRITE` statement to indicate the character to be placed in the first column of the first namelist output record. The `WNLFLAG(3F)` subroutine can also be used to specify the carriage control character to be used in the output record. The default character used by the CF77 extension in column one of a namelist output record is a blank.

Fortran 90 namelist input may begin in column one of the first namelist input record. The CF90 implementation does not support an echo or carriage control character in column one of the first namelist record unless the file is opened as a CF77 file.

4.2.1.1.4 Format of the namelist *name=value* sequence

A CF77 extension allows you to change the equal sign (=) in the *name=value* sequence in a namelist record. You can use the `WNLREP(3F)` subroutine to indicate the replacement character to be used by a namelist `WRITE` statement. You can use the `RNLREP(3F)` subroutine to indicate the replacement character to be used during a namelist `READ` statement.

Fortran 90 does not provide the ability to change the equal sign in the *name=value* sequence. The CF90 implementation does not support a replacement character for the equal sign in the *name=value* sequence unless the file is opened as a CF77 file.

The size of the value written by a CF90 namelist `WRITE` statement is the same size as a value written by a CF90 list-directed `WRITE` statement. The size differs from that in the CF77 extension. The size of a double-precision floating-point value in CF77 namelist output can be only 27 characters, but it can be 34 characters in CF90 namelist output.

4.2.1.1.5 Value separators in namelist records

Fortran 90 allows the blank, comma, or the slash as value separators. A CF77 extension allows the comma as the default value separator but allows a replacement character to be specified for the comma through the `WNLSEP(3F)` subroutine for a namelist `WRITE` statement or the `RNLSEP(3F)` subroutine for a namelist `READ` statement. The CF90 implementation does not provide the ability to change the value separator in namelist I/O unless the file was opened as a CF77 file.

4.2.1.1.6 Character data format in a namelist record

A CF77 extension supports both delimited and undelimited character string input. Fortran 90 states that namelist character input must be delimited by apostrophes or quotation marks. Character constants produced by Fortran 90 namelist output statements are not delimited by apostrophes or quotation marks unless the `DELIM=` specifier was present on the `OPEN` statement for the file in the namelist output statement. The CF77 extension delimits a character string with apostrophes on output.

The CF90 implementation does not support undelimited character input unless the file is opened as a CF77 file. The CF90 implementation delimits character string output with apostrophes or quotation marks only if the `DELIM` specifier for the file indicates that apostrophes or quotation marks are to be written on output.

4.2.1.1.7 Comments in namelist input records

The CF77 extension allows comments in namelist input records. These comments are indicated by the use of the semicolon (`:`). You can use the `RNLCOMM(3F)` subroutine to allow a different character to indicate the presence of a comment in the namelist input record. If the specified comment character is detected within a namelist record, the rest of the namelist record is skipped.

Fortran 90 does not provide for comments in namelist input records. The CF90 compiler supports this feature when the namelist input file is opened as a CF77 file.

4.2.1.1.8 `RNLSKIP(3F)` and namelist records

A CF77 extension provides the `RNLSKIP(3F)` subroutine, which takes a specific action when a namelist group object name does not match the namelist group object name in the namelist record. Fortran 90 does not provide this option.

The CF90 implementation supports this feature for input files opened as CF77 files only.

4.2.1.1.9 Structures and namelist records

The CF77 compiling system does not support derived data types and structures. Fortran 90 allows structures in namelist I/O.

The format of the namelist output of a structure is the structure name, followed by an equal sign, and then followed by the value of each component of that structure. The values are separated by commas. None of the individual component names is written by the namelist write of a structure. Because the CF77 extension does not accept structures, the CF90 implementation issues an error if it detects that the type of the namelist item is a structure and if the namelist file was opened as a CF77 file.

4.2.1.1.10 Array sections and namelist records

The CF77 compiling system does not accept the presence of array section notation, such as $x(3:4)=$, in a namelist input record. Fortran 90 and the CF90 implementation allow section notation in a namelist input record.

4.2.1.2 Similarities between CF90 and CF77 namelist input

Both the CF77 compiling system and the CF90 compiler accept the ampersand (&) preceding the namelist name. The CF77 extension and the CF90 implementation accept the slash (/) as the end of the namelist input. The CF77 extension and the CF90 implementation accept the &END as the end of the namelist input. The following is an example of namelist input data:

```
&TODAY I = 12345, X(1)=12345, X(3:4) = 2*1.5, I=6,P = "data"/
```

The CF90 implementation accepts Hollerith constants in namelist input records. Note that Hollerith data is an outmoded feature. See Chapter 6, page 161, for information on outmoded features and preferred alternatives.

4.2.1.3 Differences between CF90 and CF77 namelist output

The following sections describe the differences in namelist output between the CF90 compiler and the CF77 compiling system.

4.2.1.3.1 CF90 features not in CF77 namelist

The CF90 compiler contains the following features that were not in the CF77 compiling system:

- The CF90 record length for a namelist output record is obtained from a `RECL=` specifier on the `OPEN` statement for this file or from a default size.
- The CF90 compiler provides structures and a double complex data type.

4.2.1.3.2 CF77 namelist features and CF90 namelist

The CF77 compiling system provides several user output control subroutines that can be called to provide portable namelist output. These were provided before the addition of namelist I/O to the Fortran standard. The facilities provided by these routines are only available through the CF90 compiler when the `-f 77` option is used on the `assign(1)` command for a specific file or if you open a file with a procedure compiled with the CF77 compiling system. This facility allows you to use a file containing namelist data with either the CF77 compiling system or with the CF90 compiler. Errors are generated if any new CF90 features are in a namelist that is written to this CF77 file.

<u>CF77 routine</u>	<u>Description</u>
<code>CALL WNLLONG(<i>len</i>)</code>	Sets the output line length to <i>len</i> . For the CF77 compiling system, the minimum is 8 and the maximum is 196. If <i>len</i> is too short, the program aborts. If <i>len</i> is equal to -1, the line length defaults to 267. For the CF90 compiler, the <code>RECL=</code> specifier is used, if present. Otherwise, it defaults to 267.
<code>CALL WNLDELM(<i>char</i>)</code>	Sets the character preceding the group name and the <code>END</code> terminator. An ampersand (&) is the default <i>char</i> .
<code>CALL WNLSEP(<i>char</i>)</code>	Changes the separator character immediately following each value from a comma (,) to <i>char</i> .
<code>CALL WNLREP(<i>char</i>)</code>	Changes the assignment operator between the namelist output variable name and the values from an equal sign (=) to <i>char</i> .
<code>CALL WNLFLAG(<i>char</i>)</code>	Changes the character written in column 1 during a namelist output statement from a blank to <i>char</i> . The <i>char</i> is used for carriage control if the output

is printed or to force echoing if the output is used as input for a namelist READ statement.

CALL WNLLINE(*value*)

Begins each namelist variable name on a new line. If *value* is 0, no new line is generated. If *value* is 1, a new line is generated for each variable.

4.2.1.4 Portability between CF77 and CF90 namelist

The CF77 compiling system does not accept structures or the double complex data type. The CF90 compiler accepts the user output control subroutines for namelist output, but you must indicate that the file is to be treated as a CF77 file. Zero-length entities are not accepted from and are not written to files that are compatible with the CF77 compiling system.

To choose the CF77 format of namelist output with its additional subroutine choices, you can do one of two things:

- Open the file with a procedure that was compiled with the CF77 compiling system.
- Assign the file with the `-f 77` option on the `assign(1)` command, as shown in the following example:

```

INTEGER i(2), j(2,2)
NAMELIST /abc/ i, j
i(1) = 1
i(2) = 3
j = 22
WRITE(6,abc,IOSTAT=ios1)
IF(ios1 .NE. 0) THEN
    PRINT *, 'write iostat = ', ios1
    IF (ios1 .EQ. 1328) PRINT *, 'expected iostat = 1328'
ENDIF
PRINT *, 'end of f90 namelist output'
WRITE(7,abc,IOSTAT=ios1)
IF (ios1 .NE. 0) THEN
    PRINT *, 'write iostat = ', ios1
    IF (ios1 .EQ. 1328) PRINT *, 'expected iostat = 1328'
ENDIF
PRINT *, 'end of CF77 namelist output'
PRINT *, 'end of program'
END

```

The output of the program for a conventional CF90 compilation and execution is as follows:

```
% assign -R
% a.out
&ABC I = 1, 3, J = 4*22 /
end of f90 namelist output
end of CF77 namelist output
end of program
% more fort.7
&ABC I = 1, 3, J = 4*22 /
```

The output of the program when specifying the `-f 77` option on the `assign(1)` command is as follows:

```
% assign -R
% assign -f 77 u:6
% assign -f 77 u:7
% rm fort.7
% a.out
&ABC I = 1, 3, J = 4*22 &END
end of f90 namelist output
end of CF77 namelist output
end of program
% more fort.7
&ABC I = 1, 3, J = 4*22 &END
```

The namelist library routine for CF77 namelist accepts the slash as the terminator in both UNICOS 7.0 and UNICOS 8.0.

4.2.2 List-directed I/O

Fortran 90 provides two new additions to list-directed I/O. These are the ability to write delimited character strings during list-directed output and the ability to use an internal file in a list-directed I/O statement.

4.2.2.1 Delimited and undelimited character strings in list-directed I/O

The CF77 compiling system supports only delimited character string input to a list-directed item that will be stored to a list item of type character. The CF77 compiling system writes only undelimited character strings from a list-directed output list item of type character.

4.2.2.2 List-directed I/O and internal files

An internal file on CF77 compiling systems can be a positive, nonzero-length character variable or array. It cannot be a character array section.

4.2.2.3 List-directed I/O and Hollerith constants

Fortran 90 does not support Hollerith constants in list-directed input files, but the CF90 system provides this as an extension. Note that Hollerith data is an outmoded feature. See Chapter 6, page 161, for information on outmoded features and preferred alternatives.

4.2.2.4 List-directed I/O and floating-point zero

Fortran 90 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 under CF77 6.0:

```

% cft77 tt.f
% segldr tt.o
% a.out
0.
      0.00E+00

```

It generates different output for the CF90 compiler:

```

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

```

4.2.3 OPEN statement

Fortran 90 provides several new specifiers for the OPEN statement and additional values for existing specifiers. The POSITION, PAD, and ACTION specifiers are new specifiers. The RECL, STATUS, and DELIM specifiers may have additional values. Neither the new specifiers nor the additional values to the existing specifiers are accepted by the CF77 compiling system.

4.2.4 INQUIRE statement

There are several new specifiers for the `INQUIRE` statement and additions to existing specifiers. The new specifiers are `IOLength`, `POSITION`, `ACTION`, `READ`, `WRITE`, `READWRITE`, `DELIM`, and `PAD`. The `NUMBER`, `RECL`, and `BLANK` specifiers are existing FORTRAN 77 specifiers that have some additions in Fortran 90. Neither the new specifiers nor the additional values to the existing specifiers are accepted by the CF77 compiling system.

4.2.5 READ and WRITE statements

The following sections explain the differences and similarities between the `READ` and `WRITE` statements in the CF77 compiling system and the CF90 compiler.

4.2.5.1 Differences in the `G` edit descriptor

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 90 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 `Fw.d`-formatted value by the CF77 compiling system and as an `Ew.d`-formatted value by the CF90 compiler.

Consider the following code fragment:

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

The preceding code generates the following output under CF77 6.0:

```
0.00E+00
```

It generates different output under CF90:

```
0.0
```

4.2.5.2 Differences in the B, O, and Z edit descriptors

The B, O, and Z edit descriptors are available in Fortran 90. They are limited to integer I/O list items. The CF77 compiling system allows real and other data types to be described with these edit descriptors. The B edit descriptor from Fortran 90 is implemented in the CF77 compiling system.

The CF77 compiling system allows the use of signed octal and hexadecimal values with OW and ZW, but signed octal and hexadecimal values are not allowed in Fortran 90. The CF90 compiler allows signed input values, but it writes only unsigned values.

If the size of the value is less than *w* in OW or ZW on output, the CF77 compiling system pads on the left with zeros. Fortran 90 requires blank padding on the left. If the edit descriptor OW.*m* or ZW.*m* is used, the field must contain at least *m* digits. The .*m* form is one way to get leading zeros with Fortran 90. If the size of the value is greater than *w* in OW or ZW on output, the CF77 compiling system truncates the values while Fortran 90 fills the field with asterisks. The CF90 compiler provides the Fortran 90 form of OW and ZW output unless the file was opened as a CF77 file.

4.2.5.3 Implied-DO variables in an I/O list

Values of implied-DO variables may not be relied upon when the END=, ERR=, or IOSTAT= values indicate that the statement was not completed. Both FORTRAN 77 and Fortran 90 indicate that these values are undefined. Some CF77 programs may be relying on the values in these variables but these variables may not contain the same values in a CF90 program.

4.2.6 Common blocks and I/O

Some programs using common blocks might rely on the order of the allocation of common blocks by the compiler and `segldr(1)`. This order might not be the same in the CF90 system. This affects programs using `BUFFER IN` and `BUFFER OUT` that transfer data across multiple common blocks in a single statement. The order of separate common blocks may not be used in a CF90 program. The `SEGLDR COMMONS` directive can be used to specify an order of common blocks for testing. Such programs should be changed to use one common block rather than several common blocks to ensure the order and size of a common block.

4.3 CF90 restrictions on CF77 I/O extensions

The `ENCODE`, `DECODE`, `BUFFER IN`, and `BUFFER OUT` statements are CF77 I/O extensions. The CF90 compiler does not allow array sections in `ENCODE` and `DECODE` statements as the source or target.

`ENCODE` and `DECODE` statements are restricted to formatted I/O. The addition of list-directed I/O on an internal file is not extended to these statements.

Note that `ENCODE` and `DECODE` statements are supported as outmoded features. See Chapter 6, page 161, for information on outmoded features and preferred alternatives.

Array sections are not allowed in `BUFFER IN` and `BUFFER OUT` statements in a CF90 compilation.

4.4 CF90 and CF77 integrated environment differences

The following sections describe differences in loading program units, the `assign(1)` command, and the I/O environment.

4.4.1 Loading CF77 and CF90 program units

The `f90(1)` command line allows you to do the following:

- Link `.o` files produced by the CF77 compiling system with the new libraries
- Compile and link FORTRAN 77 codes with the CF90 compiler
- Compile and link Fortran 90 codes
- Compile Fortran 90 code and link with `.o` files produced by the CF77 compiling system

When linking `.o` files produced by both the CF77 compiling system and the CF90 compiler, the interaction of I/O on the following types of files can result in some differences in format of output values for formatted I/O or in the form of namelist output:

- A file opened by a `.o` file produced by the CF77 compiling system but updated by a `.o` file produced by the CF90 compiler
- A file opened by a `.o` file produced by the CF90 compiler but updated by a `.o` file produced by the CF77 compiling system

The `.o` files produced by the CF90 compiler must not be loaded with default UNICOS 7.0 libraries.

4.4.2 The `assign(1)` command and the CF77 file attribute for the CF90 compiler

The `-f 77` option on the `assign(1)` command allows you to specify a CF77 attribute on a per-file basis for a CF90 program unit. With this facility, a program unit compiled with the CF90 compiler can specify that a file is to be treated as a CF77 file. A separate program unit that is compiled by the CF77 compiling system is not needed to open the file as a CF77 file.

The following example shows the format of the option:

```
% assign -f 77 u:8
```

This form is allowed for units 5 and 6 but is ignored for the default units attached to the asterisk unit on input and output; that is, `READ(*, GROUP)`.

4.4.3 New I/O environment

More data typing information is provided to the library for foreign dataset conversion using flexible file I/O (FFIO). This information is not available for program units compiled with the CF77 compiling system.

4.5 CF90 extensions to Fortran 90

The following sections summarize the CF90 extensions to the Fortran 90 standard. These extensions consist, primarily, of CF77 extensions to the FORTRAN 77 standard that are also CF90 extensions to Fortran 90. The topics discussed are as follows:

- Source forms, character sets, and compiler directives
- Data types and constants
- `DATA`, `COMMON`, and `EQUIVALENCE` statements
- Expressions and assignment
- I/O, including `FORMAT` statements
- Flow control and other statements
- Program units, functions, subroutines, and statement functions

- Intrinsic procedures

4.5.1 Source forms, character sets, and compiler directives

The CF90 compiler allows the following:

- \$ in identifiers but not as the first character.
- TAB character in both free and fixed forms. The *Fortran Language Reference Manual, Volume 1*, publication SR-3902, describes the fixed form rules for expanding TAB characters. In free form, TAB characters are treated as blank characters.
- Fixed source form lines greater than 72 characters. By default, columns 73 through 132 are ignored. The `-N col` command line option extends the source line. When `-N 80` is specified, columns 1 through 80 are used. When `-N 132` is specified, columns 1 through 132 are used. Fortran 90 restricts fixed source form lines to 72 characters.
- 99 continuation lines in fixed source form (100 total lines) and 99 continuation lines in free source form (100 total lines). Fortran 90 limits continuation lines to 19 in fixed form and 39 in free source form.
- CDIR\$ or CDIR@ starting in column 1 of fixed source form to indicate compiler directives. The CF90 compiler allows !DIR\$ or !DIR@ as the first nonblank character of a free or fixed source form line to indicate compiler directives.

4.5.2 Data types and constants

The CF90 compiler allows the following Boolean (typeless) constants:

<u>Notation</u>	<u>Form</u>
<code>dddddB</code>	Octal, where <i>d</i> is an octal digit
<code>'ddd' O</code> and <code>"ddd" O</code>	Octal, where <i>d</i> is an octal digit
<code>X'ddd'</code> and <code>X"ddd"</code>	Hexadecimal, where <i>d</i> is a hexadecimal digit
<code>'ddd' X</code> and <code>"ddd" X</code>	Hexadecimal, where <i>d</i> is a hexadecimal digit
<code>nH . . . , nL . . . , nR . . . , ' . . . ' H , ' . . . ' L , ' . . . ' R , " . . . " H , " . . . " L , and " . . . " R</code>	Hollerith constant. Note that Hollerith data is an outmoded feature. See Chapter 6, page 161, for

information on outmoded features and preferred alternatives.

Variables cannot be declared to be Boolean (typeless) type.

The CF90 compiler allows the following BOZ constants in places other than DATA statements. The CF77 compiling system treated these as integer constants. The CF90 compiler treats these as typeless constants. If they are assigned to a real variable in the CF77 compiling system, type conversion will occur. If they are assigned to a real variable in the CF90 compiler, no type conversion will occur.

- B' *bbb*'
- O' *ooo*'
- Z' *zzz*'
- B" *bbb*"
- O" *ooo*"
- Z" *zzz*"

The CF90 compiler supports Cray pointer and Cray character pointer types, as well as Fortran 90 pointers. Cray pointers and Cray character pointer types cannot be components of derived data types. Cray pointers cannot have derived types as pointees.

The CF90 compiler supports the following data type declaration forms in type declaration statements, FUNCTION statements, and IMPLICIT statements:

- Integer data type declaration forms:
 - INTEGER*1
 - INTEGER*2
 - INTEGER*4
 - INTEGER*8
- Real data type declaration forms:
 - REAL*4
 - REAL*8
- Complex data type declaration forms:

- COMPLEX*8
- COMPLEX*16
- Logical data type declaration forms:
 - LOGICAL*1
 - LOGICAL*2
 - LOGICAL*4
 - LOGICAL*8

Note that the * form for declaration statements is supported as an outmoded feature. See Chapter 6, page 161, for information on outmoded features and preferred alternatives. For more information on the type declaration forms, see the *Fortran Language Reference Manual, Volume 1*, publication SR-3902.

The CF90 compiler allows `.T.` and `.F.` as alternate forms for logical constants `.TRUE.` and `.FALSE.`, respectively. This is true only if `.T.` and `.F.` have not been defined as defined operators.

The CF90 compiler allows the `DOUBLE COMPLEX` statement. For information on the `DOUBLE COMPLEX` statement when `-d p` is specified on the `f90(1)` command line, see Section 6.9, page 176, and the `f90(1)` man page. Note that the `DOUBLE COMPLEX` statement is supported as an outmoded feature. See Chapter 6, page 161, for information on outmoded features and preferred alternatives.

The CF90 compiler allows named constants in a complex constant.

The CF90 compiler allows double-precision data items to be treated as real through use of the `-d p` command line option. This allows double-precision objects to map to real and double complex objects to map to complex. This also causes double precision and double complex intrinsics to map to their corresponding real and complex intrinsics. When `-d p` was used with the CF77 compiling system, objects declared as `REAL*16` or `DOUBLE PRECISION*16` became single precision reals. They remain double precision with the CF90 compiler. Note that the `DOUBLE PRECISION` keyword is supported as an outmoded feature. See Chapter 6, page 161, for information on outmoded features and preferred alternatives.

The CF90 compiler allows a character constant or Hollerith to be used in a context in which character constants are not allowed. Character and Hollerith constants are allowed in arithmetic and logical expressions, and in assignment statements where the left side of the equal sign is an entity of type integer or

real. Note that Hollerith data is an outmoded feature. See Chapter 6, page 161, for information on outmoded features and preferred alternatives.

4.5.3 Declaring attributes, COMMON, DATA, EQUIVALENCE, SAVE, FUNCTION, and SUBROUTINE statements

The CF90 compiler allows the following:

- The SUBROUTINE and FUNCTION statements accept the PURE and ELEMENTAL prefixes.
- More than one SAVE statement in a scoping unit when a SAVE statement without an entity list appears.
- Multiple initialization of entities in DATA statements. It is indeterminate as to which of the values assigned to the variable is the final value.
- A noncharacter array to be filled with character data in a DATA statement.
- Objects in common can be initialized with DATA statements in program units other than BLOCK DATA program units. It is recommended that you use BLOCK DATA program units to initialize common blocks. It is acceptable to initialize common blocks in a single compilation unit.
- The same named common block can be specified in more than one BLOCK DATA program unit.
- A single Hollerith or character constant value can be used to initialize multiple elements of an integer or single-precision real array in a DATA statement.
- An array in a DATA statement can be partially initialized if it is the last variable in the variable list. (Fortran 90 requires the number of entities in the variable list to match the number of constants in the constant list.)
- Character and noncharacter data objects can be equivalenced.
- The TASK COMMON statement. This statement is not allowed in modules, contained procedures, or interface bodies.
- Common blocks with the same name can be different sizes. Fortran 90 requires the size of named common blocks with the same name to be the same size in all scoping units of the executable program.
- The CF90 compiler treats all common block variables as having the SAVE attribute, which results in such variables remaining defined at execution of a

RETURN or END. Fortran 90 specifies that variables in a named common block become undefined at execution of a RETURN or END statement, unless the named common block is in at least one scoping unit making a direct or indirect reference to the subprogram.

- The CF90 compiler does not enforce the requirement that a named common block must appear in a SAVE statement in each scoping unit in which it appears, if it is in a SAVE statement in any scoping unit.
- User-defined external functions can be used in dimension bounds expressions for variably dimensioned arrays and for automatic arrays. Module procedure (function) references and internal function references are not allowed in dimension bounds expressions.
- A local variable that does not have the SAVE attribute can be given the AUTOMATIC attribute in a type specification statement or in an AUTOMATIC statement.

4.5.4 Expressions and assignments

The CF90 compiler allows the following with regard to expressions and assignment:

- An array reference with fewer subscript expressions than the rank of the array. The lower bound is assumed for each missing subscript. Fortran 90 requires that the number of subscript expressions match the rank of the array.
- Masking expressions in which a bitwise logical operator operates on bits within an integer, single-precision real, Cray pointer, or Boolean value and generates a Boolean result type. If a user-defined interface defines one of these operators for these data types, the intrinsic definition is overridden for those types.
- .N., .A., .O., .X., and .XOR. logical operators as alternate forms for .NOT., .AND., .OR., and .NEQV., respectively, when these forms have not been defined as a defined operator.

If you declare a dot operator or logical literal constant to be a defined operator, the extension is not allowed.

Example:

```
INTERFACE OPERATOR(.A.)
  FUNCTION a_op(l, r)
    LOGICAL a_op
```

```

    INTEGER, INTENT(IN) :: l
    REAL, INTENT(IN) :: r
    END FUNCTION
END INTERFACE

```

.A. is a defined operator, so it cannot be used as an abbreviation for .AND..

- The functional forms `COMPL(a)`, `AND(a, b)`, `OR(a, b)`, `XOR(a, b)`, `NEQV(a, b)`, and `EQV(a, b)` as replacements for logical operators. Fortran 90 does not recognize these forms as intrinsic functions. Note that the `COMPL` and `AND` forms are supported as outmoded features. See Chapter 6, page 161, for information on outmoded features and preferred alternatives.
- Complex and double-precision types to be mixed in exponentiation operations.
- The intrinsic functions `LLE(3I)`, `LLT(3I)`, `LGE(3I)`, and `LGT(3I)` to be passed as actual arguments.
- Redundant parentheses in expressions.

4.5.5 I/O, including `FORMAT` statements

The CF90 compiler allows the following with regard to `FORMAT` statements and I/O:

- The `BUFFER IN` and `BUFFER OUT` I/O statements.
- The `ENCODE` and `DECODE` I/O statements. Note that the `ENCODE` and `DECODE` statements are supported as outmoded features. See Chapter 6, page 161, for information on outmoded features and preferred alternatives.
- A repeat count before the slash (/) edit descriptor. Fortran 90 classifies the slash edit descriptor as nonrepeatable.
- The `DW.dEe` form of edit descriptor `D`.
- The asterisk (*) characters to delimit character strings in a format. Note that this practice is supported as an outmoded feature. See Chapter 6, page 161, for information on outmoded features and preferred alternatives.
- Empty parentheses groups in a format descriptor.
- The `RW` and `$` edit descriptors. Note that the `RW` edit descriptor is supported as an outmoded feature. See Chapter 6, page 161, for information on outmoded features and preferred alternatives.

- A negative n in the nX edit descriptor.
- A comma to precede a right parenthesis in a format descriptor.
- A file identifier in a control information list on an I/O statement to be a Hollerith string of characters enclosed in single quote or quotation marks. Fortran 90 does not provide for external file identifiers. The CF90 compiler allows file identifiers to be used in place of unit identifiers.
- A format identifier on an I/O statement to be the name of an array of type integer, real, or logical.
- An I/O statement to contain a specifier that is a character expression involving concatenation of a character entity with length declared (*).
- An assumed-size array to be an internal file.
- A WRITE statement with no unit specifier. The syntax for a WRITE statement can look like that of the PRINT statement.
- NAMELIST statements to be mixed in with executable statements. All references to the NAMELIST group name must follow all definitions of the NAMELIST group name.

The CF90 compiler requires commas in a format list only between two adjacent digits belonging to different list items, between two adjacent quotes or apostrophes of separate edit descriptors, and after a D, E, or G descriptor that precedes an E descriptor. Fortran 90 allows the comma to be optional only in the following cases: before or after a slash or colon descriptor and immediately following an F, E, D, or G edit descriptor.

Example. The following program shows the use of $-n X$, the quoted string, and the use of asterisks to delimit a string in a format:

```
A = 1.0
B = 2.0
C = 3.0
WRITE(6,1) A,B,C
1  FORMAT(2X,-1X,"QUOTESTRING",F8.2,1/,* ASTERSTRING*,
+       F8.2,2/,10X,-5X,'APOSTSTRING',F8.2)
END
```

The output is as follows:

```
> QUOTESTRING      1.00
> ASTERSTRING      2.00
>
```



```
>  
>      APOSTSTRING      3.00
```

In this example, note that the use of `-n X` is supported as an outmoded feature. See Chapter 6, page 161, for information on outmoded features and preferred alternatives.

4.5.6 Flow control and other statements

The CF90 compiler allows the `STOP` statement and `PAUSE` statement stop code to be an unsigned integer, character constant (the maximum is 80 characters), a character variable, a character array element, or a character function. Fortran 90 requires the stop code to be a *scalar_char_constant* or 1 to 5 digits.

The CF90 compiler allows transfer of control into a `DO` or `IF` block; this allows for extended-range `DO` loops. The CF90 compiler also supports a command line option that forces at least one execution of `DO` loops.

4.5.7 Program units, functions, subroutines, and statement functions

The CF90 compiler allows a parenthesized list following the program name in a `PROGRAM` statement. The list is ignored.

The CF90 compiler allows up to 26 unnamed `BLOCK DATA` program units in an executable program. Fortran 90 allows only one such program unit.

The CF90 compiler accepts the `PURE` and `ELEMENTAL` prefixes as attributes on the `SUBROUTINE` and `FUNCTION` statements.

Fortran 90 provides a `RESULT` variable specifier for direct recursive functions to distinguish between a reference to the function result variable and a reference to the function. The CF90 compiler allows use of the same name for references to the function result variable and the function itself, and distinguishes between them by the form of reference. This is allowed only for a function with a scalar result of type real, logical, integer, double precision, complex, or character. The `RECURSIVE` attribute must also be specified for the function name.

4.5.8 Call by value

The CF90 compiler allows the `%VAL` intrinsic function for passing arguments by value. Typically, the CF90 compiler passes arguments by passing the address of the argument instead of the value of the argument. `%VAL` tells the compiler to pass the value instead of the address.

The %VAL intrinsic can be used only within an actual argument list of a function or subroutine call. It cannot be used as an argument to an overloaded operator. Because of the ambiguity of the % character, %VAL is not recognized outside of its allowed context.

The format for this intrinsic function is as follows:

$i = c_function(\%VAL(k))$

The %VAL intrinsic function cannot be used as an argument to a Fortran 90 procedure.

4.5.9 Intrinsic procedures

The CF90 compiler supports several intrinsic procedures, all of which are functions, as extensions to the Fortran 90 standard. Some of these are outmoded. Some do not work on all platforms supported by the CF90 compiler. Chapter 6, page 161, lists the outmoded intrinsics and the recommended standard alternatives.

The *Fortran Language Reference Manual, Volume 2*, publication SR-3903, and the *Intrinsic Procedures Reference Manual*, publication SR-2138, describe the intrinsic procedures in more detail. The *Intrinsic Procedures Reference Manual*, publication SR-2138, also contains copies of the online man pages that describe each intrinsic procedure. See the man pages in that manual for more information on the intrinsic procedures implemented as extensions to the Fortran 90 standard.

4.6 CF90 and CF77 implementation differences

The following sections describe implementation differences with regard to the bit matrix multiply (BMM) intrinsic functions and between various other miscellaneous features.

4.6.1 BMM intrinsic function differences (UNICOS systems only)

The following sections describe aspects of using the BMM intrinsics with regard to integer types, integer constants, and vectorization.

4.6.1.1 Integer types

The CF90 compiler is more restrictive than the CF77 compiling system when it comes to integer types. Because of this, some programs written for BMM operations using the CF77 compiling system either do not compile cleanly or do not run identically with the CF90 compiler due to the stricter integer precision requirements.

To force 64-bit precision with the CF90 compiler, declare bit matrix arrays to be `INTEGER(KIND=8)`. Alternatively, you can use the `f90(1)` command line options `-i 64` and `-s default64` to force all integer arithmetic to be performed using 64 bits, but this is discouraged because it is a global change and can degrade performance significantly.

4.6.1.2 Integer constants

Default-sized integer constants used in logical operations do not guarantee a full 64 bits of precision because `-O fastint` is on by default on the `f90(1)` command line. To make integer constants 64-bits wide, you have to convert them to the proper type. There are two ways to do this:

- With the two-argument form of the `INT(3)` intrinsic:
`ISHFT(INT(-1 , 8) , AMOUNT)`
- By including the kind value in the constant itself: `ISHFT(-1_8 , AMOUNT)`

Without this conversion, you receive only the lower 46 bits of the expression in the result by default.

4.6.1.3 Vectorization

The CF90 compiler does not require vectorization to be enabled for most simple array syntax statements to have vector code generated for them. Applications that use array syntax statements for BMM operations exclusively can usually be compiled with optimization disabled. However, vectorization of array syntax with optimization disabled is neither guaranteed or optimal.

4.6.2 Miscellaneous differences

The following paragraphs describe miscellaneous differences between using the CF77 compiling system and the CF90 compiler.

- The CF77 compiling system allows assumed-size character dummy procedures. The CF90 compiler does not allow this.

- Because of the Fortran 90 rules of type conformance, the CF90 compiler might be more restrictive than the CF77 compiling system with regard to intrinsic assignment.
- The CF77 compiling system allowed `COMPLEX*4`. No other vendor appears to allow this. The CF90 compiler does not allow it, but it allows `COMPLEX(KIND=4)`, which is equivalent to `COMPLEX*8`. Note that the `*` form for declaration statements is supported as an outmoded feature. See Chapter 6, page 161, for information on outmoded features and preferred alternatives.
- The CF90 compiler requires the `RECURSIVE` keyword for recursive routines. The following code compiles without error with the CF77 compiling system, but an error is generated when compiled with the CF90 compiler. The CF90 treatment of this code complies with the Fortran 90 standard, and it allows the compiler to diagnose accidental causes of recursion:

```
INTEGER FUNCTION I()
  I = I() + 1
END
```

- Unlike the CF77 compiling system, the CF90 compiler generates a compile-time error when an `IMPLICIT` statement follows a `PARAMETER` statement if the information on the `IMPLICIT` statement contradicts the type of the named constant, as in the following example:

```
PARAMETER(A=1)
IMPLICIT INTEGER(A-Z)
PRINT *, A
END
```

The CF90 treatment of this code is consistent with the Fortran 90 standard, and it avoids potential ambiguities.

- Treatment of `BOZ` and typeless constants is different between the CF77 compiling system and the CF90 compiler in `DATA` statements. Consider the following code fragment:

```
COMPLEX C1, C2
REAL R1, R2

DATA C1 /O'77'/
DATA C2 /Z'10'/
DATA R1 /X'77'/
DATA R2 /77B/
```

```
PRINT *, C1, C2, R1, R2
END
```

The CF77 compiler produces no diagnostic messages at compile time for the preceding code. Execution results in the following:

```
% a.out
(63.,0.), (16.,0.), 0.E+0, 0.E+0
```

The CF90 compiler generates a compile-time message for each DATA statement. The CF90 treatment of this code is consistent with the Fortran 90 standard and is more consistent than the CF77 compiling system in handling constants.

- When the value of an expression depends on the order of evaluation, and the order of evaluation is processor dependent, the CF90 compiler and the CF77 compiling system may evaluate the items in a different order. This can lead to differences in the generated output. The following example code contains such expressions:

```
PROGRAM CPROP

      RBIG = 1.2E+83
! Use of variable in expression
      R20 = RBIG - 1.2E+83 + 20
      PRINT *, "Expected: 20"
      PRINT *, "Received: ", R20
! Use of all constants in expression
      R20 = 1.2E+83 - 1.2E+83 + 20
      PRINT *, "Expected: 20"
      PRINT *, "Received: ", R20

      END
```

The expression $1.2E+83 - 1.2E+83 + 20$ can yield different answers depending on whether the $-$ or $+$ operation is evaluated first. This is due to the large difference in magnitude of the operands and the fixed precision of the machine. The order of evaluation in these expressions is *processor dependent*, according to the Fortran 90 standard. The CF90 compiler evaluates the operators of the first expression in a different order from the CF77 compiling system. It also evaluates the operators of the second expression in a different order from the order it evaluates the first expression due to the different syntax.

- FORTRAN 77 provided one precision of integer, complex and logical data and two precisions of real data. Fortran 90 allows an implementation to have any number of precisions for these data types. Many vendors provided additional precisions as an extension to their FORTRAN 77 implementations through the *type*byte_count* form of declaration. The CF77 compiling system accepted these extensions, but it mapped them onto the basic types required by the FORTRAN 77 standard. The CF90 compiler, like the CF77 compiling system, accepts this syntax but treats these additional data types as distinct types. This is done to allow for unambiguous resolution of procedure interfaces, overloaded operators, and user defined generics. Because of this difference between the CF77 compiling system and the CF90 compiler, some CF77 programs that employ the *type*byte_count* syntax may not be accepted by the CF90 compiler. These differences occur in the definition and use of statement functions. For example:

```
REAL*4 X
STMT_FUNC(R) = R + 1.0      ! Statement function
                               ! definition. . .
X = STMT_FUNC(X)
```

The preceding program fragment would compile without error with the CF77 compiling system. On a platform where the default real kind is 8, the CF90 compiler issues an error for the statement function use because the statement function is defined with a default real argument but is passed a nondefault real actual argument. You can use the `-s cf77types` option on the `f90(1)` command line to avoid receiving this error message. Note that the `*` form for declaration statements is supported as an outmoded feature. See Chapter 6, page 161, for information on outmoded features and preferred alternatives.

- The CF77 compiling system and the CF90 compiler differ in the way overindexed code is handled.

The CF77 compiling system permitted constructs such as the following:

```
SUBROUTINE (A, N)
DIMENSION A(20, 400)
DO I = 1,N
  A(I) = . . .
END DO
```

The CF90 compiler does not overindex if the leading dimension is known. Use the `-O overindex` option on the `f90(1)` command if you want to overindex an array. For more information on the `f90(1)` command, see the *CF90 Commands and Directives Reference Manual*, publication SR-3901.

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 90 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 Research hardware documentation. It does not agree with some conventions used in other Cray Research 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 145.

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 as follows:

- On UNICOS systems other than CRAY T90 systems: $-2^{46} < I < 2^{46}$ or approximately $-10^{13} < I < 10^{13}$.
- On CRAY T90 systems: $-2^{52} < I < 2^{52}$ or approximately $-10^{15} < I < 10^{15}$.

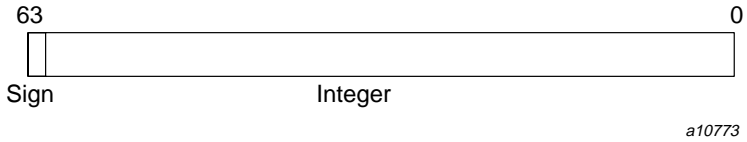


Figure 1. Default 64-bit integers

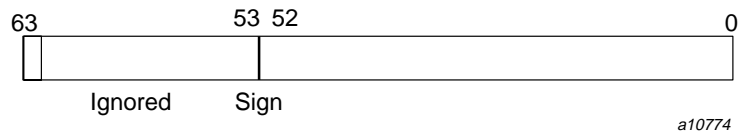


Figure 2. Fast integer operations with `INTEGER(KIND=8)`, CRAY T90 systems

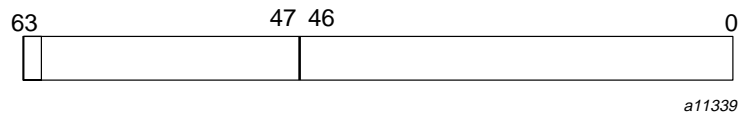


Figure 3. Fast integer operations with `INTEGER(KIND=8)`, UNICOS systems (except CRAY T90 systems)

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:

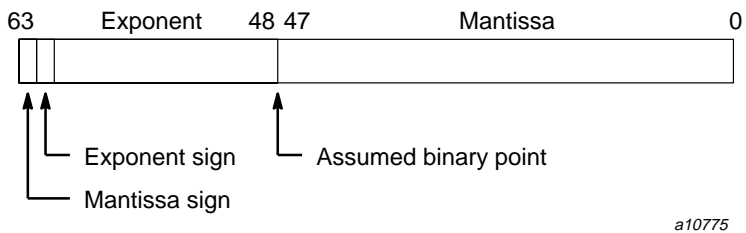


Figure 4. Real type

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 $+10.0$.

Note that a real data object with `KIND=4` has the same internal representation as 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
```

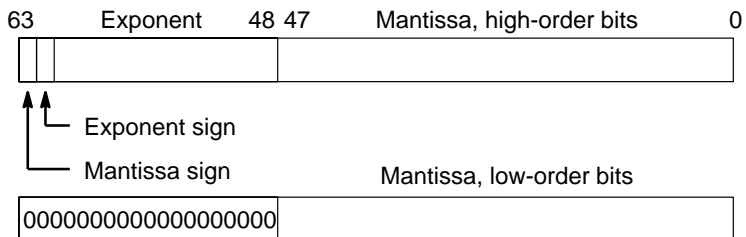
5.1.3 Double-precision type

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:

- $2^{-8188} \leq R < 2^{8189}$

or approximately

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



a10777

Figure 6. 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.

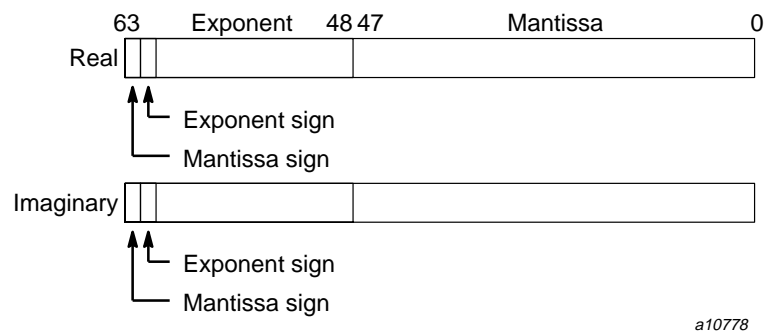


Figure 7. Single-precision complex type

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`.

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.

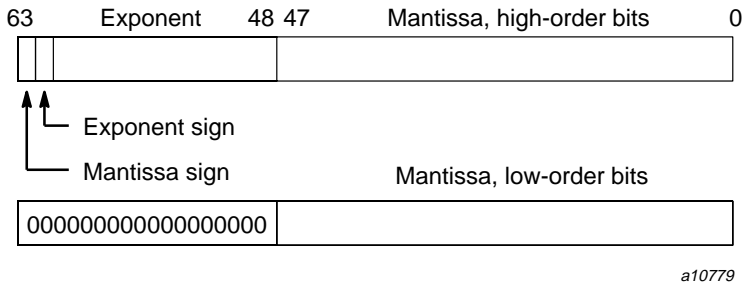


Figure 8. Double-precision complex type (real portion)

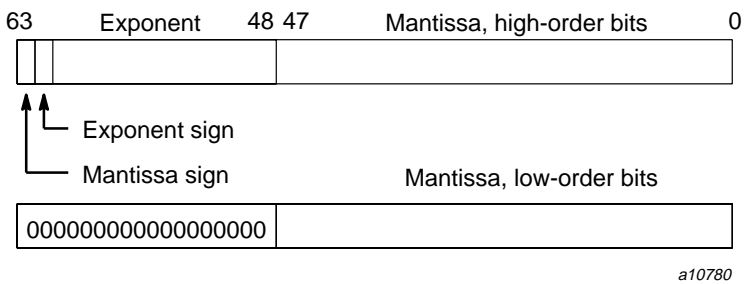


Figure 9. Double-precision complex type (imaginary portion)

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.

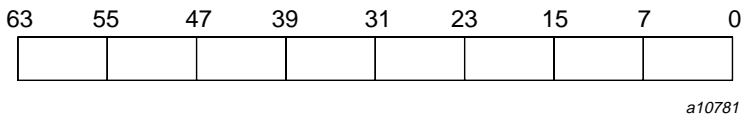


Figure 10. Character type

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 Cray Research 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: Cray Research 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 90 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.

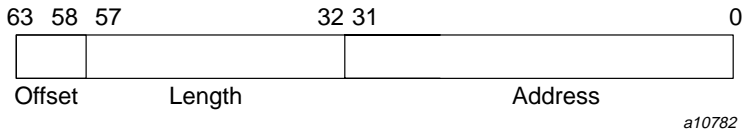


Figure 11. 32-bit addressing for UNICOS systems (except CRAY T90 systems)

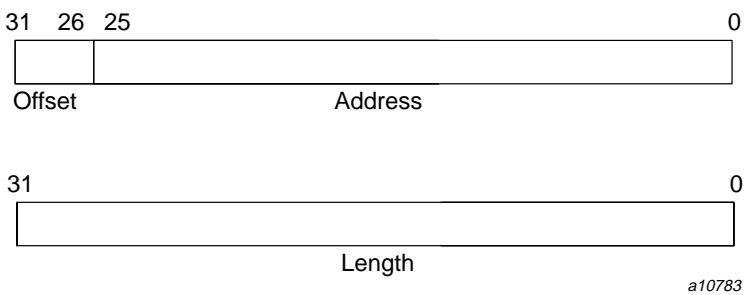


Figure 12. 32-bit addressing for CRAY T90 systems

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 $-10^2 < I < 10^2$

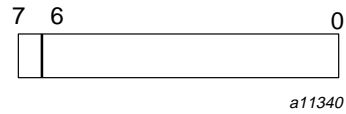


Figure 13. INTEGER(KIND=1) on IRIX systems

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 $-10^4 < I < 10^4$

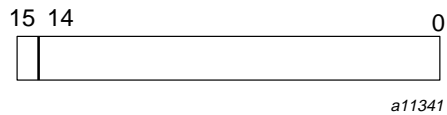


Figure 14. INTEGER(KIND=2) on IRIX systems

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 $-10^9 < I < 10^9$

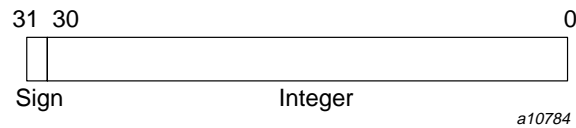


Figure 15. INTEGER(KIND=4) on IRIX systems

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 $-10^{18} < I < 10^{18}$

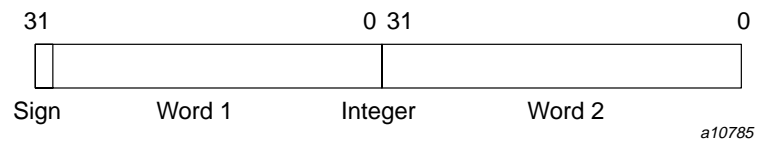


Figure 16. INTEGER(KIND=8) on IRIX systems

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: $-2^{-125} \leq I < 2^{128}$ or approximately $-10^{-38} \leq I < 10^{38}$

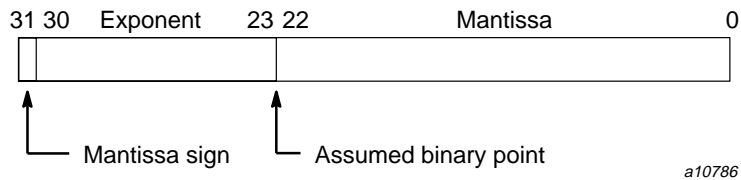


Figure 17. REAL(KIND=4) on IRIX systems

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
0.1	007563146315	3DCCCCCD
-0.1	027563146315	BDCCCCCD

```

010000010010000000000000000000000000
      ↑
    Bit 22
                                     a10787

```

Figure 18. 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 (1000010, 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: $-2^{-1021} \leq I < 2^{1024}$ or approximately $-10^{-308} \leq I < 10^{308}$

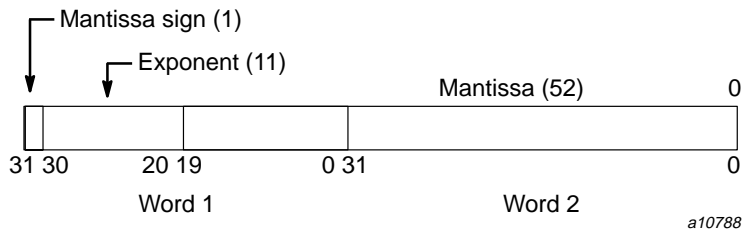


Figure 19. `REAL(KIND=8)` on IRIX systems

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: $-2^{-967} \leq I < 2^{1023}$ or approximately $-10^{-292} \leq I < 10^{308}$

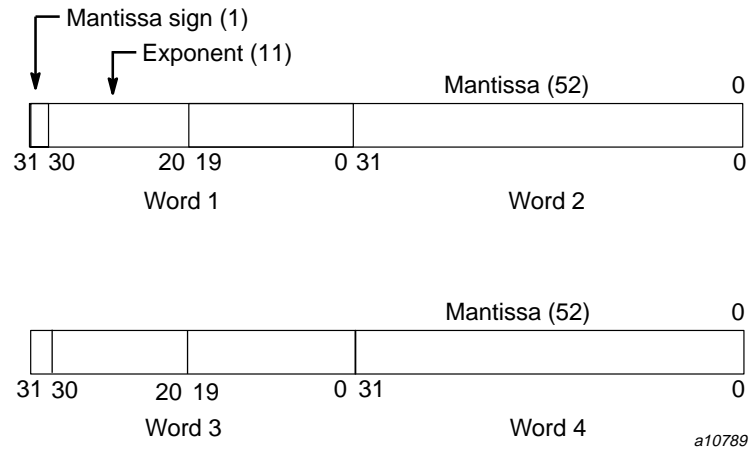


Figure 20. REAL(KIND=16) on IRIX systems

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: $-2^{-125} \leq I < 2^{128}$ or approximately $-10^{38} \leq I < 10^{38}$

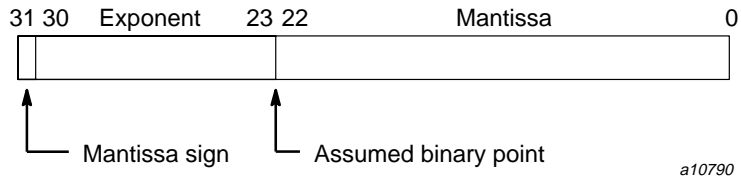


Figure 21. COMPLEX(KIND=4) on IRIX systems (real portion)

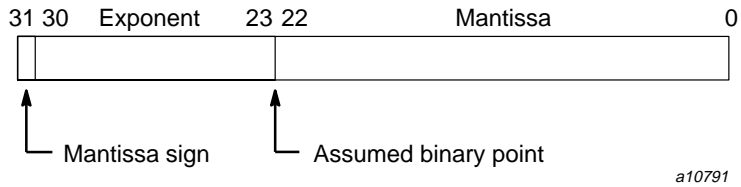


Figure 22. COMPLEX(KIND=4) on IRIX systems (imaginary portion)

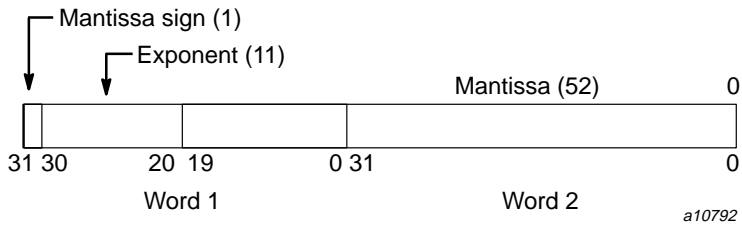
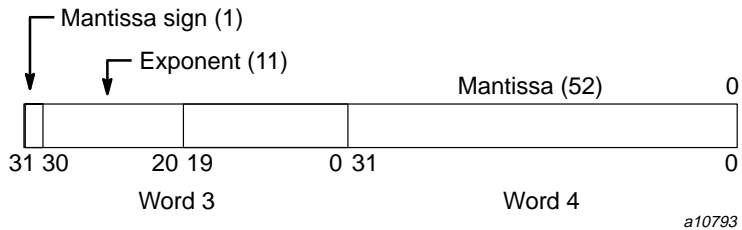
To declare an entity to be of single-precision, complex type, specify one of the following:

- 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: $-2^{-1021} \leq I < 2^{1024}$ or approximately $-10^{308} \leq I < 10^{308}$

Figure 23. `COMPLEX(KIND=8)` on IRIX systems (real portion)Figure 24. `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: $-2^{-967} \leq I < 2^{1023}$ or approximately $-10^{-292} \leq I < 10^{308}$

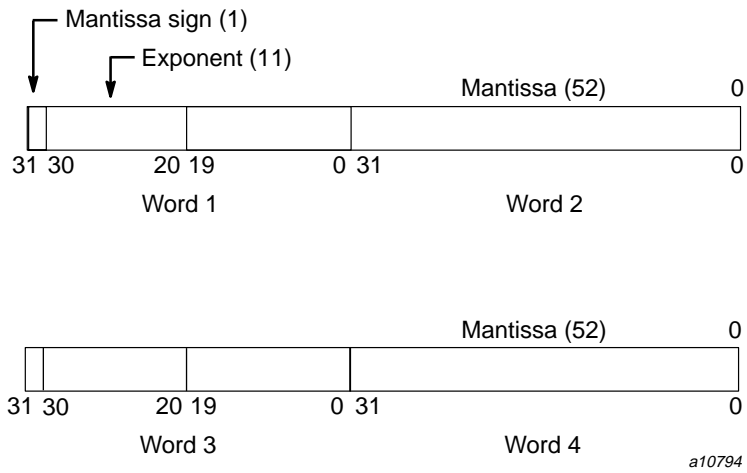


Figure 25. COMPLEX(KIND=16) on IRIX systems (real portion)

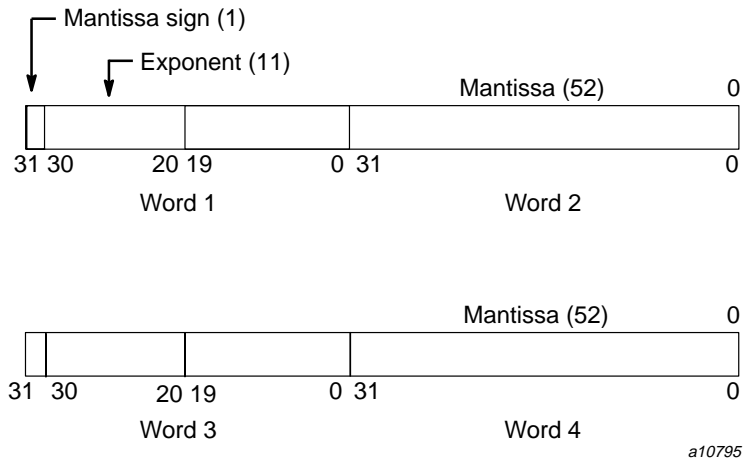


Figure 26. COMPLEX(KIND=16) on IRIX systems (imaginary portion)

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.

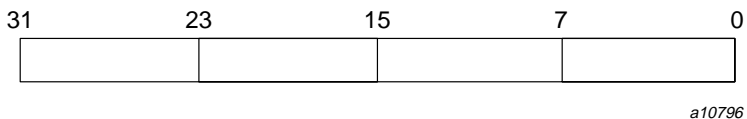


Figure 27. Character type

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: Cray Research and Silicon Graphics do 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 90 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.

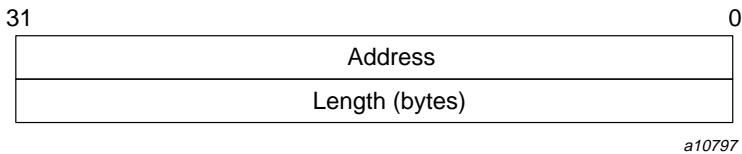


Figure 28. 32-bit addressing on IRIX systems

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 $-10^9 < I < 10^9$

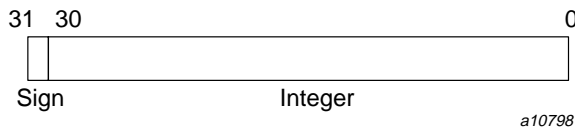


Figure 29. Integer `KIND=1`, `2`, or `4` on UNICOS/mk systems

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 $-10^{18} < I < 10^{18}$

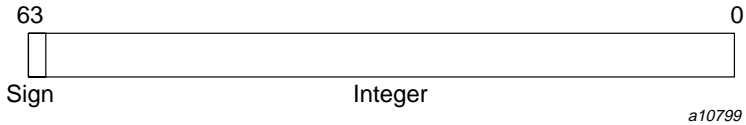


Figure 30. `INTEGER(KIND=8)` on UNICOS/mk systems

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: $-2^{-125} \leq I < 2^{128}$ or approximately $-10^{-38} \leq I < 10^{38}$

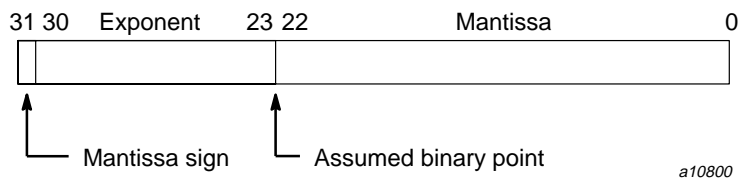


Figure 31. `REAL(KIND=4)` on UNICOS/mk systems

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 0111111_2 through 1111110_2 for a positive exponent, and 0000000_2 through 0111110_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:

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: $-2^{-125} \leq I < 2^{128}$ or approximately $-10^{-38} \leq I < 10^{38}$

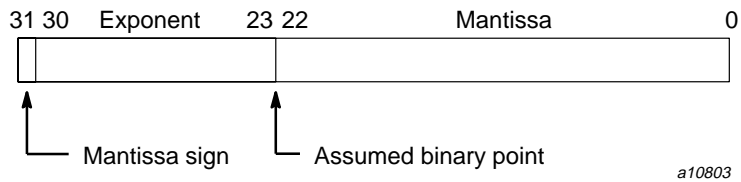


Figure 34. `COMPLEX(KIND=4)` on UNICOS/mk systems (real portion)

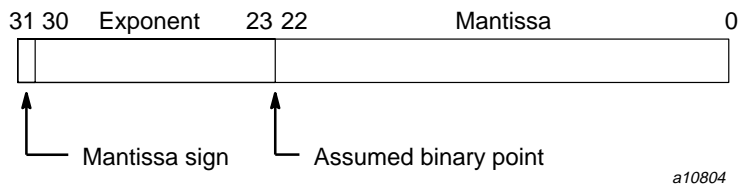


Figure 35. `COMPLEX(KIND=4)` on UNICOS/mk systems (imaginary portion)

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: $-2^{-1021} \leq I < 2^{1024}$ or approximately $-10^{-308} \leq I < 10^{308}$

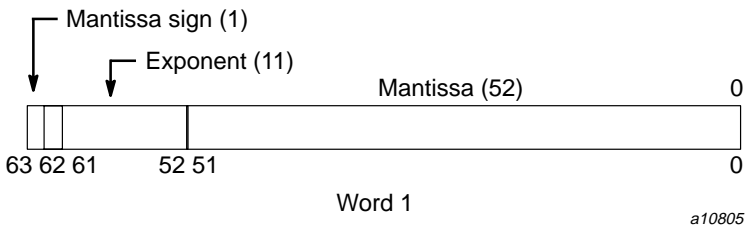


Figure 36. `COMPLEX(KIND=8)` on UNICOS/mk systems (real portion)

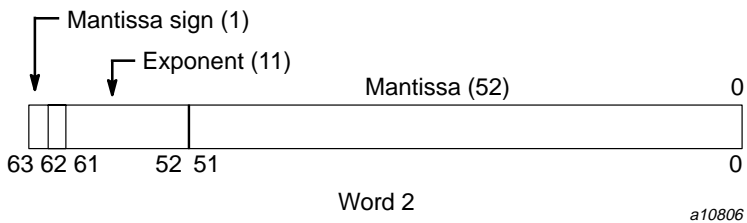


Figure 37. `COMPLEX(KIND=8)` on UNICOS/mk systems (imaginary portion)

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.

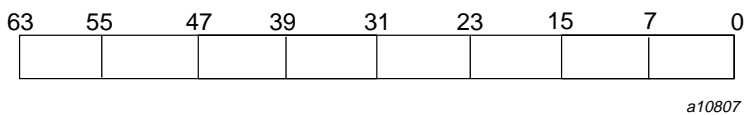


Figure 38. Character type

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: Cray Research 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 90 standard. Therefore, it is not good programming practice to use logical values as numbers or vice versa.

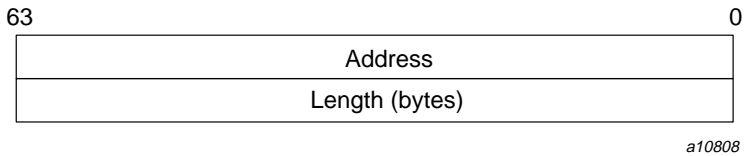
To declare an entity to be of logical type, you can specify one of the following:

- `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:



a10808

Figure 39. Cray character pointers on UNICOS/mk systems

5.4 Data representation for CRAY T90 systems that support IEEE floating-point arithmetic

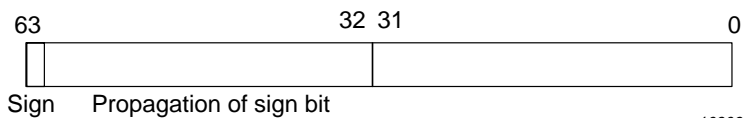
The following sections describe the representation of data on CRAY T90 systems that support IEEE floating-point arithmetic.

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 $-10^9 < I < 10^9$



a10809

Figure 40. 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 $-10^{15} < I < 10^{15}$.

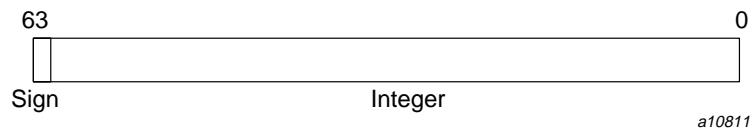


Figure 41. Default `INTEGER(KIND=8)` on CRAY T90 systems that support IEEE floating-point arithmetic

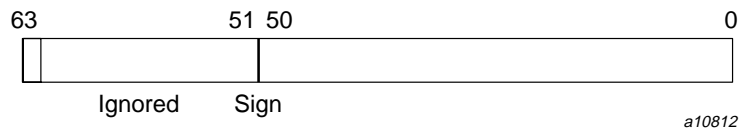


Figure 42. Fast operations with `INTEGER(KIND=8)` on CRAY T90 systems that support IEEE floating-point arithmetic

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: $-2^{-1021} \leq I < 2^{1024}$ or approximately $-10^{-308} \leq I < 10^{308}$

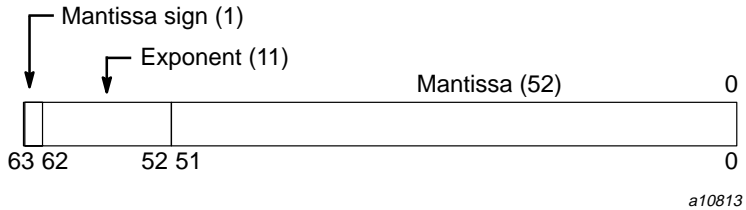


Figure 43. Real KIND=4 or 8 on CRAY T90 systems that support IEEE floating-point arithmetic

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 139. The information presented there for UNICOS/mk systems applies to CRAY T90 systems that support IEEE floating-point arithmetic.

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: $-2^{-16381} \leq I < 2^{16384}$ or approximately $-10^{-4932} \leq I < 10^{4932}$

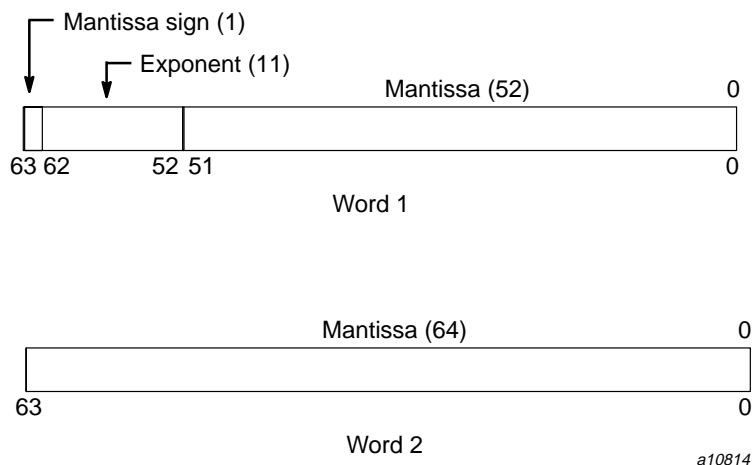


Figure 44. REAL(KIND=16) on CRAY T90 systems that support IEEE floating-point arithmetic

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: $-2^{-1021} \leq I < 2^{1024}$ or approximately $-10^{-308} \leq I < 10^{308}$

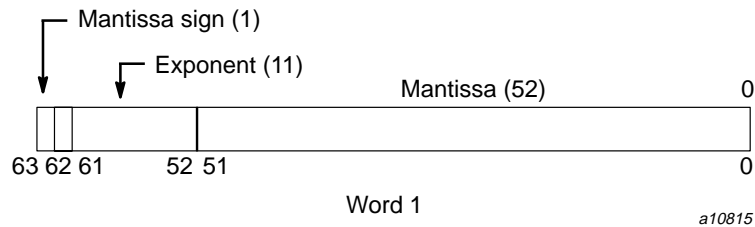


Figure 45. Complex `KIND=8` or `4` on CRAY T90 systems that support IEEE floating-point arithmetic (real portion)

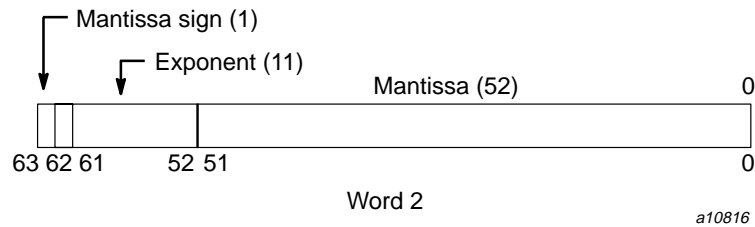


Figure 46. Complex `KIND=8` or `4` on CRAY T90 systems that support IEEE floating-point arithmetic (imaginary portion)

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.

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: $-2^{-16381} \leq I < 2^{16384}$ or approximately $-10^{-4932} \leq I < 10^{4932}$

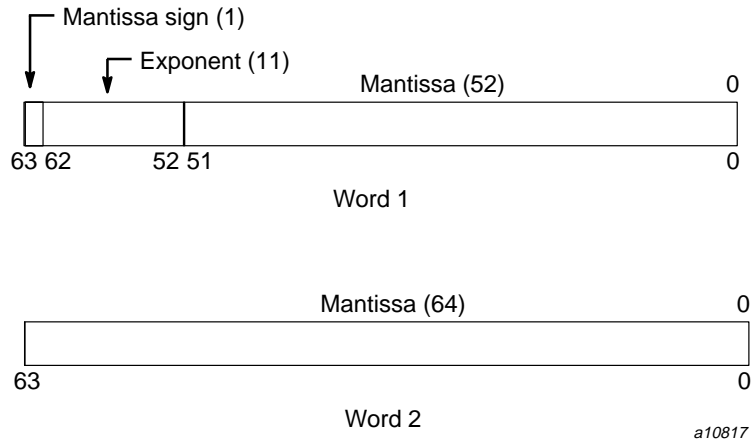


Figure 47. `COMPLEX(KIND=16)` on CRAY T90 systems that support IEEE floating-point arithmetic (real portion)

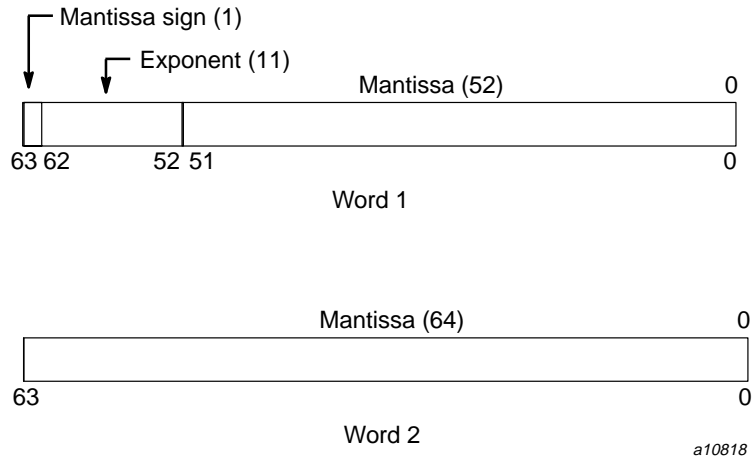


Figure 48. `COMPLEX(KIND=16)` on CRAY T90 systems that support IEEE floating-point arithmetic (imaginary portion)

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.

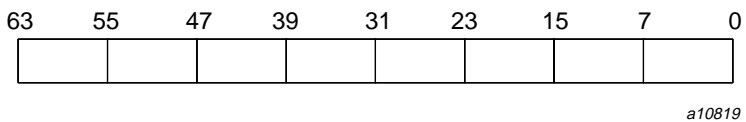


Figure 49. Character type

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: Cray Research 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 90 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.

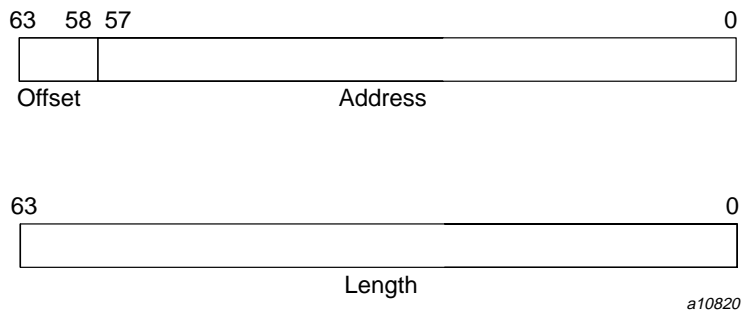


Figure 50. Cray character pointer for CRAY T90 systems that support IEEE floating-point arithmetic

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`, `-i 64`, `-s float64`, `-s default64`, 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 Cray Research word of 64 bits, for UNICOS and UNICOS/mk systems.
- 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` and `-s default64` options on the `f90(1)` command line change the number of bits in a numeric storage unit for UNICOS/mk systems. There is no longer a relationship between storage units after these command line options are used.

The following list shows the storage units for the default types on UNICOS systems:

<u>Type</u>	<u>Storage units</u>
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 90 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*, publication SR-3901, or the *MIPSpro 7 Fortran 90 Commands and Directives Reference Manual*, publication SR-3907. 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 51, page 158, 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.

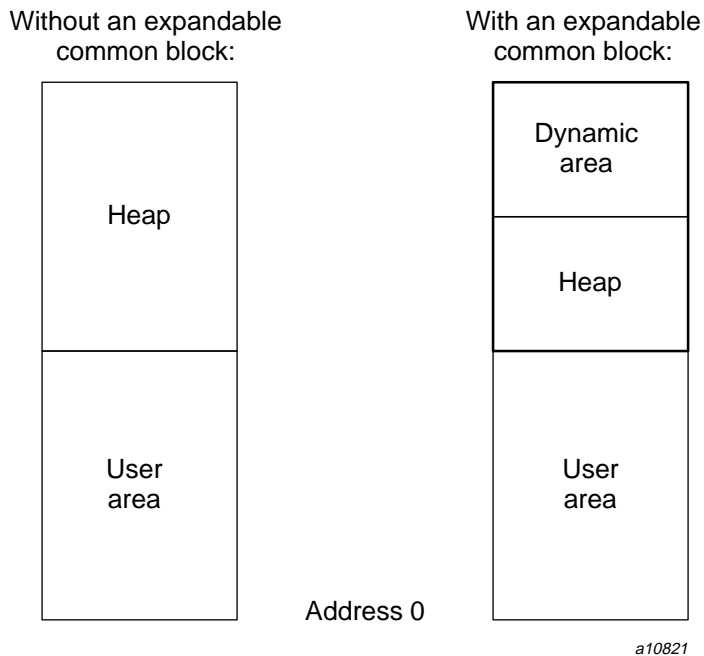


Figure 51. Memory use under UNICOS

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 90 allocatable arrays in a Fortran 90 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 Research pointers that will 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 Cray Research 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 Cray Research extensions supported in older Cray Research compilers. The outmoded features and their preferred alternatives are listed in Table 6.

Table 6. 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.
<code>[-b]x</code> edit descriptor	TL edit descriptor, <code>1X</code> .
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 (<code>END=</code>) or status specifier (<code>IOSTAT=</code>).
Initialization using long strings	Replace the numeric target with a character item. Replace a Hollerith constant with a character constant.
Type statements with <code>*n</code>	Standard type statements (<code>KIND=</code>).
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 (<code>KIND=</code>) statement and standard intrinsic functions. See Section 6.9, page 176, for more information.

Outmoded feature	Preferred alternative
Bitwise intrinsic functions	Standard intrinsic functions. See Section 6.10, page 177, 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.
I24MULT(3I) intrinsic function	Declare integers with KIND=1, KIND=2, or KIND=4.
INT24(3I) and LINT(3I) intrinsic functions	INT(3I) 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) and IRTC(3I) intrinsic functions	SYSTEM_CLOCK(3I) intrinsic subroutine.

Note: The following outmoded intrinsic functions will be removed in the CF90 4.0 release: I24MULT(3I) and LINT(3I).

6.1 Hollerith type

Hollerith data is a sequence of any characters capable of internal representation as specified in Table 5, page 83. 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 constant	Internal byte, beginning on bit:							
	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
'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 90 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:

<pre>ENCODE (<i>n</i>, <i>f</i>, <i>dest</i>) [<i>elist</i>]</pre>
--

<i>n</i>	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.
<i>f</i>	Format identifier. It cannot be an asterisk.
<i>dest</i>	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.
<i>elist</i>	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 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:

<pre>DECODE (<i>n</i>, <i>f</i>, <i>source</i>) [<i>dlist</i>]</pre>
--

- 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.

$*h_1 h_2 \dots h_n*$

$*$	Delimiter for a literal character string
h	Any ASCII character indicated by a C in Table 5, page 83 (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 *Rw* descriptor and the use of the *Aw* 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 *Aw* descriptor works with noncharacter list items containing character data in essentially the same way as described in the *Fortran Language Reference Manual, Volume 1*, publication SR-3902. The *Rw* descriptor works like *Aw* 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 *Aw* and *Rw* 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-OPQRSTUVWXYZ-xyzabcdefghijklmnopqrst uvwxyz
      ^^^^^
> ooooooAB-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 ] ( a1 [ -a2 ] [ , a1 [ -a2 ] ] ... )
      [ , type ... ] ...

[ type [ *n ] ] FUNCTION fun ([ d [ , d ] ... ] )

```

<i>type</i>	Can be INTEGER, REAL, DOUBLE PRECISION, COMPLEX, or LOGICAL.
<i>*n</i>	Data length as shown in Table 7, Table 8, page 173, and Table 9, page 173. Any other data length generates an error.
<i>v</i>	Name of a constant, variable, or array declarator.
<i>a_n</i>	A letter. A range of letters is denoted by the first and last letters of the range separated by a hyphen. A range (<i>a₁ – a_n</i>) has the same effect as a list of the letters (<i>a₁, a₂, ... a_n</i>).
<i>fun</i>	Name of the function subprogram.
<i>d</i>	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 7. Data length (UNICOS systems)

<i>type</i>	<i>n:</i>	*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 8. Data length (UNICOS/mk systems)

<i>type</i>	<i>n:</i>	*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 9. Data length (IRIX systems)

<i>type</i>	<i>n:</i>	*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:

<code>IF (e) s₁ , s₂</code>

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:

<pre>IF (<i>le</i>) <i>s_t</i> <i>s_f</i></pre>

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*, publication SR-3901. 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 ] . . .
```

<i>cb</i>	Task common block name.
<i>member_list</i>	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.

On UNICOS systems, to perform data initialization of `TASK COMMON` requires `SEGLDR` version 9.2 or higher. 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*, publication SR-3901.

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 90 standard. The Fortran 90 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 10 shows the CF90 double complex intrinsic functions and the preferred standard alternatives:

Table 10. 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 1's or leading 0's, are extensions to the Fortran 90 standard. Generally, these bitwise functions have equivalent Fortran 90 standard intrinsic procedures.

Table 11 shows the bitwise functions and, where possible, their equivalent Fortran 90 standard intrinsic procedures:

Table 11. Standard alternatives to CF90 and MIPSpro 7 Fortran 90 bitwise functions

Bitwise function	Fortran 90 standard alternative
AND(3M)	IAND(3I)
COMPL(3I)	NOT(3I)
CSMG(3I)	MERGE(3I)
CVMGM(3I)	MERGE(3I)
CVMGN(3I)	MERGE(3I)
CVMGP(3I)	MERGE(3I)
CVMGT(3I)	MERGE(3I)
CVMGZ(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)	ISHFT(3I), ISHFTC(3I)
SHIFTR(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 12, page 179, shows which data types can be used together in bitwise logical operations.

Table 12. 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 valid¹” 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_exprs*,

summation_exprs, 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:

<code>.NOT.</code>	1100		1100		1100		1100
<code>=0011</code>		<code>.AND.</code>	1010	<code>.OR.</code>	1010	<code>.XOR.</code>	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 `clld(1)` directives for load time optimization and activating library features are described in the *Application Programmer's I/O Guide*, publication SG-2168.

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.

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 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**.

A

ASCII character set, 83

B

Backus-Naur Form, 1
Bitwise logical expressions, 177
BNF syntax summary, 1

C

CF77 compiling system, 89
Character
 Hollerith, 162
Character data representation
 CRAY T90 (IEEE) systems, 151
 IRIX systems, 137
 UNICOS systems, 125
 UNICOS/mk systems, 144
Character set, 83
Complex data representation
 CRAY T90 (IEEE) systems, 148
 IRIX systems, 133
 UNICOS/mk systems, 142
Complex type (single precision), internal
 representation, 133
Constraints, 5
Cray character pointer data representation
 CRAY T90 (IEEE) systems, 152
 IRIX systems, 137
 UNICOS systems, 126
 UNICOS/mk systems, 145

D

Data

SR-3905 3.0.1

type

 Hollerith, 162
DATA statement, 174
DECODE statement, 167
Decremental features, 79
Defined externals, 181
Differences (from CF77 compiling system), 89
DOUBLE COMPLEX statement, 176
Double-precision complex data representation
 UNICOS systems, 124
Double-precision data representation
 UNICOS systems, 123
Dynamic memory allocation, 157

E

Edit descriptors
 outmoded, 169
ENCODE statement, 165
Extensions, 89
Externals (defined), 181

F

Formatted
 I/O and internal files, 165
Fortran
 keywords, 1

G

Global variables, 181

H

Hollerith type, 162

I

I/O

formatted, 165

IF statement, 174

Integer data representation

CRAY T90 (IEEE) systems, 145

IRIX systems, 127

UNICOS systems, 119

UNICOS/mk systems, 138

IRIX system data representation, 127

K

Keywords, 1

L

Logical data representation

CRAY T90 (IEEE) systems, 151

IRIX systems, 137

UNICOS systems, 126

UNICOS/mk systems, 144

M

Memory allocation, 157

O

Obsolescent features, 79

Outmoded features, 161

R

Real data representation

CRAY T90 (IEEE) systems, 146

IRIX systems, 129

UNICOS systems, 120

UNICOS/mk systems, 139

Real type, internal representation, 129

S

Single-precision complex data representation

UNICOS systems, 124

Stack storage, 155

Static storage, 155

Storage, 152

Syntax summary (in BNF), 1

T

TASKCOMMON statement, 175

U

UNICOS data representation, 119