Fortran Language Reference Manual, Volume 2

SR–3903 3.0.1

New Features

Fortran Language Reference Manual, Volume 2 SR–3903 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

Version	*Description*
2.0	November 1995. Original printing. This document supports the CF90 compiler release 2.0 running on Cray PVP systems, CRAY T3E systems, and SPARC systems. The implementation of features on CRAY T3E systems is deferred. This manual contains information that was previously contained in the *CF90 Fortran Language Reference Manual*, revision 1.0, publication SR–3902, and in the *Application Programmer's Library Reference Manual*, publication SR–2165.
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 7.2 release running on the IRIX operating system. The implementation of features on IRIX operating system platforms is deferred. This manual contains information that was previously contained in the *CF90 Fortran Language Reference Manual, Volume 1*, revision 2.0, publication SR–3902.
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

Page

Page

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:

Convention	Meaning
`command`	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.
variable	Italic typeface denotes variable entries and words or concepts being defined.
`user input`	This bold, fixed-space font denotes literal items that the user enters in interactive sessions. Output is shown in nonbold, fixed-space font.
[ ]	Brackets enclose optional portions of a command or directive line.
...	Ellipses indicate that a preceding element can be repeated.
EXT or EXTENSION	The EXT or EXTENSION notation indicates that the feature being described is an extension to the Fortran 90 standard.

*scalar_*	When *scalar_* is the first item in a syntax description, it indicates that the item is a scalar, not an array, value.
_name	When *_name* is part of a syntax definition, it indicates that the item is a name with no qualification. For example, the item must not have a subscript list, so `ARRAY` is a name, but `ARRAY(I)` is not.
(R*nnnn*)	Indicates that the Fortran 90 standard has rules regarding the characteristic of the language being discussed. All rules are numbered, and the numbered list appears in the *Fortran Language Reference Manual, Volume 3*, 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 `POINTER` refers to the Fortran 90 `POINTER` attribute.
Cray pointer	The term *Cray pointer* 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:

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

The syntax rule defines *goto_stmt* to be `GO TO` *label*, which describes the format of the `GO TO` statement. The description of the `GO TO` statement is incomplete until the definition of *label* is given. *label* is also a nonterminal symbol. A further search for *label* will result in a specification of *label* and thereby provide the complete statement definition. A *terminal* part of a syntax rule is one that does not need further definition. For example, `GO TO` is a terminal keyword and is a required part of the statement form. The complete BNF list appears in the *Fortran Language Reference Manual, Volume 3*, publication SR–3905

The following abbreviations are commonly used in naming nonterminal keywords:

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

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

add_op	**is**	+
	or	-

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

R525	*dimension_stmt*	**is**	`DIMENSION [ :: ]` *array_name* `(`*array_spec*`)`
			`[,` *array_name* `(`*array_spec*`) ] ...`

Reader comments

If you have comments about the technical accuracy, content, or organization of this document, please tell us. 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.

Input and Output (I/O) Processing [1]

Many programs need data to begin a calculation. After the calculation is completed, often the results need to be printed, displayed graphically, or saved for later use. During execution of a program, sometimes there is a large amount of data produced by one part of the program that needs to be saved for use by another part of the program, and the amount of data is too large to store in variables, such as arrays. Also, the editing capabilities of the data transfer statements for internal files can be used for processing character strings. Each of these tasks is accomplished using Fortran I/O statements described in this chapter.

The I/O statements are as follows:

`READ`

`PRINT`

`WRITE`

`OPEN`

`CLOSE`

`INQUIRE`

`BACKSPACE`

`ENDFILE`

`REWIND`

The `READ` statement is a *data transfer input statement* and provides a means for transferring data from an external media to internal storage or from an internal file to internal storage through a process called *reading*. The `WRITE` and `PRINT` statements are both *data transfer output statements* and provide a means for transferring data from internal storage to an external media or from internal storage to an internal file. This process is called *writing*. The `OPEN` and `CLOSE` statements are both *file connection statements*. The `INQUIRE` statement is a *file inquiry statement*. The `BACKSPACE`, `ENDFILE`, and `REWIND` statements are *file positioning statements*.

The first part of this chapter discusses terms and concepts you need to gain a thorough understanding of all of the I/O facilities. These include internal and external files, formatted and unformatted records, sequential and direct access methods for files, advancing and nonadvancing I/O for the sequential formatted access method, file and record positions, units, and file connection properties. Following the concepts are descriptions of the READ, WRITE, and PRINT data transfer statements and the effect of these statements when they are executed. A model for the execution of data transfer statements and a description of the possible error and other conditions created during the execution of data transfer statements are provided next. Following the model are the descriptions of the OPEN, CLOSE, and INQUIRE statements that establish, respectively, the connection properties between units and files; that disconnect units and files; and that permit inquiry about the state of the connection between a unit and file. Lastly, file position statements are specified, which include the BACKSPACE and REWIND statements, followed by the description of the ENDFILE statement that creates end-of-file records.

The chapter concludes with a summary of the terms, concepts, and statements used for input and output processing and some examples.

1.1 Records, files, access methods, and units

Collections of data are stored in files. The data in a file is organized into records. Fortran treats a record, for example, as a line on a computer terminal, a line on a listing, or a logical record on a magnetic tape or disk file. However, the general properties of files and records do not depend on how the properties are acquired or how the files and records are stored. This section discusses the properties of records and files, and the various kinds of data transfer.

A *file* is a sequence of records. The following section describes the properties of records.

Cray Research physical files and records are described in the *UNICOS File Formats and Special Files Reference Manual*, publication SR–2014. If you are using the MIPSpro 7 Fortran 90 compiler, you can assume that standard UNIX file formats are used.

1.1.1 Records

There are two kinds of records: data and end-of-file. A *data record* is a sequence of values.

The values in a data record can be represented in one of two ways: formatted or unformatted. *Formatted* data consists of characters that are viewable on some medium. For example, a record could contain the following four character values:

```
6
,
1
1
```

These are intended to represent the two numbers 6 and 11. In this case, the record might be represented schematically as shown in Figure 1.

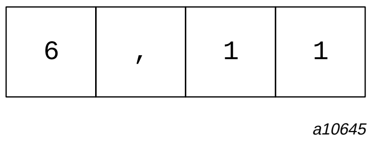


Figure 1. A formatted record with four character values

Unformatted data consists of values represented just as they are stored in computer memory. For example, if integers are stored using a binary representation, an unformatted record, consisting of two integer values, 6 and 11, might look like Figure 2.

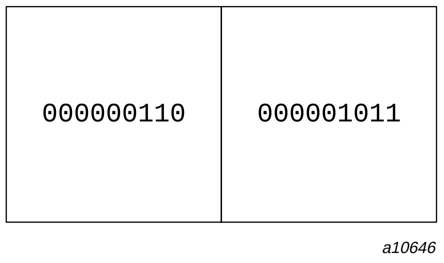


Figure 2. An unformatted record with two integer values

The values in a data record are either all formatted or all unformatted. A *formatted record* is one that contains only formatted data. It can be created by a person typing at a terminal or by a Fortran program that converts values stored internally into character strings that form readable representations of those values. When a program reads formatted data, the characters must be converted to the computer's internal representation of values. Even character values can be converted from one character representation in the record to another internal representation. The length of a formatted record is the number of characters in it; the length can be zero.

An *unformatted record* is one that contains only unformatted data. Unformatted records usually are created by running a Fortran program, but they can be created by other means. Unformatted data often requires less space on an external device. Also, it is usually faster to read and write because no conversion is required. However, it is not as suitable for reading by people and usually it is not suitable for transferring data from one computer to another because the internal representation of values is machine-dependent. The length of an unformatted data record depends on the number of values in it, but is measured in bytes; it can be zero. The length of an unformatted record that will be produced by a particular output list can be determined with the `INQUIRE` statement.

In general, a formatted record is read and written by a formatted data transfer I/O statement, and an unformatted record is read and written by an unformatted data transfer I/O statement.

Another kind of record is the *end-of-file record*; it has no value and has no length.

On UNICOS and UNICOS/mk systems, certain file structures allow more than one end-of-file record in a file. The COS blocking format is the default file structure for all UNICOS Fortran sequential unformatted files (except tape files). It can have multiple end-of-file records. For general information on records supported on UNICOS and UNICOS/mk systems, see the *Application Programmer's I/O Guide,* publication SG–2168.

For Cray Research structures other than COS blocked, and for structures on IRIX systems, an end-of-file record is always the last record of a file. It is used to mark the end of a file. To write it explicitly for files connected for sequential access, use the `ENDFILE` statement; to write it implicitly, use a file positioning statement (`REWIND` or `BACKSPACE` statement), close the file (`CLOSE` statement), open the file (`OPEN` statement), or use normal termination of a program.

ANSI/ISO: The Fortran 90 standard does not allow for multiple end-of-file records in a file.

1.1.2 Kinds of files

A file can contain a mixture of formatted and unformatted records.

ANSI/ISO: The Fortran 90 standard does not allow formatted and unformatted records to be mixed in a file.

The file can be named. The length of the file name and path name depend on the platform, as follows:

- UNICOS and UNICOS/mk systems allow 256 characters for a file name. This file name can include a directory name and 1023 characters for the full path name.
- IRIX systems allow 256 characters for a file name. The full path name is limited to 1023 characters.

A distinction is made between files that are located on an external device like a disk, and files in memory accessible to the program. These two kinds of files are as follows:

- External files
- Internal files

The use of these files is illustrated schematically in Figure 3.

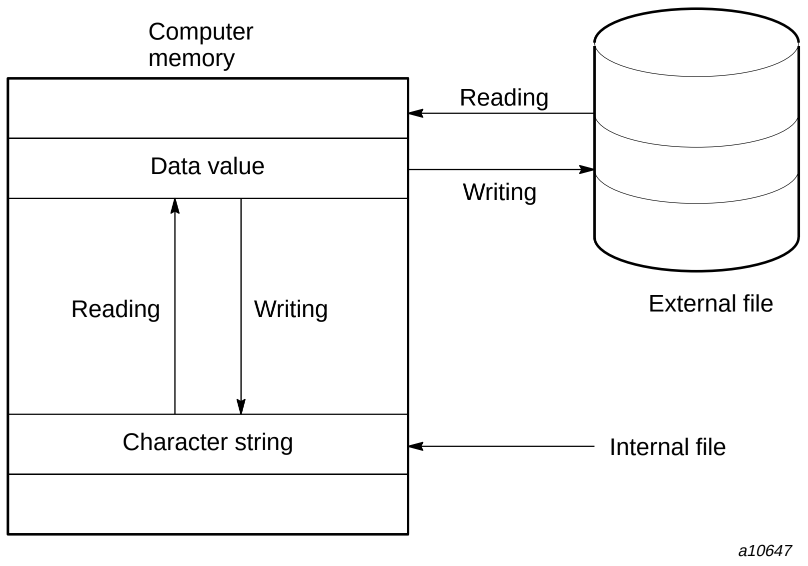


Figure 3. Internal and external files

1.1.2.1 External files

External files are located on external devices such as tapes, disks, or computer terminals. For each external file, there is a set of allowed access methods, a set of allowed forms, a set of allowed actions, and a set of allowed record lengths. These characteristics are determined by a combination of requests by the user of the file and by actions of the operating system. Each characteristic is discussed later in this section.

An external file connected to a unit has the position property. The file is positioned at the current record. In some cases, the file is positioned within the current record.

1.1.2.2 Internal files

The contents of internal files are stored as values of variables of type character. You can create the character values by using all the usual means of assigning character values, or you can create them with an output statement that specifies the variable as an internal file. Data transfer to and from internal files is described in Section 1.2.9, page 38. Such data transfer to and from an internal file must use formatted sequential access I/O statements, including list-directed data transfer, but not namelist data transfer.

File connection, file positioning, and file inquiry must not be used with internal files. If the variable representing the internal file is a scalar, the file has just one record; if the variable is an array, the file has one record for each element of the array. The order of the records is the order of the elements in the array. The length of each record is the length of one array element.

1.1.2.3 File existence

Certain files are made known to the system for any executing program, and these files are said to *exist* at the time the program begins executing. A file might not exist because it is not anywhere on the disks accessible to a system or because the user of the program is not authorized to access the file.

In addition to files that are made available to programs for input, output, and other special purposes, programs can create files needed during and after program execution. When the program creates a file, it is said to exist, even if no data has been written into it. A file no longer exists after it has been deleted. Any of the I/O statements can refer to files that exist for the program at that point during execution. Some of the I/O statements (`INQUIRE`, `OPEN`, `CLOSE`, `WRITE`, `PRINT`, `REWIND`, and `ENDFILE`) can refer to files that do not exist. A

WRITE or PRINT statement can create a file that does not exist and put data into that file, unless an error condition occurs.

An internal file always exists.

1.1.3 File position

Each file being processed by a program has a *position*. During the course of program execution, records are read or written, causing the file position to change. Also, there are other Fortran statements that cause the file position to change; an example is the BACKSPACE statement. The action produced by the I/O statements is described in terms of the file position, so it is important that file position be discussed in detail.

The *initial point* is the point just before the first record. The *terminal point* is the point just after the last record. If the file is empty, the initial point and the terminal point are the same. A file position can become indeterminate, in particular, when an error condition occurs. When the file position becomes indeterminate, the programmer cannot rely on the file being in any particular position.

A file can be positioned between records. In the example pictured in Figure 4, the file is positioned between records 2 and 3. In this case, record 2 is the *preceding record* and record 3 is the *next record.* Of course, if a file is positioned at its initial point, there is no preceding record, and there is no next record if it is positioned at its terminal point.

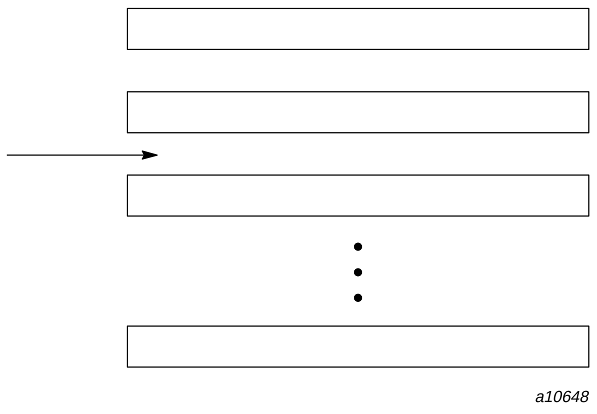


Figure 4. A file positioned between records

There may be a current record during execution of an I/O statement or after completion of a nonadvancing I/O statement as shown in Figure 5 where record 2 is the current record. If the file is positioned within a current record, the preceding record is the record immediately previous to the current record, unless the current record is also the initial record, in which case there is no preceding record. Similarly, the next record is the record immediately following the current record, unless the current record is also the final record, in which case there is no next record.

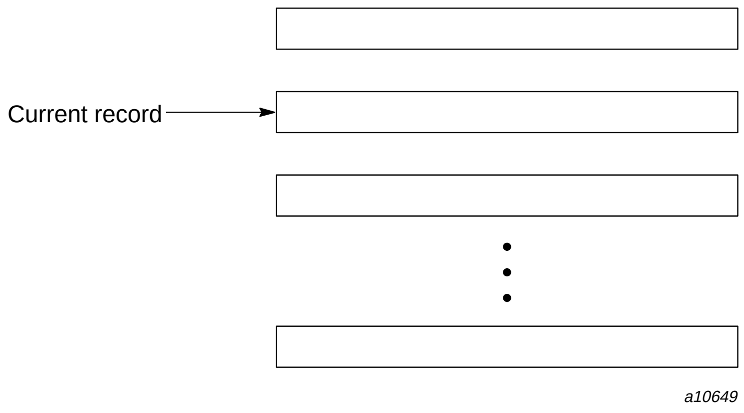


Figure 5. A file positioned with a current record

When there is a current record, the file is positioned at the initial point of the record, between values in the record, or at the terminal point of the record.

An internal file is always positioned at the beginning of a record just prior to data transfer.

Advancing I/O is record oriented; completion of such an operation always positions a file at the end of a record or between records, unless an error condition occurs. In contrast, nonadvancing I/O is character oriented; after reading and writing, the file can be positioned between characters within the current record.

While reading a file, the position of a nonadvancing file depends on whether success, data transfer, error, end-of-file, or end-of-record conditions occur while reading the file. The file position is indeterminate following an error condition when reading a file.

While writing a file, the position of a nonadvancing file depends on whether success, data transfer, or error conditions occur while writing the file. The file position is indeterminate following an error condition when writing a file.

When a nonadvancing input operation is performed, the file can be positioned after the last character of the file and before the logical or physical end-of-record. A subsequent nonadvancing input operation causes an end-of-record condition to occur, regardless of whether this record is the last record of the file. If another read operation is executed after the end-of-record condition occurs and the record is the last record of the file, an end-of-file condition occurs.

1.1.4 File access methods

There are two access file methods:

- Sequential access
- Direct access

Some files can be accessed by both methods; other files can be restricted to one access method or the other. For example, a magnetic tape can be accessed only sequentially. While each file is connected, it has a set of permissible access methods, which usually means that it can be accessed either sequentially or directly. However, a file must not be connected for both direct and sequential access simultaneously; that is, if a file is connected for direct access, it must be disconnected with a `CLOSE` statement and reconnected with an `OPEN` statement specifying sequential access before it can be referenced in a sequential access data transfer statement, and vice versa.

The actual file access method used to read or write the file is not a property of the file itself, but is indicated when the file is connected to a unit or when the file is created, if the file is preconnected. The same file can be accessed sequentially by a program, then disconnected, and then later accessed directly by the same program, if both types of access are permitted for the file.

1.1.4.1 Sequential access

Sequential access to the records in the file begins with the first record of the file and proceeds sequentially to the second record, and then to the next record, record-by-record. The records are accessed serially as they appear in the file. It is not possible to begin at some particular record within the file without reading down to that record in sequential order.

When a file is being accessed sequentially, the records are read and written sequentially. For example, if the records are written in any arbitrary order using direct access (see Section 1.1.4.2) and then read using sequential access, the records are read beginning with record number 1 of the file, regardless of when it was written.

1.1.4.2 Direct access

When a file is accessed directly, the records are selected by record number. Using this identification, the records can be read or written in any order. Therefore, it is possible to write record number 47 first, then number 13. Reading or writing such records is accomplished by direct access data transfer I/O statements. Either record can be written without first accessing the other.

A file can be accessed by using both the direct and sequential access methods (but not both at the same time). However, direct access reads are restricted to records that have been written, and direct access writes are restricted to files connected for direct access. If a file contains an end-of-file record and is connected for direct access, the end-of-file record is not considered part of the file. If the sequential access method is not an allowed access method between the unit and the file, the file must not contain an end-of-file record.

A file can be created to support sequential or direct access, using the `ACCESS` specifier on the `OPEN` statement. A file can support both kinds of access if it is closed and reopened with a different specification and if records are of equal length. If a file supports both methods of access, the first record accessed sequentially is the record numbered 1 for direct access; the second sequential record is record 2 for direct access, and so on.

While a file is connected for one kind of access (sequential or direct), only I/O statements using that kind of access can be used with the file.

1.1.5 Units

I/O statements refer to a particular file by providing an *I/O unit*. An I/O unit is either an external unit or an internal unit. An external unit is either a nonnegative integer or an asterisk (`*`). When an external file is a nonnegative integer, it is called an *external file unit*. Table 1, page 11, lists the unit numbers supported by the CF90 and MIPSpro 7 Fortran 90 compilers.

A unit number identifies one and only one external unit in all program units in a Fortran program at a time.

File positioning, file connection, and inquiry statements must use an external unit.

If you are running the CF90 compiler on a UNICOS or UNICOS/mk system, an *external_unit_name* can be used to designate an external file unit in a data transfer or file positioning statement; if you use an external name, it must be from 1 to 7 characters in length and be enclosed in apostrophes or quotation marks.

ANSI/ISO: The Fortran 90 standard does not describe the use of external names to designate a unit.

An internal unit must be a default character variable. The name of an internal file also is called a unit.

1.1.5.1 Unit existence

The collection of unit numbers that can be used in a program for external files consists of the integers from 0 through 299. The unit numbers that can be used are said to *exist*. I/O statements must refer to units that exist, except for those that close a file or inquire about a unit. The assignments for unit numbers 100, 101, and 102 cannot be changed, and these numbers cannot appear in an `OPEN` statement. Numbers 0, 5, and 6 are not equivalent to the asterisk, and they can be reassigned.

The following table shows the unit numbers available:

Table 1. CF90 and MIPSpro 7 Fortran 90 unit numbers

Unit number	Availability
0	Associated with the `stderr` file or available for user specification
1–4	Available for user specification
5	Associated with the `stdin` file or available for user specification
6	Associated with `stdout` or available for user specification
7–99	Available for user specification
100	Reserved for system use as the `stdin` file
101	Reserved for system use as the `stdout` file

Unit number	Availability
102	Reserved for system use as the `stderr` file
103–104	Reserved for system use
105–299	Available for user specification

1.1.5.2 Establishing a connection to a unit

In order to transfer data to or from an external file, the file must be connected to a unit. An internal file is always connected to the unit that is the name of the character variable. There are two ways to establish a connection between a unit and an external file:

- Execution of an `OPEN` statement in the executing program
- Preconnection by the operating system

Only one file can be connected to a unit at any given time and vice versa. If the unit is disconnected after its first use on a file, it can be reconnected later to another file or to the same file. A file that is not connected to a unit must not be used in any statement, except the `OPEN`, `CLOSE`, or `INQUIRE` statements.

Some units can be preconnected to files for each Fortran program by the operating system without any action necessary by the program. For example, units 5 and 6 are preconnected to files `stdin` and `stdout`, respectively. You also can use the `assign`(1) command to preconnect units. In either of these cases, the user program does not require an `OPEN` statement to connect a file for sequential access; it is preconnected.

Once a file has been disconnected, the only way to reference it is by its name, using an `OPEN` or `INQUIRE` statement. There is no means of referencing an unnamed file once it is disconnected.

1.2 Data transfer statements

When a unit is connected, either by preconnection or execution of an `OPEN` statement, data can be transferred by reading and writing to the file associated with the unit. The transfer can occur to or from internal or external files.

The *data transfer statements* are the `READ`, `WRITE`, and `PRINT` statements. This section presents the general form of the data transfer statements first, followed by the forms that specify the major uses of data transfer statements.

The `READ`, `WRITE`, and `PRINT` statements are defined as follows:

R909	*read_stmt*	**is**	`READ(`*io_control_spec_list*`)` [*input_item_list*]
		or	`READ` *format* [`,` *input_item_list*]
R910 EXT	*write_stmt*	**is**	`WRITE (`*io_control_spec_list*`)` [*output_item_list*]
		or	`WRITE` *format* [`,` *output_item_list*]
R911	*print_stmt*	**is**	`PRINT` *format* [`,` *output_item_list*]

ANSI/ISO: The Fortran 90 standard does not specify the `WRITE` *format* [`,` *output_item_list*] form of the `WRITE` statement.

The *io_control_spec_list*, the *input_item_list*, the *output_item_list*, and the *format* variables are described in subsequent sections. The format item is called a *format specifier.*

1.2.1 The I/O control specifiers

The I/O control specifiers are defined as follows:

R912	*io_control_spec*	**is**	[`UNIT =`] *io_unit*
		or	[`FMT =`] *format*
		or	[`NML =`] *namelist_group_name*
		or	`REC =` *scalar_int_expr*
		or	`IOSTAT =` *scalar_default_int_variable*
		or	`ERR =` *label*
		or	`END =` *label*
		or	`ADVANCE =` *scalar_char_expr*
		or	`SIZE =` *scalar_default_int_variable*
		or	`EOR =` *label*

The `UNIT=` specifier, with or without the keyword `UNIT`, is called a *unit specifier.*

The `FMT=` specifier, with or without the keyword `FMT`, is called a *format specifier.*

The `NML=` specifier, with or without the keyword `NML`, is called a *namelist specifier.*

The data transfer statement is called a *formatted I/O statement* if a format or namelist group name specifier is present; it is called an *unformatted I/O statement* if neither is present. If a namelist group name specifier is present, it is called a *namelist I/O statement*. It is called a *direct access I/O statement* if a `REC=` specifier is present; otherwise, it is called a *sequential access I/O statement*.

The I/O control specification list must contain a unit specifier and can contain any of the other I/O control specifiers (but none can appear more than once). An `FMT=` and an `NML=` specifier cannot both appear in the same list.

This section describes the form and effect of the control information specifiers that are used in the data transfer statements. The `NML=`, `ADVANCE=`, `END=`, `EOR=`, `REC=`, and `SIZE=` specifiers are each unique to one of the forms of the data transfer statements, whereas the other specifiers are used in more than one form. In particular, `NML=` is used in the namelist data transfer statement; the `ADVANCE=`, `EOR=`, and `SIZE=` specifiers are used in input data transfer statements to specify nonadvancing formatted sequential data transfer; and the `REC=` specifier is used for direct access data transfer.

1.2.1.1 `UNIT=` control specifier

The format of the `UNIT=` control specifier is as follows:

```
[ UNIT = ] io_unit
```

An *io_unit* is defined as follows:

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

ANSI/ISO: The Fortran 90 standard does not specify the use of a *unit_name.*

If you specify a *scalar_int_expr* for *io_unit*, it must be nonnegative and in the range of 0 through 299. Units 100 through 104 are reserved for the system.

Specifying an asterisk (`*`) for *io_unit* indicates a default external unit. It is the same unit number that would be defined if a `READ` or `PRINT` statement appeared without the unit number. The *io_unit* specified by an asterisk can be used only for formatted sequential access. The external unit used for a `READ` statement without a unit specifier or a `READ` statement with an asterisk unit specifier need not be the same as that used for a `PRINT` statement or `WRITE` statement with an asterisk unit specifier.

If you are using the CF90 compiler on a UNICOS or UNICOS/mk system, specifying an *external_unit_name* indicates a file name. The name can consist of 1 to 7 left-justified, blank-filled, or zero-filled ASCII characters. The use of uppercase and lowercase is significant for file names.

A unit specifier is required. A unit number identifies one and only one external unit in all program units in a Fortran program.

If the `UNIT` keyword is omitted, the I/O unit must be first. In this case, the keyword `FMT` or `NML` can be omitted from the format or namelist specifier. If the format or namelist specifier appears in the list, it must be the second specifier in the list.

1.2.1.2 `FMT=` control specifier

The format of the `FMT=` control specifier is as follows:

```
[ FMT = ] format
```

The *format* is defined as follows:

R913	*format*	**is**	*char_expr*
		or	*label*
		or	*
		or	*scalar_default_int_variable*

Specifying a *char_expr*, which provides the format specification in the form of a character string, indicates formatted I/O. The character expression must be a valid format specification. If the expression is an array, it is treated as if all elements of the array were concatenated together in array element order and must be a valid format specification.

Specifying a *label*, which provides the statement label of a FORMAT statement containing the format specification, indicates formatted I/O. The *label* must be the label of a FORMAT statement in the same scoping unit as the data transfer statement.

Specifying a *scalar_default_int_variable*, which provides an integer variable that has been assigned the label of a FORMAT statement, indicates formatted I/O. The label value must have been assigned to the integer variable using an ASSIGN statement and must be the label of a FORMAT statement in the same scoping unit as the data transfer statement. For information on the FORMAT statement, see Section 2.1.1, page 86.

Specifying an asterisk (*) indicates list-directed formatting.

The keyword FMT= can be omitted if the format specifier is the second specifier in the control information list and if the UNIT keyword was also omitted; otherwise, it is required.

If a format specifier is present, a namelist specifier (NML=) must not be present.

1.2.1.3 NML= control specifier

The format of the NML= control specifier is as follows:

```
[ NML = ] namelist_group_name
```

For *namelist_group_name*, specify the name of a namelist group declared in a NAMELIST statement. The namelist group name identifies the list of data objects to be transferred by the READ or WRITE statement with the NML= specifier.

If a namelist specifier is present, a format specifier must not be present.

1.2.1.4 ADVANCE= control specifier

The format of the ADVANCE= control specifier is as follows:

```
ADVANCE = scalar_char_expr
```

For *scalar_char_expr*, specify YES or NO. NO indicates nonadvancing formatted sequential data transfer. YES indicates advancing formatted sequential data transfer.

Trailing blanks in the character expression are ignored. The value of the specifier is without regard to case (upper or lower); that is, the value `no` is the same as `NO`.

If an `ADVANCE=` specifier appears in the control information list, the data transfer must be a formatted sequential data transfer statement connected to an external unit. List-directed and namelist I/O are not allowed.

If the `EOR=` or `SIZE=` specifier appears in the control information list, an `ADVANCE=` specifier must also appear with the value `NO`.

1.2.1.5 `END=` control specifier

The format of the `END=` control specifier is as follows:

```
END = label
```

If an end-of-file condition occurs and no error condition occurs during the execution of the `READ` statement, the program branches to the label in the `END=` specifier. The label must be the label of a branch target statement in the same scoping unit as the `READ` statement.

The `END=` specifier can appear only in a sequential access `READ` statement; note that the `END=` specifier must not appear in a `WRITE` statement.

If the file is an external file, it is positioned after the end-of-file record.

If an `IOSTAT=` specifier is present and end-of-file condition occurs, but no error condition occurs, the `IOSTAT` variable specified becomes defined with a negative value.

1.2.1.6 `EOR=` control specifier

The format of the `EOR=` control specifier is as follows:

```
EOR = label
```

The program branches to the labeled statement specified by the `EOR=` specifier if an end of record is encountered for a nonadvancing `READ` statement. The label must be the label of a branch target statement in the same scoping unit as the `READ` statement.

The EOR= specifier can appear only in a READ statement with an ADVANCE= specifier with a value of NO, that is, a nonadvancing READ statement.

If an end-of-record condition occurs and no error condition occurs during the execution of the READ statement:

- The file is positioned after the current record.
- The variable given in the IOSTAT= specifier, if present, becomes defined with a negative value.
- If the connection has been made with the PAD= specifier of YES, the record is padded with blanks to satisfy both the input item list and the corresponding data edit descriptor if the input item and/or data edit descriptor requires more characters than are provided in the record.
- The variable given in the SIZE= specifier, if present, becomes defined with an integer value equal to the number of characters read from the input record; however, blank padding characters inserted because the PAD= specifier is YES are not counted.
- Execution of the READ statement terminates, and the program branches to the label in the EOR= specifier.

1.2.1.7 ERR= control specifier

The format of the ERR= control specifier is as follows:

```
ERR = label
```

If an error condition occurs, the position of the file becomes indeterminate.

The program branches to the label in the ERR= specifier if an error occurs in a data transfer statement. The label must be the label of a branch target statement in the same scoping unit as the data transfer statement.

If an IOSTAT= specifier is also present and an error condition occurs, the IOSTAT variable specified becomes defined with a positive value.

If the data transfer statement is a READ statement, contains a SIZE= specifier, and an error condition occurs, then the variable specified by the SIZE= specifier becomes defined with an integer value equal to the number of characters read from the input record; however, blank padding characters inserted because the PAD= specifier is YES are not counted.

1.2.1.8 IOSTAT= control specifier

The format of the IOSTAT= control specifier is as follows:

```
IOSTAT = scalar_default_int_variable
```

An integer value is returned in *scalar_default_int_variable*. The meaning of the return value is as follows:

- If *scalar_default_int_variable* > 0, an error condition occurred.
- If *scalar_default_int_variable* = 0, no error, end-of-file, or end-of-record condition occurred.
- If *scalar_default_int_variable* < 0, an end-of-file or end-of-record condition occurred. The end-of-file negative value is not the same as the negative value indicating the end-of-record condition.

The IOSTAT= specifier applies to the execution of the data transfer statement itself.

The variable specified in the IOSTAT= specifier must not be the same as or associated with any entity in the I/O item list or in the namelist group or with the variable specified in the SIZE= specifier, if present.

If the variable specified in the IOSTAT= specifier is an array element, its subscript values must not be affected by the data transfer, by any implied-DO item or processing, or with the definition or evaluation of any other specifier in the control specifier list.

You can obtain an online explanation of an error identified by the IOSTAT variable value. To do this, join the returned value with its group name, shown in the following list, and use the resulting string as an argument to the explain(1) command. For example:

```
explain lib-5000
explain 90476
```

Table 2. Message number identifiers

Message number	Group name	Source of message
1 through 899	`sys`	Operating system
1000 through 1999	`lib`	Fortran library (UNICOS and UNICOS/mk systems)
4000 through 4999	`lib`	Fortran library (IRIX systems)
5000 through 5999	`lib`	Flexible File I/O (FFIO) library
90000 through 90500	None	Tape system

1.2.1.9 `REC=` control specifier

The format of the `REC=` control specifier is as follows:

```
REC = scalar_int_expr
```

The *scalar_int_expr* specifies an integer value that indicates the record number to be read or written.

The `REC=` specifier can appear only in a data transfer statement with a unit that is connected for direct access.

If the `REC=` specifier is present in a control information list, an `END=` or format specifier with an asterisk (for list-directed data transfer) must not be specified in the same control information list.

1.2.1.10 `SIZE=` control specifier

The format of the `SIZE=` control specifier is as follows:

```
SIZE = scalar_default_int_variable
```

The I/O system returns a nonnegative integer value in *scalar_default_int_variable* that indicates the number of characters read.

The `SIZE=` specifier applies to the execution of the `READ` statement itself and can appear only in a `READ` statement with an `ADVANCE=` specifier with the value `NO`.

Blanks inserted as padding characters when the `PAD=` specifier is `YES` for the connection are not counted.

The variable specified in the `SIZE=` specifier must not be the same as or associated with any entity in the I/O item list or in the namelist group or with the variable specified in the `IOSTAT=` specifier, if present.

If the variable specified in the `SIZE=` specifier is an array element, its subscript values must not be affected by the data transfer, by any implied-`DO` item or processing, or with the definition or evaluation of any other specifier in the control specifier list.

1.2.2 The I/O item list

The I/O item list consists basically of a list of variables in a `READ` statement and a list of expressions in a `WRITE` or `PRINT` statement. In addition, in any of these statements, the I/O item list can contain an I/O implied-`DO` list, containing a list of variables or expressions indexed by `DO` variables.

The components of the I/O lists are defined as follows:

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

The `DO` variable must be a scalar integer or real variable. If it is real, it must be default real or double-precision real; the use of real `DO` variables is considered obsolescent.

Each scalar numeric expression must be of type integer or real. If it is of type real, each must be of type default real or default double precision; the use of such real expressions is considered obsolescent. They need not be all of the same type nor of the type of the `DO` variable.

The DO variable must not be one of the input items in the implied-DO; it must not be associated with an input item either.

Two nested implied-DOs must not have the same (or associated) DO variables.

An implied-DO object, when it is part of an input item, must itself be an input item; that is, it must be a variable or an implied-DO object whose objects are ultimately variables. Similarly, an implied-DO object, when it is part of an output item, must itself be an output item; that is, it must be an expression or an implied-DO object whose objects are ultimately expressions.

For an I/O implied-DO, the loop is initialized, executed, and terminated in the same manner as for the DO construct. Its iteration count is established at the beginning of processing of the items that constitute the I/O implied-DO.

An array appearing without subscripts in an I/O list is treated the same as if all elements of the array appeared in array-element order. In the following example, assume UP is an array of shape (2,3):

```
READ *, UP
```

The preceding statement is the same as the following code:

```
READ *, UP(1,1), UP(2,1), UP(1,2),  &
        UP(2,2), UP(1,3), UP(2,3)
```

When a subscripted array is an input item, it is possible that when a value is transferred from the file to the variable, it might affect another part of the input item. This is not allowed. Consider the following READ statements, for example:

```
INTEGER  A(100), V(10)

READ *, A(A)
READ *, A(A(1):A(9))
! Suppose V’s elements are defined with
! values in the range 1 to 100.
READ *, A(V)
```

The first two READ statements are not valid because the data values read affect other parts of array A. The third READ statement is allowed as long as no two elements of V have the same value.

Assumed-size arrays cannot appear in I/O lists unless a subscript, a section subscript specifying an upper bound, or a vector subscript appears in the last dimension.

In formatted I/O, a structure is treated as if, in place of the structure, all components were listed in the order of the components in the derived-type definition. In the following statement, assume that `FIRECHIEF` is a structure of type `PERSON`:

```
READ *, FIRECHIEF
```

The preceding statement is the same as the following:

```
READ *, FIRECHIEF%AGE, FIRECHIEF%NAME
```

A pointer can be an I/O list item but it must be associated with a target at the time the data transfer statement is executed. For an input item, the data in the file is transferred to the associated target. For an output item, the target associated with the pointer must be defined, and the value of the target is transferred to the file.

On UNICOS and UNICOS/mk systems, the following types of items cannot be used in an I/O list: a structure with an ultimate component that is a pointer, auxiliary variables, or auxiliary arrays.

> **ANSI/ISO:** The Fortran 90 standard does not specify auxiliary variables or auxiliary arrays.

A constant or an expression with operators, parentheses, or function references cannot appear as an input list item, but can appear as an output list item. A function reference in an output list must not cause execution of another I/O statement on this same unit.

> **ANSI/ISO:** The Fortran 90 standard does not allow the execution of another I/O statement on any unit through a function reference in an I/O list. For more information on restrictions, see Section 1.9, page 80.

An input list item, or an entity associated with it, must not contain any portion of an established format specification.

On output, every entity whose value is to be written must be defined.

1.2.3 Explicitly formatted advancing sequential access data transfer

For formatted input and output, the file consists of characters. These characters are converted into representations suitable for storing in the computer memory during input and converted from an internal representation to characters on output. When a file is accessed sequentially, records are processed in the order in which they appear in the file.

The explicitly formatted, advancing, sequential access data transfer statements have the following formats:

```
READ ([ UNIT = ] io_unit, &
   [ FMT = ] format &
   [, IOSTAT = scalar_default_int_variable ] &
   [, ERR = label ] &
   [, END = label ] &
   [, ADVANCE = 'YES' ]) &
   [, input-item-list ]

READ format [, input_item_list ]

WRITE ([ UNIT = ] io_unit, &
   [ FMT = ] format &
   [, IOSTAT = scalar_default_int_variable ] &
   [, ERR = label ] &
   [, ADVANCE = 'YES' ]) &
   [ output_item_list ]

PRINT format [, output_item_list ]

WRITE format [, output_item_list ]
```

The I/O unit is a scalar integer expression with a nonnegative value, an asterisk (`*`), or a character literal constant (external name). All three I/O unit forms indicate that the unit is a formatted sequential access external unit.

ANSI/ISO: The Fortran 90 standard does not specify the use of a character literal constant (external name) as an I/O unit.

If an `ADVANCE=` specifier is present, the format must not be an asterisk (`*`).

When an advancing I/O statement is executed, reading or writing of data begins with the next character in the file. If a previous I/O statement was a nonadvancing statement, the next character transferred may be in the middle of a record, even if the statement being executed is an advancing statement. The essential difference between advancing and nonadvancing sequential data transfer is that an advancing I/O statement always leaves the file positioned at the end of the record.

During the data transfer, data is transferred with editing between the file and the entities specified by the I/O list. Format control is initiated and editing is performed as described in Chapter 2, page 83. The current record and possibly additional records are read or written.

For the data transfer, values can be transmitted to or from objects of intrinsic or derived types. In the latter case, the transmission is in the form of values of intrinsic types to or from the components of intrinsic types, which ultimately comprise these structured objects.

For input data transfer, the file must be positioned so that the record read is a formatted record or an end-of-file record.

For input data transfer, the input record is logically padded with blanks to satisfy an input list and format specification that requires more characters than the record contains, unless the `PAD=` specifier is specified as `NO` in the `OPEN` statement. If the `PAD=` specifier is `NO`, the input list and format specification must not require more characters from the record than the record contains.

For output data transfer, the output list and format specification must not specify more characters for a record than the record size; recall that the record size for an external file is specified by a `RECL=` specifier in the `OPEN` statement.

If the file is connected for formatted I/O, unformatted data transfer is prohibited.

Execution of an advancing sequential access data transfer statement terminates when one of the following occurs:

- Format processing encounters a data or colon edit descriptor, and there are no remaining elements in the input item list or output item list.
- On input, an end-of-file condition is encountered.
- An error condition is encountered.

Example of formatted `READ` statements are as follows:

```
! Assume that FMT_5 is a character string
! whose value is a valid format specification.
READ (5, 100, ERR=99, END=200)  &
               A, B, (C(I), I = 1, 40)
READ (9, IOSTAT=IEND, FMT=FMT_5) X, Y
READ (FMT="(5E20.0)", UNIT=5,  &
      ADVANCE="YES") (Y(I), I = 1, KK)
READ 100, X, Y
```

Examples of WRITE statements are as follows:

```
! Assume FMT_103 is a character string with a
! valid format specification.
WRITE (9, FMT_103, IOSTAT=IS, ERR=99) A, B, C, S
WRITE (FMT=105, ERR=9, UNIT=7)  X
WRITE (*, "(F10.5)")  X
PRINT "(A, E14.6)", " Y = ", Y
```

1.2.4 Unformatted sequential access

For unformatted sequential input and output, the file consists of values stored using an internal binary representation. This means that little or no conversion is required during input and output.

Unformatted sequential access data transfer statements are the READ and WRITE statements with no format specifier or namelist group name specifier. The formats are as follows:

```
READ ([ UNIT = ] scalar_int_expr &
   [, IOSTAT = scalar_default_int_variable ] &
   [, ERR = label ] &
   [, END = label ]) &
   [ input_item_list ]

WRITE ([ UNIT = ] scalar_int_expr &
   [, IOSTAT = scalar_default_int_variable ] &
   [, ERR = label ]) &
   [ output_item_list ]
```

Data is transferred without editing between the current record and the entities specified by the I/O list. Exactly one record is read or written.

Objects of intrinsic or derived types can be transferred through an unformatted data transfer statement.

For input data transfer, the file must be positioned so that the record read is an unformatted record or an end-of-file record.

For input data transfer, the number of values required by the input list must be less than or equal to the number of values in the record. Each value in the record must be of the same type as the corresponding entity in the input list, except that one complex value may correspond to two real list entities or two real values may correspond to one complex list entity. The type parameters of the corresponding entities must be the same; if two real values correspond to one complex entity or one complex value corresponds to two real entities, all three must have the same kind type parameter values. If an entity in the input list is of type character, the character entity must have the same length as the character value.

On output, if the file is connected for unformatted sequential access data transfer, the record is created with a length sufficient to hold the values from the output list. This length must be one of the set of allowed record lengths for the file and must not exceed the value specified in the `RECL=` specifier, if any, of the `OPEN` statement that established the connection.

Execution of an unformatted sequential access data transfer statement terminates when one of the following occurs:

- The input item list or output item list is exhausted.
- On input, an end-of-file condition is encountered.
- An error condition is encountered.

If the file is connected for unformatted I/O, formatted data transfer is prohibited.

The following are examples of unformatted sequential access `READ` statements :

```
READ (5, ERR=99, END=100)  A, B, (C(I), I = 1, 40)
READ (IOSTAT=IEND, UNIT=9) X, Y
READ (5) Y
```

The following are examples of unformatted sequential access `WRITE` statements:

```
WRITE (9, IOSTAT=IS, ERR=99) A, B, C, S
WRITE (ERR=99, UNIT=7) X
WRITE (9) X
```

If the access is sequential, the file is positioned at the beginning of the next record prior to data transfer and positioned at the end of the record when the I/O is finished, because nonadvancing unformatted I/O is not permitted.

1.2.5 Nonadvancing formatted sequential data transfer

Nonadvancing formatted sequential I/O provides the capability of reading or writing part of a record. It leaves the file positioned after the last character read or written, rather than skipping to the end of the record. Processing of nonadvancing input continues within a current record, until an end-of-record condition occurs. Nonadvancing input statements can read varying-length records and determine their lengths. Nonadvancing I/O is sometimes called *partial record* or *stream I/O*. It may be used only with explicitly formatted, external files connected for sequential access.

The formats for the nonadvancing I/O statements are as follows:

```
READ ([ UNIT = ] io_unit, &
   [ FMT = ] format, &
   ADVANCE = 'NO' &
   [, SIZE = scalar_default_int_variable ] &
   [, EOR = label ] &
   [, IOSTAT = scalar_default_int_variable ] &
   [, ERR = label ] &
   [, END = label ]) &
   [ input_item_list ]

WRITE ([ UNIT = ] io_unit, &
   [ FMT = ] format, &
    ADVANCE = 'NO' &
   [, IOSTAT = scalar_default_int_variable ] &
   [, ERR = label ]) &
   [ output_item_list ]
```

The I/O unit is a scalar integer expression with a nonnegative value, an asterisk (*), or a character literal constant (external name). The I/O unit forms indicate that the unit is a formatted sequential access external unit.

The format must not be an asterisk (*).

During the data transfer, data is transferred with editing between the file and the entities specified by the I/O list. Format control is initiated and editing is performed as described in Chapter 2, page 83. The current record and possibly additional records are read or written.

For the data transfer, values can be transmitted to or from objects of intrinsic or derived types. In the latter case, the transmission is in the form of values of intrinsic types to or from the components of intrinsic types, which ultimately comprise these structured objects.

For input data transfer, the file must be positioned at the beginning of, end of, or within a formatted record or at the beginning of an end-of-file record.

For input data transfer, the input record is logically padded with blanks to satisfy an input list and format specification that requires more characters than the record contains, unless the PAD= specifier is specified as NO in the OPEN statement. If the PAD= specifier is NO, the input list and format specification must not require more characters from the record than the record contains, unless either the EOR= or IOSTAT= specifier is present. In the exceptional cases during nonadvancing input, the actions and execution sequence are described in Section 1.4, page 47.

For output data transfer, the output list and format specification must not specify more characters for a record than the record size; recall that the record size for an external file may be specified by a RECL= specifier in the OPEN statement.

The variable in the SIZE= specifier is assigned the number of characters read on input. Blanks inserted as padding characters when the PAD= specifier is YES are not counted.

The program branches to the label given by the EOR= specifier, if an end-of-record condition is encountered during input. The label must be the label of a branch target statement in the same scoping unit as the data transfer statement.

Execution of a nonadvancing, formatted, sequential access data transfer statement terminates when one of the following occurs:

- Format processing encounters a data edit descriptor or colon edit descriptor, and there are no remaining elements in the input item list or output item list.
- On input, an end-of-file or end-of-record condition is encountered.
- An error condition is encountered.

Unformatted data transfer is prohibited.

In the following statements, assume that N has the value 7:

```
WRITE (*, '(A)', ADVANCE="NO") "The answer is "
PRINT '(I1)', N
```

The preceding statements produce the following single output record:

```
The answer is 7
```

In the following statements, assume that SSN is a rank-one array of size 9 with values (1, 2, 3, 0, 0, 9, 8, 8, 6):

```
DO I = 1, 3
  WRITE (*, ’(I1)’, ADVANCE="NO") SSN(I)
ENDDO
WRITE (*, ’("-")’, ADVANCE="NO")
DO I = 4, 5
  WRITE (*, ’(I1)’, ADVANCE="NO") SSN(I)
ENDDO
WRITE (*, ’(A1)’, ADVANCE="NO")  ’-’
DO I = 6, 9
  WRITE (*, ’(I1)’, ADVANCE="NO") SSN(I)
ENDDO
```

The preceding statements produce the following record:

```
123-00-9886
```

1.2.6 Direct access data transfer

In direct access data transfer, the records are selected by record number. The record number is a scalar integer expression whose value represents the record number to be read or written. The records can be written in any order, but all records must be of the length specified by the `RECL=` specifier in an `OPEN` statement.

If a file is connected using the direct access method, then nonadvancing, list-directed, and namelist I/O are prohibited. Also, an internal file must not be accessed using the direct access method.

It is not possible to delete a record using direct access. However, records may be rewritten so that a record can be erased by writing blanks into it.

1.2.6.1 Formatted direct access data transfer

For formatted input and output, the file consists of characters. These characters are converted into internal representation during input and are converted from an internal representation to characters on output.

Formatted direct access data transfer statements are `READ` and `WRITE` statements with a `REC=` specifier and a format specifier. The formats for formatted direct access data transfer statements are as follows:

```
READ ([ UNIT = ] scalar_int_expr, &
   [ FMT = ] format, &
   REC = scalar_int_expr &
   [, IOSTAT = scalar_default_int_variable ] &
   [, ERR = label ]) &
   [ input_item_list ]

WRITE ([ UNIT = ] scalar_int_expr, &
   [ FMT = ] format, &
   REC = scalar_int_expr &
   [, IOSTAT = scalar_default_int_variable ] &
   [, ERR = label ]) &
   [ output_item_list ]
```

The I/O unit is a scalar integer expression with a nonnegative value, an asterisk (*), or a character literal constant (external name). The I/O unit forms indicate that the unit is a formatted sequential access external unit.

The format must not be an asterisk (*).

Note that the above forms are intentionally structured so that the unit cannot be an internal file, and the END=, ADVANCE=, and namelist specifiers cannot appear.

On input, an attempt to read a record of a file connected for direct access that has not previously been written causes all entities specified by the input list to become undefined.

During the data transfer, data is transferred with editing between the file and the entities specified by the I/O list. Format control is initiated and editing is performed as described in Chapter 2, page 83. The current record and possibly additional records are read or written.

For the data transfer, values can be transmitted to or from objects of intrinsic or derived types. In the latter case, the transmission is in the form of values of intrinsic types to or from the components of intrinsic types, which ultimately comprise these structured objects.

For input data transfer, the file must be positioned so that the record read is a formatted record.

For input data transfer, the input record is logically padded with blanks to satisfy an input list and format specification that requires more characters than the record contains, unless the PAD= specifier was specified as NO in the OPEN statement. If the PAD= specifier is NO, the input list and format specification must not require more characters from the record than the record contains.

For output data transfer, the output list and format specification must not specify more characters for a record than the record size; recall that the record size for an external file is specified by a `RECL=` specifier in the `OPEN` statement.

If the format specification specifies another record (say, by the use of the slash edit descriptor), the record number is increased by one as each succeeding record is read or written by that I/O statement.

For output data transfer, if the number of characters specified by the output list and format do not fill a record, blank characters are added to fill the record.

Execution of a formatted direct access data transfer statement terminates when one of the following events occurs:

- Format processing encounters a data edit descriptor or colon edit descriptor, and there are no remaining elements in the input item list or output item list.
- An error condition is encountered.

If the file is connected for formatted I/O, unformatted data transfer is prohibited.

Examples of formatted direct access I/O statements are as follows. In these examples, assume that `FMT_X` is a character entity:

```
READ (7, FMT_X, REC=32, ERR=99) A
READ (IOSTAT=IO_ERR, REC=34,  &
      FMT=185, UNIT=10, ERR=99)  A, B, D
WRITE (8, "(2F15.5)", REC = N + 2) X, Y
```

1.2.6.2 Unformatted direct access data transfer

For unformatted input and output, the file consists of values stored using a representation that is close to or the same as that used in program memory. This means that little or no conversion is required during input and output.

Unformatted direct access data transfer statements are `READ` and `WRITE` statements with a `REC=` specifier and no format specifier. The formats for unformatted direct access data transfer statements are as follows:

```
READ ([ UNIT = ] scalar_int_expr, &
   REC = scalar_int_expr &
   [, IOSTAT = scalar_default_int_variable ] &
   [, ERR = label ]) &
   [ input_item_list ]

WRITE ([ UNIT = ] scalar_int_expr, &
   REC = scalar_int_expr &
   [, IOSTAT = scalar_default_int_variable ] &
   [, ERR = label ]) &
   [ output_item_list ]
```

The I/O unit is a scalar integer expression with a nonnegative value, an asterisk (*), or a character literal constant (external name). The I/O unit forms indicate that the unit is a formatted sequential access external unit.

Note that the preceding forms are intentionally structured so that the unit cannot be an internal file, and the `FMT=`, `END=`, `ADVANCE=`, and namelist specifiers cannot appear.

On input, an attempt to read a record of a file connected for direct access that has not previously been written causes all entities specified by the input list to become undefined.

The number of items in the input list must be less than or equal to the number of values in the input record.

Data is transferred without editing between the current record and the entities specified by the I/O list. Exactly one record is read or written.

Objects of intrinsic or derived types may be transferred.

For input data transfer, the file must be positioned so that the record read is an unformatted record.

For input data transfer, the number of values required by the input list must be less than or equal to the number of values in the record. Each value in the record must be of the same type as the corresponding entity in the input list, except that one complex value may correspond to two real list entities or two real values may correspond to one complex list entity. The type parameters of the corresponding entities must be the same; if two real values correspond to one complex entity or one complex value corresponds to two real entities, all three must have the same kind type parameter values. If an entity in the input list is of type character, the character entity must have the same length as the character value.

The output list must not specify more values than can fit into the record. If the file is connected for direct access and the values specified by the output list do not fill the record, the remainder of the record is undefined.

Execution of an unformatted direct access data transfer statement terminates when one of the following events occurs:

- The input item list or output item list is exhausted.
- An error condition is encountered.

If the file is connected for unformatted direct access I/O, formatted data transfer is prohibited.

The following are examples of unformatted direct access I/O statements:

```
READ (7, REC=32, ERR=99) A
READ (IOSTAT=MIS, REC=34, UNIT=10, ERR=99) A, B, D
WRITE (8, REC = N + 2) X, Y
```

1.2.7 List-directed data transfer

List-directed formatting can occur only with files connected for sequential access; however, the file can be an internal file. The I/O data transfer must be advancing. The records read and written are formatted.

List-directed data transfer statements are any data transfer statement for which the format specifier is an asterisk (*). The formats for the list-directed data transfer statements are as follows:

```
READ ([ UNIT = ] io_unit, &
   [ FMT = ] * &
   [, IOSTAT = scalar_default_int_variable ] &
   [, END = label ] &
   [, ERR = label ]) &
   [ input_item_list ]

READ * [, input-item-list ]

WRITE ([ UNIT = ] io_unit, &
   [ FMT = ] * &
   [, IOSTAT = scalar_default_int_variable ] &
   [, ERR = label ]) &
   [ output_item_list ]

PRINT * [, output_item_list ]

WRITE format [, output_item_list ]
```

Note that the preceding forms are intentionally structured so that the ADVANCE= and namelist specifiers cannot appear.

During the data transfer, data is transferred with editing between the file and the entities specified by the I/O list. The rules for formatting the data transferred are discussed in Section 2.11, page 120. The current record and, possibly, additional records are read or written.

For the data transfer, values can be transmitted to or from objects of intrinsic or derived types. In the latter case, the transmission is in the form of values of intrinsic types to or from the components of intrinsic types, which ultimately comprise these structured objects.

For input data transfer, the file must be positioned so that the record read is a formatted record or an end-of-file record.

For output data transfer, one or more records will be written. Additional records are written if the output list specifies more characters for a record than the record size.

If the file is connected for list-directed data transfer, unformatted data transfer is prohibited.

Execution of a list-directed data transfer statement terminates when one of the following conditions occurs:

- The input item list or the output item list is exhausted.
- On input, an end-of-file is encountered, or a slash (/) is encountered as a value separator.
- An error condition is encountered.

The following are examples of list-directed input and output statements:

```
READ (5, *, ERR=99, END=100) A, B, (C(I), I = 1, 40)
READ (FMT=*, UNIT=5) (Y(I), I = 1, KK)
READ *, X, Y
WRITE (*, *) X
PRINT *, " Y = ", Y
```

1.2.8 Namelist data transfer

Namelist I/O uses a group name for a list of variables that are transferred. Before the group name can be used in the transfer, the list of variables must be declared in a NAMELIST statement, which is a specification statement. Using the namelist group name eliminates the need to specify the list of variables in the sequence for a namelist data transfer. The formatting of the input or output record is not specified in the program; it is determined by the contents of the record itself or the items in the namelist group. Conversion to and from characters is implicit for each variable in the list.

1.2.8.1 Form of a namelist group declaration

All namelist I/O data transfer statements use a namelist group name, which must be declared. The format of the namelist group name declaration is as follows:

```
NAMELIST / namelist_group_name / variable_name [, variable_name] ...
   [ [ , ] / namelist_group_name / variable_name [, variable_name ] ... ]
   ...
```

Examples:

```
NAMELIST /GOAL/ G, K, R
NAMELIST /XLIST/ A, B /YLIST/ Y, YY, YU
```

1.2.8.2 Forms of namelist I/O statements

Namelist input and output data transfer statements are READ and WRITE statements with a namelist specifier. The formats for namelist data transfer statements are as follows:

```
READ ([ UNIT = ] io_unit, &
   [ NML = ] namelist_group_name &
   [, IOSTAT = scalar_default_int_variable ] &
   [, END = label ] &
   [, ERR = label ])

WRITE ([ UNIT = ] io_unit, &
   [ NML = ] namelist_group_name &
   [, IOSTAT = scalar_default_int_variable ] &
   [, ERR = label ])
```

Note that the preceding forms do not contain the REC=, FMT=, and ADVANCE= specifiers. There must be no input or output item list.

The I/O unit must not be an internal file.

During namelist data transfer, data is transferred with editing between the file and the entities specified by the namelist group name. Format control is initiated and editing is performed as described in Section 2.12, page 126. The current record and possibly additional records are read or written.

For the data transfer, values can be transmitted to or from objects of intrinsic or derived types. In the latter case, the transmission is in the form of values of intrinsic types to or from the components of intrinsic types, which ultimately comprise these structured objects.

For namelist input data transfer, the file must be positioned so that the record read is a formatted record or an end-of-file record.

For namelist output data transfer, one or more records will be written. Additional records are written if the namelist specifies more characters for a record than the record size; recall that the record size for an external file can be specified by a RECL= specifier in the OPEN statement.

If an entity appears more than once within the input record for a namelist input data transfer, the last value is the one that is used.

For namelist input data transfer, all values following a *name*= part within the input record are transmitted before processing any subsequent entity within the namelist input record.

Execution of a namelist data transfer statement terminates when one of the following events occurs:

- On input, an end-of-file is encountered, or a slash (/) is encountered as a value separator.
- On input, the end of the namelist input record is reached and a name-value subsequence has been processed for every item in the namelist group object list.
- On output, the namelist group object list is exhausted.
- An error condition is encountered.

If the file is connected for namelist data transfer, unformatted data transfer is prohibited.

Examples of namelist data transfer statements are as follows:

```
READ (NML=NAME_LIST_23, IOSTAT=KN,  UNIT=5)
WRITE (6, NAME_LIST_23, ERR=99)
```

1.2.9 Data transfer on internal files

Transferring data from an internal representation to characters or from characters back to internal representation can be done between two variables in an executing program. A formatted sequential access input or output statement, including list-directed formatting, is used for the transfer. The format is used to interpret the characters. The internal file and the internal unit are the same character variable.

With this feature, it is possible to read in a string of characters without knowing its exact format, examine the string, and then interpret it according to its contents.

Formatted sequential access data transfer statements on an internal file have the following formats:

```
READ ([ UNIT = ] char_variable, &
   [ FMT = ] format &
   [, IOSTAT = scalar_default_int_variable ] &
   [, ERR = label ] &
   [, END = label ]) &
   [ input_item_list]

WRITE ([ UNIT = ] char_variable, &
   [ FMT = ] format &
   [, IOSTAT = scalar_default_int_variable ] &
   [, ERR = label ]) &
   [ output_item_list ]
```

Examples of data transfer on internal files are:

```
READ (CHAR_124, 100, IOSTAT=ERR) MARY, X, J, NAME
WRITE (FMT=*, UNIT=CHAR_VAR) X
```

The unit must be a character variable that is not an array section with a vector subscript.

Each record of an internal file is a scalar character variable.

If the character variable is an array or an array section, each element of the array or section is a scalar character variable and thus a record. The order of the records is array element order. The length, which must be the same for each record, is the length of one array element.

If the character variable is an array or part (component, element, section, or substring) of an array that has the ALLOCATABLE attribute, the variable must be allocated before it is used as an internal file. It must be defined if it is used as an internal file in a READ statement.

If the character variable is a pointer, it must be associated with a target. The target must be defined if it is used as an internal file in a READ statement.

During data transfer, data is transferred with editing between the internal file and the entities specified by the I/O list. Format control is initiated and editing is performed as described in Chapter 2, page 83. The current record and possibly additional records are read or written.

For the data transfer, values can be transmitted to or from objects of intrinsic or derived types. In the latter case, the transmission is in the form of values of intrinsic types to or from the components of intrinsic types, which ultimately comprise these structured objects.

For output data transfer, the output list and format specification must not specify more characters for a record than the record size. Recall that the record size for an internal file is the length of the scalar character variable or an element in a character array that represents the internal file. The format specification must not be part of the internal file or associated with the internal file or part of it.

If the number of characters written is less than the length of the record, the remaining characters are set to blank.

The records in an internal file are defined when the record is written. An I/O list item must not be in the internal file or associated with the internal file. An internal file also can be defined by a character assignment statement, or some other means, or can be used in expressions in other statements. For example, an array element may be given a value with a `WRITE` statement and then used in an expression on the right-hand side of an assignment statement.

To read a record in an internal file, the character variable comprising the record must be defined.

Before a data transfer occurs, an internal file is positioned at the beginning of the first record (that is, before the first character, if a scalar, and before the first character of the first element, if an array). This record becomes the current record.

Only formatted sequential access, including list-directed formatting, is permitted on internal files. Namelist formatting is prohibited.

On input, an end-of-file condition occurs when there is an attempt to read beyond the last record of the internal file.

During input processing, all nonleading blanks in numeric fields are treated as if they were removed, right justifying all characters in the field (as if a `BN` edit descriptor were in effect). In addition, records are blank padded when an end of record is encountered before all of the input items are read (as if `PAD=YES` were in effect).

For list-directed output, character values are not delimited.

File connection, positioning, and inquiry must not be used with internal files.

Execution of a data transfer statement on an internal file terminates when one of the following events occurs:

- Format processing encounters a data or colon edit descriptor, and there are no remaining elements in the input item list or output item list.

- If list-directed processing is specified, the input item list or the output item list is exhausted; or on input, a slash (`/`) is encountered as a value separator.
- On input, an end-of-file condition is encountered.
- An error condition is encountered.

1.2.10 Printing formatted records

Sometimes output records are sent to a device that interprets the first character of the record as a control character. This is usually the case with line printers. If a formatted record is transferred to such a device, the first character of the record is not printed, but instead is used to control vertical spacing. The remaining characters of the record, if any, are printed on one line beginning at the left margin. This transfer of information is called *printing*.

The first character of such a record must be of character type and determines vertical spacing as specified in Table 3.

Table 3. Interpretation of the first character for printing control

Character	Vertical spacing before printing
Blank	One line (single spacing)
0	Two lines (double spacing)
1	To first line of next page (begin new page)
+	No advance (no spacing - printing on top of previous line)

If there are no characters in the record, a blank line is printed. If the first character is not a blank, `0`, `1`, or `+`, the character is treated as a blank.

The `PRINT` statement does not imply that printing actually will occur on a printer, and the `WRITE` statement does not imply that printing will not occur. Whether printing occurs depends on the device connected to the unit number.

1.2.11 `BUFFER IN` and `BUFFER OUT` statements (EXTENSION)

You can use the `BUFFER IN` and `BUFFER OUT` statements to transfer data.

ANSI/ISO: The Fortran 90 standard does not describe the `BUFFER IN` or `BUFFER OUT` statements.

On UNICOS and UNICOS/mk systems, data can be transferred while allowing the subsequent execution sequence to proceed concurrently. This is called asynchronous I/O. Asynchronous I/O requires the use of pure data (unblocked) files, as discussed in the *Application Programmer's I/O Guide*, publication SG–2168. `BUFFER IN` and `BUFFER OUT` operations can proceed simultaneously on several units or files. On UNICOS/mk systems, asynchronous I/O is allowed when the operating system permits.

Note: Asynchronous I/O is not available on IRIX systems. The I/O is mapped to synchronous I/O.

`BUFFER IN` is for reading, and `BUFFER OUT` is for writing. A `BUFFER IN` or `BUFFER OUT` operation includes only data from a single array or a single common block.

Either statement initiates a data transfer between a specified file or unit (at the current record) and memory. If the unit or file is completing an operation initiated by any earlier `BUFFER IN` or `BUFFER OUT` statement, the current `BUFFER IN` or `BUFFER OUT` statement suspends the execution sequence until the earlier operation is complete. When the unit's preceding operation terminates, execution of the `BUFFER IN` or `BUFFER OUT` statement completes as if no delay had occurred.

You can use the `UNIT`(3I) or `LENGTH`(3I) intrinsic procedures to delay the execution sequence until the `BUFFER IN` or `BUFFER OUT` operation is complete. These functions can also return information about the I/O operation at its termination. On UNICOS and UNICOS/mk systems, you can use the `SETPOS`(3) library routine with `BUFFER IN` and `BUFFER OUT` for random positioning.

The general format of the `BUFFER IN` and `BUFFER OUT` statements follows:

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

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

The `BUFFER IN` and `BUFFER OUT` statements are defined as follows.

```
BUFFER IN (io_unit, mode) (start_loc, end_loc)

BUFFER OUT (io_unit, mode) (start_loc, end_loc)
```

io_unit	An identifier that specifies a unit or a file name. The I/O unit is a scalar integer expression with a nonnegative value, an asterisk (`*`), or a character literal constant (external name). The I/O unit forms indicate that the unit is a formatted sequential access external unit.
mode	Mode identifier. This integer expression controls the record position following the data transfer. The mode identifier is ignored on files that do not contain records; only full record processing is available.
start_loc, *end_loc*	Symbolic names of the variables, arrays, or array elements that mark the beginning and ending locations of the `BUFFER IN` or `BUFFER OUT` operation. These names must be either elements of a single array (or equivalenced to an array) or members of the same common block. If *start_loc* or *end_loc* is of type character, then both must be of type character. If *start_loc* and *end_loc* are noncharacter, then the item length of each must be equal. For example, if the internal length of the data type of *start_loc* is 64 bits, the internal length of the data type of *end_loc* must be 64 bits. To ensure that the size of *start_loc* and *end_loc* are the same, use the same data type for both.

The mode identifier, *mode*, controls the position of the record at unit *io_unit* after the data transfer is complete. The values of *mode* have the following effects:

- Specifying *mode* ≥ `0` causes full record processing. File and record positioning works as with conventional I/O. The record position following such a transfer is always between the current record (the record with which the transfer occurred) and the next record. Specifying `BUFFER OUT` with *mode* ≥ `0` ends a series of partial-record transfers.
- Specifying *mode* < `0` causes partial record processing. In `BUFFER IN`, the record is positioned to transfer its (n +1)th word if the *n*th word was the last transferred. In `BUFFER OUT`, the record is left positioned to receive additional words.

The amount of data to be transferred is specified in words without regard to types or formats. However, the data type of *end_loc* affects the exact ending location of a transfer. If *end_loc* is of a multiple-word data type, the location of the last word in its multiple-word form of representation marks the ending location of the data transfer.

`BUFFER OUT` with *start_loc* `=` *end_loc* + `1` and *mode* ≥ `0` causes a zero-word transfer and concludes the record being created. Except for terminating a partial record, *start_loc* following *end_loc* in a storage sequence causes a run-time error.

Example:

```
  PROGRAM XFR
  DIMENSION A(1000), B(2,10,100), C(500)
  ...
  BUFFER IN(32,0) (A(1),A(1000))
  ...
  DO 9 J=1,100
    B(1,1,J) = B(1,1,J) + B(2,1,J)
9 CONTINUE
  BUFFER IN(32,0) (C(1),C(500))
  BUFFER OUT(22,0) (A(1),A(1000))
  ...
  END
```

The first `BUFFER IN` statement in this example initiates a transfer of 1000 words from unit 32. If asynchronous I/O is available, processing unrelated to that transfer proceeds. When this is complete, a second `BUFFER IN` is encountered, which causes a delay in the execution sequence until the last of the 1000 words is received. A transfer of another 500 words is initiated from unit 32 as the execution sequence continues. `BUFFER OUT` begins a transfer of the first 1000 words to unit 22. In all cases *mode* = `0`, indicating full record processing.

1.3 Execution model for data transfer statements

When a data transfer statement is executed, the following steps are taken:

1. Determine the direction of data transfer. A `READ` statement indicates that data is to be transferred from a file to program variables. A `WRITE` or `PRINT` statement indicates that data is to be transferred from program variables to a file.

2. Identify the unit. The unit identified by a data transfer I/O statement must be connected to a file when execution of the statement begins. Note that the file can be preconnected.

3. Establish the format, if one is specified. If specified, the format specifier is given in the data transfer statement and implies list-directed, namelist, or formatted data transfer.

4. Position the file prior to transferring the data. The position depends on the method of access (sequential or direct) and is described in Section 1.3.2, page 46.

5. Transfer data between the file and the entities specified by the I/O item list (if any). The list items are processed in the order of the I/O list for all data transfer I/O statements, except namelist input data transfer statements, which are processed in the order of the entities specified within the input records. For namelist output data transfer, the output items are specified by the namelist when a namelist group name is used.

6. Determine if an error, end-of-record, or end-of-file condition exists. If one of these conditions occurs, the status of the file and the I/O items is as specified in Section 1.4, page 47.

7. Position the file after transferring the data. The file position depends on whether one of the conditions in step 6 above occurred or if the data transfer was advancing or nonadvancing.

8. Cause the variables specified in the `IOSTAT=` and `SIZE=` specifiers, if present, to become defined. See the description of these specifiers in the `READ` and `WRITE` data transfer statements in Section 1.2.2, page 21.

9. If `ERR=`, `END=`, or `EOR=` specifiers appear in the statement, transfer to the branch target corresponding to the condition that occurs. If an `IOSTAT=` specifier appears in the statement and the label specifier corresponding to the condition that occurs does not appear in the statement, the next statement in the execution sequence is executed. Otherwise, the execution

of the program terminates. See the descriptions of these label specifiers in Section 1.2.2, page 21.

1.3.1 Data transfer

Data is transferred between records in the file and entities in the I/O list or namelist. The list items are processed in the order of the I/O list for all data transfer I/O statements except namelist data transfer statements. The list items for a namelist formatted data transfer input statement are processed in the order of the entities specified within the input records. The list items for a namelist data transfer output statement are processed in the order in which the data objects (variables) are specified in the namelist group object list.

The next item to be processed in the input or output item list is the *next effective item*, which is used to determine the interaction between the I/O item list and the format specification.

Zero-sized arrays and implied-`DO` lists with zero iteration counts are ignored in determining the next effective item.

Before beginning the I/O processing of a particular list item, the compiler first evaluates all values needed to determine which entities are specified by the list item. For example, the subscripts of a variable in an I/O list are evaluated before any data is transferred.

The value of an item that appears early in an I/O list can affect the processing of an item that appears later in the list. In the following example, the old value of `N` identifies the unit, but the new value of `N` is the subscript of `X`:

```
READ(N) N, X(N)
```

1.3.2 File position prior to data transfer

The file position prior to data transfer depends on the method of access: sequential or direct.

For sequential access on input, if there is a current record, the file position is not changed; this will be the case if the previous data transfer was nonadvancing. Otherwise, the file is positioned at the beginning of the next record and this record becomes the current record. Input must not occur if there is no next record (there must be an end-of-file record at least) or if there is a current record and the last data transfer statement accessing the file performed output.

If the file contains an end-of-file record, the file must not be positioned after the end-of-file record prior to data transfer. If you are using the CF90 compiler, multiple end-of-file records are permitted for some file structures. The MIPSpro 7 Fortran 90 compiler does not allow multiple end-of-file records, but a `REWIND` or `BACKSPACE` statement can be used to reposition the file.

> **ANSI/ISO:** The Fortran standard does not allow files that contain an end-of-file record to be positioned after the end-of-file record prior to data transfer.

For sequential access on output, if there is a current record, the file position is not changed; this will be the case if the previous data transfer was nonadvancing. Otherwise, a new record is created as the next record of the file; this new record becomes the last and current record of the file and the file is positioned at the beginning of this record.

For direct access, the file is positioned at the beginning of the record specified. This record becomes the current record.

1.3.3 File position after data transfer

If an error condition exists, the file position is indeterminate. If no error condition exists, but an end-of-file condition exists as a result of reading an end-of-file record, the file is positioned after the end-of-file record.

If no error condition or end-of-file condition exists, but an end-of-record condition exists, the file is positioned after the record just read. If no error condition, end-of-file condition, or end-of-record condition exists, and the data transfer was a nonadvancing input or output statement, the file position is not changed. In all other cases, the file is positioned after the record just read or written, and that record becomes the preceding record.

1.4 Error and other conditions in I/O statements

In step 6 of the execution model in Section 1.3, page 45, the data transfer statements admit the occurrence of error and other conditions during the execution of the statement. The `OPEN`, `CLOSE`, `INQUIRE`, and file positioning statements also admit the occurrence of error conditions.

Whenever an error condition is detected, the variable of the `IOSTAT=` specifier is assigned a positive value, if it is present. Also, if the `ERR=` specifier is present, the program transfers to the branch target specified by the `ERR=` specifier upon completion of the I/O statement.

In addition, two other conditions might be detected: end-of-file and end-of-record. For each of these conditions, branches can be provided using the END= and EOR= specifiers in a READ statement to which the program branches upon completion of the READ statement. Also, the variable of the IOSTAT= specifier, if present, is set to a unique negative integer value, indicating which condition occurred.

An end-of-file condition occurs when either an end-of-file record is encountered during a sequential READ statement, or an attempt is made to read beyond the end of an internal file. An end-of-file condition can occur at the beginning of the execution of an input statement or during the execution of a formatted READ statement when more than one record is required by the interaction of the format specification and the input item list.

An end-of-record condition occurs when a nonadvancing input statement attempts to transfer data from beyond the end of the record.

Two or more conditions can occur during a single execution of an I/O statement. If one or more of the conditions is an error condition, one of the error conditions takes precedence, in the sense that the IOSTAT= specifier is given a positive value designating the particular error condition and the action taken by the I/O statement is as if only that error condition occurred.

In summary, an error condition can be generated by any of the I/O statements, and an end-of-file or end-of-record condition can be generated by a READ statement. The IOSTAT=, END=, EOR=, and ERR= specifiers allow the program to recover from such conditions rather than terminate execution of the program. In particular, when any one of these conditions occurs, the CF90 and MIPSpro 7 Fortran 90 compilers take the following actions:

1. If an end-of-record condition occurs and if the connection has been made with the PAD= specifier of YES, the record is padded, as necessary, with blanks to satisfy the input item list and the corresponding data edit descriptor. For the file position, see Section 1.3.3, page 47.

2. Execution of the I/O statement terminates.

3. If an error condition occurs, the position of the file becomes indeterminate; if an end-of-file condition occurs, the file is positioned after the end-of-file record; if an end-of-record condition occurs, the file is positioned after the current record.

4. If the statement also contains an IOSTAT= specifier, the variable specified becomes defined with a nonzero integer value; the value is positive if an

error condition occurs and is negative if either an end-of-file or end-of-record condition occurs.

5. If the statement is a READ statement with a SIZE= specifier and an end-of-record condition occurs, then the variable specified by the SIZE= specifier becomes defined with an integer value equal to the number of characters read from the input record; blank padding characters inserted because the PAD= specifier is YES are not counted. For the file position, see Section 1.3.3, page 47.

6. All implied-DO variables in the I/O statement become undefined; if an error or end-of-file condition occurs during execution of a READ statement, all list items become undefined; if an error condition occurs during the execution of an INQUIRE statement, all specifier variables except the IOSTAT= variable become undefined.

7. If an END= specifier is present and an end-of-file condition occurs, execution continues with the statement specified by the label in the END= specifier. If an EOR= specifier is present and an end-of-record condition occurs, execution continues with the statement specified by the label in the EOR= specifier. If an ERR= specifier is present and an error condition occurs, execution continues with the statement specified by the label in the ERR= specifier.

   If none of the preceding cases applies, but the I/O statement contains an IOSTAT= specifier, the normal execution sequence is resumed. If there is no IOSTAT=, END=, EOR=, or ERR= specifier, and an error or other condition occurs, the program terminates execution.

The following program segment illustrates how to handle end-of-file and error conditions:

```
READ (FMT = "(E8.3)", UNIT=3, IOSTAT=IOSS) X

IF (IOSS < 0) THEN

   ! PERFORM END-OF-FILE PROCESSING ON THE
   ! FILE CONNECTED TO UNIT 3.
   CALL END_PROCESSING

ELSE IF (IOSS > 0) THEN

   ! PERFORM ERROR PROCESSING
   CALL ERROR_PROCESSING
```

```
END IF
```

The procedure END_PROCESSING is used to handle the case where an end-of-file condition occurs and the procedure ERROR_PROCESSING is used to handle all other error conditions, because an end-of-record condition cannot occur.

1.5 OPEN statement

The OPEN statement establishes a connection between a unit and an external file and determines the connection properties. In order to perform data transfers (reading and writing), the file must be connected with an OPEN statement or be preconnected. It can also be used to change certain properties of the connection between the file and the unit, to create a file that is preconnected, or create a file and connect it.

The OPEN statement can appear anywhere in a program, and once executed, the connection of the unit to the file is valid in the main program or any subprogram for the remainder of that execution, unless a CLOSE statement affecting the connection is executed.

If a file is already connected to one unit, it must not be connected to a different unit.

1.5.1 Connecting a file to a unit

The OPEN statement connects an external file to a unit. If the file does not exist, it is created. If a unit is already connected to a file that exists, an OPEN statement referring to that unit may be executed. If the FILE= specifier is not included, the unit remains connected to the file. If the FILE= specifier names the same file, the OPEN statement may change the connection properties. If it specifies a different file by name, the effect is as if a CLOSE statement without a STATUS= specifier is executed on that unit and the OPEN statement is then executed. For information on default values for the STATUS= specifier, see Section 1.6.1.3, page 61.

1.5.2 Creating a file on a preconnected unit

If a unit is preconnected to a file that does not exist, the OPEN statement creates the file and establishes properties of the connection.

1.5.3 Changing the connection properties

Execution of an OPEN statement can change the properties of a connection that is already established. The properties that can be changed are those indicated by BLANK=, DELIM=, PAD=, ERR=, and IOSTAT= specifiers. If new values for DELIM=, PAD=, and BLANK= specifiers are specified, these will be used in subsequent data transfer statements; otherwise, the old ones will be used. However, the values in ERR= and IOSTAT= specifiers, if present, apply only to the OPEN statement being executed; after that, the values of these specifiers have no effect. If no ERR= or IOSTAT= specifier appears in the new OPEN statement, error conditions will terminate the execution of the program.

1.5.4 Form of the OPEN statement

The OPEN statement and the connection specifier are defined as follows:

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

A unit specifier is required. If the keyword UNIT is omitted, the scalar integer expression must be the first item in the list. Note that the form * for the unit specifier is not allowed in the OPEN statement. However, in cases where the default external unit specified by an asterisk also corresponds to a nonnegative unit specifier (such as unit numbers 5 and 6 on many systems), inquiries about these default units and connection properties are possible.

A specifier must not appear more than once in an OPEN statement.

The character expression established for many of the specifiers must contain one of the permitted values from the list of alternative values for each specifier described in the following section. For example, OLD, NEW, REPLACE, UNKNOWN, or SCRATCH are permitted for the STATUS= specifier; any other combination of letters is not permitted. Trailing blanks in any specifier are ignored. The value specified can contain both uppercase and lowercase letters.

If the last data transfer to a unit connected for sequential access to a particular file is an output data transfer statement, an OPEN statement for that unit connecting it to a different file writes an end-of-file record to the original file.

1.5.5 The connection specifiers

The OPEN statement specifies the connection properties between the file and the unit, using keyword specifiers, which are described in this section. Table 4 indicates the possible values for the specifiers in an OPEN statement and their default values when the specifier is omitted.

Table 4. Values for keyword specifier variables in an OPEN statement

Specifier	Possible values	Default value
ACCESS=	DIRECT, SEQUENTIAL	SEQUENTIAL
ACTION=	READ, WRITE, READWRITE	File dependent
BLANK=	NULL, ZERO	NULL
DELIM=	APOSTROPHE, QUOTE, NONE	NONE
ERR=	Label	No default
FILE=	Character expression	fort. is prepended to the unit number
FORM=	FORMATTED	FORMATTED for sequential access
	UNFORMATTED	UNFORMATTED for direct access
IOSTAT=	Scalar default integer variable	No default
PAD=	YES, NO	YES
POSITION=	ASIS, REWIND, APPEND	ASIS
RECL=	Positive scalar integer expression	File dependent

Specifier	Possible values	Default value
`STATUS=`	`OLD`, `NEW`, `UNKNOWN`, `REPLACE`, `SCRATCH`	`UNKNOWN`
`UNIT=`	Scalar integer expression	No default

1.5.5.1 `UNIT=` specifier

The `UNIT=` specifier has the following format:

```
[ UNIT= ] scalar_int_expr
```

The value of *scalar_int_expr* must be nonnegative.

A unit specifier with an external unit is required. If the keyword `UNIT` is omitted, the unit specifier must be the first item in the list.

A unit number identifies one and only one external unit in all program units in a Fortran program.

1.5.5.2 `ACCESS=` specifier

The `ACCESS=` specifier has the following format:

```
ACCESS= scalar_char_expr
```

For *scalar_char_expr*, specify either `DIRECT` or `SEQUENTIAL`. `DIRECT` specifies the direct access method for data transfer. `SEQUENTIAL` specifies the sequential access method for data transfer; `SEQUENTIAL` is the default.

If the file exists, the method specified must be an allowed access method for the file.

If the file is new, the allowed access methods given for the file must include the one indicated.

If the `ACCESS=` specifier has the value `DIRECT`, a `RECL=` specifier must be present.

1.5.5.3 `ACTION=` specifier

The `ACTION=` specifier has the following format:

```
ACTION= scalar_char_expr
```

Specify one of the following for *scalar_char_expr*:

Value	Function
`READ`	Indicates that `WRITE`, `PRINT`, and `ENDFILE` statements are prohibited
`WRITE`	Indicates that `READ` statements are prohibited
`READWRITE`	Indicates that any I/O statement is permitted

The default value depends on the file structure.

If `READWRITE` is an allowed `ACTION=` specifier, `READ` and `WRITE` must also be allowed `ACTION=` specifiers.

For an existing file, the specified action must be an allowed action for the file.

For a new file, the value of the `ACTION=` specifier must be one of the allowed actions for the file.

1.5.5.4 `BLANK=` specifier

The `BLANK=` specifier has the following format:

```
BLANK= scalar_char_expr
```

For *scalar_char_expr*, specify either `NULL` or `ZERO`. `NULL` specifies that the compiler should ignore all blanks in numeric fields; this is the default. `ZERO` specifies that the I/O library should interpret all blanks except leading blanks as zeros.

A field of all blanks evaluates to zero regardless of the argument specified to `BLANK=`.

The `BLANK=` specifier may be specified for files connected only for formatted I/O.

1.5.5.5 `DELIM=` specifier

The `DELIM=` specifier has the following format:

```
DELIM= scalar_char_expr
```

Specify one of the following for *scalar_char_variable*:

Value	Function
`APOSTROPHE`	Use the apostrophe as the delimiting character for character constants written by a list-directed or namelist-formatted data transfer statement.
`QUOTE`	Use the quotation mark as the delimiting character for character constants written by a list-directed or namelist-formatted data transfer statement.
`NONE`	Use no delimiter to delimit character constants written by a list-directed or namelist-formatted data transfer statement. This is the default.

If the `DELIM=` specifier is `APOSTROPHE`, any occurrence of an apostrophe within a character constant will be doubled; if the `DELIM=` specifier is `QUOTE`, any occurrence of a quote within a character constant will be doubled.

The specifier is permitted only for a file connected for formatted I/O; it is ignored for formatted input.

1.5.5.6 `ERR=` specifier

The `ERR=` specifier has the following format:

```
ERR= label
```

The program branches to the label in the `ERR=` specifier if an error occurs in the `OPEN` statement. The label must be the label of a branch target statement in the same scoping unit as the `OPEN` statement.

If an error condition occurs, the position of the file becomes indeterminate.

If an `IOSTAT=` specifier is present and an error condition occurs, the `IOSTAT` variable specified becomes defined with a positive value.

1.5.5.7 `FILE=` specifier

The `FILE=` specifier has the following format:

```
FILE= scalar_char_expr
```

For *scalar_char_expr*, specify the name of the file to be connected. This is called the *file name expression*.

Trailing blanks in the file name are ignored.

The rules regarding file name lengths and path name lengths depend upon your platform, as follows:

- UNICOS and UNICOS/mk systems allow 256 characters for a file name. This file name can include a directory name and 1023 characters for the full path name.
- IRIX systems allow 256 characters for a file name. The full path name is limited to 1023 characters.

Example:

```
FILENM = 'dir/fnam'
FNUM = '/subfnum'
OPEN (UNIT = 8, FILE = 'beauty')
OPEN (UNIT = 9, FILE = FILENM)
OPEN (UNIT = 10, FILE = FNAME//FNUM)
```

The `FILE=` specifier must appear if the `STATUS=` specifier is `OLD`, `NEW`, or `REPLACE`; the `FILE=` specifier must not appear if the `STATUS=` specifier is `SCRATCH`.

If the `FILE=` specifier is omitted and the unit is not already connected to a file, the `STATUS=` specifier must have the value `SCRATCH` and the unit becomes connected to a file with the unit number appended to the `fort.` string.

1.5.5.8 `FORM=` specifier

The `FORM=` specifier has the following format:

```
FORM= scalar_char_expr
```

For *scalar_char_expr*, specify either `FORMATTED` or `UNFORMATTED`. `FORMATTED` indicates that all records are formatted. `UNFORMATTED` indicates that all records are unformatted.

The default value is `UNFORMATTED` if the file is connected for direct access and the `FORM=` specifier is absent.

The default value is `FORMATTED` if the file is connected for sequential access and the `FORM=` specifier is absent.

If the file is new, the allowed forms given for the file must include the one indicated.

If the file exists, the form specified by the `FORM=` specifier must be one of the allowed forms for the file.

1.5.5.9 `IOSTAT=` specifier

The `IOSTAT=` specifier has the following format:

`IOSTAT=` *scalar_default_int_variable*

The I/O library will return an integer value in *scalar_default_int_variable*. If *scalar_default_int_variable* > `0`, an error condition occurred. If *scalar_default_int_variable* = `0`, no error condition occurred. Note that the value cannot be negative.

The `IOSTAT=` specifier applies to the execution of the `OPEN` statement itself.

1.5.5.10 `PAD=` specifier

The `PAD=` specifier has the following format:

`PAD=` *scalar_char_expr*

For *scalar_char_expr*, specify either `YES` or `NO`. `YES` indicates that the I/O library should use blank padding when the input item list and format specification require more data than the record contains; this is the default. `NO` indicates that the I/O library requires that the input record contain the data indicated by the input list and format specification.

The `PAD=` specifier is permitted only for a file connected for formatted I/O; it is ignored for formatted output.

If this specifier has the value `YES` and an end-of-record condition occurs, the data transfer behaves as if the record were padded with sufficient blanks to satisfy the input item and the corresponding data edit descriptor.

1.5.5.11 `POSITION=` specifier

The `POSITION=` specifier has the following format:

```
POSITION= scalar_char_expr
```

Specify one of the following for *scalar_char_expr*:

Value	Function
`ASIS`	Indicates that the file position is to remain unchanged for a connected file and is unspecified for a file that is not connected. `ASIS` permits an `OPEN` statement to change other connection properties of a file that is already connected without changing its position. This is the default.
`REWIND`	Indicates that the file is to be positioned at its initial point.
`APPEND`	Indicates that the file is to be positioned at the terminal point or just before an end-of-file record, if there is one.

The file must be connected for sequential access.

If the file is new, it is positioned at its initial point, regardless of the value of the `POSITION=` specifier.

1.5.5.12 `RECL=` specifier

The `RECL=` specifier has the following format:

```
RECL= scalar_int_expr
```

For *scalar_int_expr*, specify a positive value that specifies either the length of each direct access record or the maximum length of a sequential access record.

If the `RECL=` specifier is absent for a file connected for sequential access, the default value is 267 characters (on UNICOS and UNICOS/mk systems) or 1000 characters (on IRIX systems).

The `RECL=` specifier must be present for a file connected for direct access.

If the file is connected for formatted I/O, the length is the number of characters.

If the file is connected for unformatted I/O, the length is measured in 8-bit bytes.

If the file exists, the length of the record specified must be an allowed record length.

If the file does not exist, the file is created with the specified length as an allowed length.

1.5.5.13 STATUS= specifier

The STATUS= specifier has the following format:

```
STATUS= scalar_char_expr
```

Specify one of the following for *scalar_char_expr*:

Value	Function
OLD	Requires that the file exist.
NEW	Requires that the file not exist.
UNKNOWN	Indicates that the file has an unknown status. This is the default.
REPLACE	Indicates that if the file does not exist, the file is created and given a status of OLD; if the file does exist, the file is deleted, a new file is created with the same name, and the file is given a status of OLD.
SCRATCH	Indicates that an unnamed file is to be created and connected to the specified unit; it is to exist either until the program terminates or a CLOSE statement is executed on that unit.

Scratch files must be unnamed; that is, the STATUS= specifier must not be SCRATCH when a FILE= specifier is present. The term *scratch file* refers to this temporary file.

Note that if the STATUS= specifier is REPLACE, the specifier in this statement is not changed to OLD; only the file status is considered to be OLD when the file is used in subsequently executed I/O statements, such as a CLOSE statement.

1.6 CLOSE statement

Executing a CLOSE statement terminates the connection between a file and a unit. Any connections not closed explicitly by a CLOSE statement are closed by the operating system when the program terminates, unless an error condition has terminated the program.

The CLOSE statement is defined as follows:

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

A specifier must not appear more than once in a CLOSE statement.

A CLOSE statement may appear in any program unit in an executing program.

A CLOSE statement may refer to a unit that is not connected or does not exist, but it has no effect.

If the last data transfer to a file connected for sequential access is an output data transfer statement, a CLOSE statement for a unit connected to this file writes an end-of-file record to the file.

After a unit has been disconnected by a CLOSE statement, it can be connected again to the same or a different file. Similarly, after a file has been disconnected by a CLOSE statement, it can also be connected to the same or a different unit, provided the file still exists. The following examples show these situations:

```
CLOSE (ERR=99, UNIT=9)
CLOSE (8, IOSTAT=IR, STATUS="KEEP")
```

The sections that follow describe the components of the *close_spec_list* in more detail.

1.6.1 UNIT= specifier

The UNIT= specifier has the following format:

```
[ UNIT= ] scalar_int_expr
```

The value of *scalar_int_expr* must be nonnegative.

A unit specifier is required. If the keyword `UNIT` is omitted, *scalar_int_expr* must be the first item in the list.

A unit number identifies one and only one external unit in all program units in a Fortran program.

1.6.1.1 ERR= specifier

The `ERR=` specifier has the following format:

```
ERR= label
```

If an error condition occurs, the position of the file becomes indeterminate.

If an `IOSTAT=` specifier is present and an error condition occurs, the `IOSTAT` variable specified becomes defined with a positive value.

The program branches to the label in the `ERR=` specifier if an error occurs in the `CLOSE` statement. The label must be the label of a branch target statement in the same scoping unit as the `CLOSE` statement.

1.6.1.2 IOSTAT= specifier

The format of the `IOSTAT=` specifier is as follows:

```
IOSTAT= scalar_default_int_variable
```

The I/O library returns an integer value in *scalar_default_int_variable*. If *scalar_default_int_variable* > `0`, an error condition occurred. If *scalar_default_int_variable* = `0`, no error condition occurred. Note that the value cannot be negative.

The `IOSTAT=` specifier applies to the execution of the `CLOSE` statement itself.

1.6.1.3 STATUS= specifier

The format of the `STATUS=` specifier is as follows:

```
STATUS= scalar_char_expr
```

For *scalar_char_expr*, specify either KEEP or DELETE. KEEP indicates that the file is to continue to exist after closing the file. DELETE indicates that the file will not exist after closing the file.

If the unit has been opened with a STATUS= specifier of SCRATCH, the default is DELETE. If the unit has been opened with any other value of the STATUS= specifier, the default is KEEP.

KEEP must not be specified for a file whose file status is SCRATCH.

If KEEP is specified for a file that does not exist, the file does not exist after the CLOSE statement is executed.

1.7 Inquiring about files

An inquiry can be made about a file's existence, connection, access method, or other properties. For each property inquired about, a scalar variable of default kind must be supplied; that variable is given a value that answers the inquiry. The variable can be tested and optional execution paths can be selected based on the answer returned. The inquiry specifiers are determined by keywords in the INQUIRE statement. The only exception is the unit specifier, which, if no keyword is specified, must be the first specifier. A file inquiry can be made by unit number, file name, or an output item list. When inquiring by an output item list, an output item list that might be used in an unformatted direct access output statement must be present.

1.7.1 The INQUIRE statement

There are three kinds of INQUIRE statements: inquiry by unit, by name, and by an output item list. The first two kinds use the *inquire_spec_list* form of the INQUIRE statement. The third kind uses the IOLENGTH form. Inquiry by unit uses a unit specifier. Inquiry by file uses a file specifier with the keyword FILE=. These statements are defined as follows:

R923	*inquire_stmt*	**is**	`INQUIRE (`*inquire_spec_list*`)`
		or	`INQUIRE (IOLENGTH =` *scalar_default_int_variable*`)` *output_item_list*
R924	*inquire_spec*	**is**	`[ UNIT = ]` *external_file_unit*
		or	`FILE =` *file_name_expr*
		or	`IOSTAT =` *scalar_default_int_variable*
		or	`ERR =` *label*
		or	`EXIST =` *scalar_default_logical_variable*
		or	`OPENED =` *scalar_default_logical_variable*
		or	`NUMBER =` *scalar_default_int_variable*
		or	`NAMED =` *scalar_default_logical_variable*
		or	`NAME =` *scalar_char_variable*
		or	`SEQUENTIAL =` *scalar_char_variable*
		or	`ACCESS =` *scalar_char_variable*
		or	`DIRECT =` *scalar_char_variable*
		or	`FORM =` *scalar_char_variable*
		or	`FORMATTED =` *scalar_char_variable*
		or	`UNFORMATTED =` *scalar_char_variable*
		or	`RECL =` *scalar_default_int_variable*
		or	`NEXTREC =` *scalar_default_int_variable*
		or	`BLANK =` *scalar_char_variable*
		or	`POSITION =` *scalar_char_variable*
		or	`ACTION =` *scalar_char_variable*
		or	`READ =` *scalar_char_variable*
		or	`WRITE =` *scalar_char_variable*
		or	`READWRITE =` *scalar_char_variable*
		or	`DELIM =` *scalar_char_variable*
		or	`PAD =` *scalar_char_variable*

An `INQUIRE` statement with an *inquire_spec_list* must have a unit specifier or a `FILE=` specifier, but not both. If the keyword `UNIT` is omitted, a scalar integer expression must be the first item in the list and must have a nonnegative value.

No specifier can appear more than once in a given inquiry specifier list.

For an inquiry by an output item list, the output item list must be a valid output list for an unformatted, direct-access, output statement. The length value returned in the scalar default integer variable must be a value that is acceptable when used as the value of the `RECL=` specifier in an `OPEN` statement. This value can be used in a `RECL=` specifier to connect a file whose records will hold the data indicated by the output list of the `INQUIRE` statement.

An `INQUIRE` statement can be executed before or after a file is connected to a unit. The specifier values returned by the `INQUIRE` statement are those current at the time at which the `INQUIRE` statement is executed.

A variable appearing in a specifier or any entity associated with it must not appear in another specifier in the same `INQUIRE` statement if that variable can become defined or undefined as a result of executing the `INQUIRE` statement. That is, do not try to assign two inquiry results to the same variable.

Except for the `NAME=` specifier, character values are returned in uppercase.

If an error condition occurs during the execution of an `INQUIRE` statement, all the inquiry specifier variables become undefined except the `IOSTAT=` specifier. The following are examples of `INQUIRE` statements:

```
INQUIRE (9, EXIST = EX)
INQUIRE (FILE = "T123", OPENED = OP, ACCESS = AC)
INQUIRE (IOLENGTH = IOLEN)  X, Y, CAT
```

1.7.2 Specifiers for inquiry by unit or file name

This section describes the format and effect of the inquiry specifiers that may appear in the inquiry by unit and file forms of the `INQUIRE` statement.

1.7.2.1 `UNIT=` specifier

The format of the `UNIT=` specifier is as follows:

```
[ UNIT= ] scalar_int_expr
```

The *scalar_int_expr* indicates an external unit (R902). The value of *scalar_int_expr* must be nonnegative.

To inquire by unit, a unit specifier must be present, and a file specifier cannot be present. If the keyword `UNIT` is omitted, *scalar_int_expr* must be the first item in the list.

A unit number identifies one and only one external unit in all program units in a Fortran program.

The file is the file connected to the unit, if one is connected; otherwise, the file does not exist.

1.7.2.2 `ACCESS=` specifier

The format of the `ACCESS=` specifier is as follows:

```
ACCESS= scalar_char_variable
```

One of the following is returned in *scalar_char_variable*:

Value	Function
`SEQUENTIAL`	Indicates that the file is connected for sequential access
`DIRECT`	Indicates that the file is connected for direct access
`UNDEFINED`	Indicates that the file is not connected

1.7.2.3 `ACTION=` specifier

The format of the `ACTION=` specifier is as follows:

```
ACTION= scalar_char_variable
```

One of the following is returned in *scalar_char_variable*:

Value	Function
`READ`	Indicates that the file is connected with access limited to input only
`WRITE`	Indicates that the file is connected with access limited to output only
`READWRITE`	Indicates that the file is connected for both input and output
`UNDEFINED`	Indicates that the file is not connected

1.7.2.4 `BLANK=` specifier

The format of the `BLANK=` specifier is as follows:

```
BLANK= scalar_char_variable
```

One of the following is returned in *scalar_char_variable*:

Value	Function
`NULL`	Indicates that null blank control is in effect
`ZERO`	Indicates that zero blank control is in effect
`UNDEFINED`	Indicates that the file is not connected for formatted I/O or the file is not connected at all

See the `BLANK=` specifier for the `OPEN` statement in Section 1.5.5, page 52, for the meaning of null and zero blank control.

1.7.2.5 `DELIM=` specifier

The format of the `DELIM=` specifier is as follows:

```
DELIM= scalar_char_variable
```

One of the following is returned in *scalar_char_variable*:

Value	Function
`APOSTROPHE`	Indicates that an apostrophe is used as the delimiter in list-directed and namelist-formatted output
`QUOTE`	Indicates that the quotation mark is used as the delimiter in list-directed and namelist-formatted output
`NONE`	Indicates that there is no delimiting character in list-directed and namelist-formatted output
`UNDEFINED`	Indicates that the file is not connected or the file is not connected for formatted I/O

1.7.2.6 `DIRECT=` specifier

The format of the `DIRECT=` specifier is as follows:

```
DIRECT= scalar_char_variable
```

One of the following is returned in *scalar_char_variable*:

Value	Function
`YES`	Indicates that direct access is an allowed access method
`NO`	Indicates that direct access is not an allowed access method
`UNKNOWN`	Indicates that the system does not know if direct access is allowed

1.7.2.7 `ERR=` specifier

The format of the `ERR=` specifier is as follows:

```
ERR= label
```

The program branches to the label in the `ERR=` specifier if there is an error in the execution of the `INQUIRE` statement itself. The label must be the label of a branch target statement in the same scoping unit as the `INQUIRE` statement.

If an error condition occurs, the position of the file becomes indeterminate.

If an `IOSTAT=` specifier is present and an error condition occurs, the `IOSTAT` variable specified becomes defined with a positive value. All other inquiry specifier variables become undefined.

1.7.2.8 `EXIST=` specifier

The format of the `EXIST=` specifier is as follows:

```
EXIST= scalar_default_logical_variable
```

The system returns either `.TRUE.` or `.FALSE.` in *scalar_default_logical_variable.* `.TRUE.` indicates that the file or unit exists. `.FALSE.` indicates that the file or unit does not exist.

1.7.2.9 `FILE=` specifier

The format of the `FILE=` specifier is as follows:

```
FILE= scalar_char_expr
```

To inquire by file, a file specifier must be present, and a unit specifier cannot be present.

The value of *scalar_char_expr* must be an acceptable file name. Trailing blanks are ignored. Both uppercase and lowercase letters are acceptable. The use of uppercase and lowercase is significant for file names.

The file name can refer to a file not connected or to one that does not exist.

1.7.2.10 `FORM=` specifier

The format of the `FORM=` specifier is as follows:

```
FORM= scalar_char_variable
```

The system returns one of the following in *scalar_char_variable*:

Value	Function
`FORMATTED`	Indicates that the file is connected for formatted I/O
`UNFORMATTED`	Indicates that the file is connected for unformatted I/O
`UNDEFINED`	Indicates that the file is not connected

1.7.2.11 `FORMATTED=` specifier

The format of the `FORMATTED=` specifier is as follows:

```
FORMATTED= scalar_char_variable
```

The system returns one of the following in *scalar_char_variable*:

Value	Function
`YES`	Indicates that formatted I/O is an allowed form for the file
`NO`	Indicates that formatted I/O is not an allowed form for the file

`UNKNOWN`	Indicates that the system cannot determine if formatted I/O is an allowed form for the file

1.7.2.12 `IOSTAT=` specifier

The format of the `IOSTAT=` specifier is as follows:

`IOSTAT=` *scalar_default_int_variable*

The I/O library returns an integer value in *scalar_default_int_variable*. If *scalar_default_int_variable* > `0`, an error condition occurred. If *scalar_default_int_variable* = `0`, no error condition occurred. Note that the value cannot be negative.

The `IOSTAT=` specifier applies to the execution of the `INQUIRE` statement itself.

1.7.2.13 `NAME=` specifier

The format of the `NAME=` specifier is as follows:

`NAME=` *scalar_char_variable*

The I/O library returns either a file name or an undefined value in *scalar_char_variable*. If the value is a file name, it is the name of the file connected to the unit, if the file has a name. An undefined value indicates that either file does not have a name or no file is connected to the unit.

A name different from the one specified in the `FILE=` specifier on the `OPEN` statement may be returned.

1.7.2.14 `NAMED=` specifier

The format of the `NAMED=` specifier is as follows:

`NAMED=` *scalar_default_logical_variable*

The system returns either `.TRUE.` or `.FALSE.` in *scalar_default_logical_variable*. `.TRUE.` indicates that the file has a name. `.FALSE.` indicates that the file does not have a name.

1.7.2.15 `NEXTREC=` specifier

The format of the `NEXTREC=` specifier is as follows:

```
NEXTREC= scalar_default_int_variable
```

The system returns one of the following in *scalar_default_int_variable*:

Value	Function
last record number + `1`	Indicates the next record number to be read or written in a file connected for direct access. The value is one more than the last record number read or written.
`1`	Indicates that no records have been processed.
undefined value	Indicates that the file is not connected for direct access or the file position is indeterminate because of a previous error condition.

This inquiry is used for files connected for direct access.

1.7.2.16 `NUMBER=` specifier

The format of the `NUMBER=` specifier is as follows:

```
NUMBER= scalar_default_int_variable
```

The system returns either a unit number or –1 in *scalar_default_int_variable*. A unit number indicates the number of the unit connected to the file. –1 indicates that there is no unit connected to the file.

1.7.2.17 `OPENED=` specifier

The format of the `OPENED=` specifier is as follows:

```
OPENED= scalar_default_logical_variable
```

The system returns either `.TRUE.` or `.FALSE.` in *scalar_default_logical_variable*. `.TRUE.` indicates that the file or unit is connected (that is, opened). `.FALSE.` indicates that the file or unit is not connected (that is, not opened).

1.7.2.18 PAD= specifier

The format of the PAD= specifier is as follows:

```
PAD= scalar_char_variable
```

The system returns either NO or YES in *scalar_char_variable*. NO indicates that the file or unit is connected with the PAD= specifier set to NO. YES indicates that either the file or unit is connected with the PAD= specifier set to YES or that the file or unit is not connected.

1.7.2.19 POSITION= specifier

The format of the POSITION= specifier is as follows:

```
POSITION= scalar_char_variable
```

The system returns one of the following in *scalar_char_variable*:

Value	Function
REWIND	Indicates that the file is connected with its position at the initial point
APPEND	Indicates that the file is connected with its position at the terminal point
ASIS	Indicates that the file is connected without changing its position
UNDEFINED	Indicates that either the file is not connected or it is connected for direct access

If any repositioning has occurred since the file was connected, the value returned is ASIS. It is not equal to REWIND unless positioned at the initial point, and it is not equal to APPEND unless positioned at the terminal point.

1.7.2.20 READ= specifier

The format of the READ= specifier is as follows:

```
READ= scalar_char_variable
```

The system returns one of the following in *scalar_char_variable*:

Value	Function
`YES`	Indicates that `READ` is one of the allowed actions for the file
`NO`	Indicates that `READ` is not one of the allowed actions for the file
`UNKNOWN`	Indicates that the action for the file cannot be determined

1.7.2.21 `READWRITE=` specifier

The format of the `READWRITE=` specifier is as follows:

```
READWRITE= scalar_char_variable
```

The system returns one of the following in *scalar_char_variable*:

Value	Function
`YES`	Indicates that `READWRITE` is an allowed action for the file
`NO`	Indicates that `READWRITE` is not an allowed action for the file
`UNKNOWN`	Indicates that the action for the file cannot be determined

1.7.2.22 `RECL=` specifier

The format of the `RECL=` specifier is as follows:

```
RECL= scalar_default_int_variable
```

The system returns either a maximum record length or an undefined value in *scalar_default_int_variable*. The maximum record length is the record length, if the file is connected for sequential access, or the length of each record, if the file is connected for direct access. An undefined value indicates that the file does not exist.

For a formatted file, the length is the number of characters for all records.

For an unformatted file, the length is in 8-bit bytes.

1.7.2.23 `SEQUENTIAL=` specifier

The format of the `SEQUENTIAL=` specifier is as follows:

```
SEQUENTIAL= scalar_char_variable
```

The system returns one of the following in *scalar_char_variable*:

Value	Function
`YES`	Indicates that sequential access is an allowed access method
`NO`	Indicates that sequential access is not an allowed access method
`UNKNOWN`	Indicates that the access is not known

1.7.2.24 `UNFORMATTED=` specifier

The format of the `UNFORMATTED=` specifier is as follows:

```
UNFORMATTED= scalar_char_variable
```

The system returns one of the following in *scalar_char_variable*:

Value	Function
`YES`	Indicates that unformatted I/O is an allowed form for the file
`NO`	Indicates that unformatted I/O is not an allowed form for the file
`UNKNOWN`	Indicates that the form cannot be determined

1.7.2.25 `WRITE=` specifier

The format of the `WRITE=` specifier is as follows:

```
WRITE= scalar_char_variable
```

The system returns one of the following in *scalar_char_variable*:

Value	Function
`YES`	Indicates that `WRITE` is an allowed action for the file
`NO`	Indicates that `WRITE` is not an allowed action for the file
`UNKNOWN`	Indicates that the action cannot be determined

1.7.3 Table of values assigned by the `INQUIRE` statement

Table 5, page 75, summarizes the values assigned to the `INQUIRE` specifier variables by the execution of an `INQUIRE` statement.

Table 5. Values for specifier variables in an INQUIRE statement

	INQUIRE by file		INQUIRE by unit	
Specifier	Unconnected	Connected	Connected	Unconnected
ACCESS=	UNDEFINED	SEQUENTIAL or DIRECT		UNDEFINED
ACTION=	UNDEFINED	READ, WRITE, or READWRITE		UNDEFINED
BLANK=	UNDEFINED	NULL, ZERO, or UNDEFINED		UNDEFINED
DELIM=	UNDEFINED	APOSTROPHE, QUOTE, NONE, or UNDEFINED		UNDEFINED
DIRECT=	UNKNOWN	YES, NO, or UNKNOWN		UNKNOWN
EXIST=	.TRUE. if file exists, .FALSE. otherwise		.TRUE. if unit exists, .FALSE. otherwise	
FORM=	UNDEFINED	FORMATTED or UNFORMATTED		UNDEFINED
FORMATTED=	UNKNOWN	YES, NO, or UNKNOWN		UNKNOWN
IOSTAT=	0 for no error, a positive integer for an error			
NAME=	File name (may not be same as FILE= value)	File name if named, else undefined		Undefined
NAMED=	.TRUE.	.TRUE. if file named, .FALSE. otherwise		.FALSE.
NEXTREC=	Undefined	If direct access, next record number; else undefined		Undefined
NUMBER=	-1	Unit number		-1
OPENED=	.FALSE.	.TRUE.		.FALSE.
PAD=	YES	YES or NO		YES
POSITION=	UNDEFINED	REWIND, APPEND, ASIS, or UNDEFINED		UNDEFINED
READ=	UNKNOWN	YES, NO, or UNKNOWN		UNKNOWN
READWRITE=	UNKNOWN	YES, NO, or UNKNOWN		UNKNOWN
RECL=	Undefined	If direct access, record length; else, maximum record length		Undefined
SEQUENTIAL=	UNKNOWN	YES, NO, or UNKNOWN		UNKNOWN

	INQUIRE by file		INQUIRE by unit	
Specifier	Unconnected	Connected	Connected	Unconnected
UNFORMATTED=	UNKNOWN	YES, NO, or UNKNOWN		UNKNOWN
WRITE=	UNKNOWN	YES, NO, or UNKNOWN		UNKNOWN
IOLENGTH=	RECL = value for *output_item_list*			

1.8 File positioning statements

Execution of a data transfer statement usually changes the file position. In addition, there are three statements whose main purpose is to change the file position. Changing the position backwards by one record is called *backspacing* and is performed by the BACKSPACE statement. Changing the position to the beginning of the file is called *rewinding* and is performed by the REWIND statement. The ENDFILE statement writes an end-of-file record and positions the file after the end-of-file record. The following are file positioning statements:

```
BACKSPACE 9
BACKSPACE (UNIT = 10)
BACKSPACE (ERR = 99, UNIT = 8, IOSTAT = STATUS)
REWIND (ERR = 102, UNIT = 10)
ENDFILE (10, IOSTAT = IERR)
ENDFILE (11)
```

The file positioning statements have common components.

- The *external_file_unit* can be a unit specifier or an external name. If it is a unit specifier, it must have a nonnegative value. See Section 1.1.5, page 10, for information on units and the *external_file_unit*.
- The *position_spec_list* is described in Section 1.8.4, page 79.

ANSI/ISO: The Fortran 90 standard does not address using external names as *external_file_unit* specifiers.

The following sections describe the BACKSPACE, REWIND, and ENDFILE statements in more detail.

1.8.1 BACKSPACE statement

Executing a BACKSPACE statement causes the file to be positioned before the current record if there is a current record, or before the preceding record if there is no current record. If there is no current record and no preceding record, the file position is not changed. If the preceding record is an end-of-file record, the file becomes positioned before the end-of-file record.

If the last data transfer to a file connected for sequential access was an output data transfer statement, a BACKSPACE statement on that file writes an end-of-file record to the file. If there is a preceding record, the BACKSPACE statement causes the file to become positioned before the record that precedes the end-of-file record.

The BACKSPACE statement is defined as follows:

R919	*backspace_stmt*	**is**	BACKSPACE *external_file_unit*
		or	BACKSPACE (*position_spec_list*)

The BACKSPACE statement is used only to position external files.

If the file is already at its initial point, a BACKSPACE statement has no effect. If the file is connected, but does not exist, backspacing is prohibited. Backspacing over records written using list-directed or namelist formatting is prohibited.

```
BACKSPACE  ERROR_UNIT        ! ERROR_UNIT is an
                             !    integer variable
BACKSPACE (10, &             ! STAT is an integer
                             !    variable of
           IOSTAT = STAT)    !    default type
```

1.8.2 The REWIND statement

A REWIND statement positions the file at its initial point. Rewinding has no effect on the file position when the file is already positioned at its initial point. If a file does not exist, but it is connected, rewinding the file is permitted, but it has no effect.

If the last data transfer to a file was an output data transfer statement, a REWIND statement on that file writes an end-of-file record to the file.

The file must be connected for sequential access.

The REWIND statement is defined as follows:

R921	*rewind_stmt*	**is**	`REWIND` *external_file_unit*
		or	`REWIND (`*position_spec_list*`)`

The `REWIND` statement is used only to position external files.

```
REWIND  INPUT_UNIT      ! INPUT_UNIT is an integer
                        !   variable
REWIND (10, ERR = 200)  ! 200 is a label of branch
                        !   target in this
                        !   scoping unit
```

1.8.3 The `ENDFILE` statement

The `ENDFILE` writes an end-of-file record as the next record and positions the file after the end-of-file record written. Writing records past the end-of-file record is prohibited, except for certain file structures.

If you are using the CF90 compiler, you can find more information on file structures in the *Application Programmer's I/O Guide,* publication SG–2168.

If you are using the MIPSpro 7 Fortran 90 compiler, IRIX record blocking is the default form for sequential unformatted files. Text files are used for all other types of Fortran I/O.

ANSI/ISO: The Fortran 90 standard does not address writing past an end-of-file record.

After executing an `ENDFILE` statement, it is necessary to execute a `BACKSPACE` or `REWIND` statement to position the file ahead of the end-of-file record before reading or writing the file. If the file is connected but does not exist, writing an end-of-file record creates the file.

The file must be connected for sequential access.

The `ENDFILE` statement is defined as follows:

R920	*endfile_stmt*	**is**	`ENDFILE` *external_file_unit*
		or	`ENDFILE (`*position_spec_list*`)`

The `ENDFILE` statement is used only to position external files.

```
                     ! OUTPUT_UNIT is an integer variable.
ENDFILE  OUTPUT_UNIT
ENDFILE (10, ERR = 200, IOSTAT = ST)
                     !   200 is a label of a branch
                     !   target in this scoping unit
                     !   ST is a default scalar integer
                     !   variable
```

A file can be connected for sequential and direct access, but not for both simultaneously. If a file is connected for sequential access and an `ENDFILE` statement is executed on the file, only those records written before the `ENDFILE` statement is executed are considered to have been written. Consequently, if the file is subsequently connected for direct access, only those records before the end-of-file record can be read.

1.8.4 Specifiers for file position statements

The *position_spec_list* is defined as follows (note that no specifier can be repeated in a *position_spec_list*):

R922	*position_spec*	**is**	`[ UNIT = ]` *scalar_int_expr*
		or	`IOSTAT =` *scalar_default_int_variable*
		or	`ERR =` *label*

The following sections describe the form and effect of the position specifiers that can appear in the file positioning statements.

1.8.4.1 `UNIT=` specifier

The format of the `UNIT=` specifier is as follows:

`[ UNIT= ]` *scalar_int_expr*

The *scalar_int_expr* specifies an external unit (R902). The value of the *scalar_int_expr* must be nonnegative.

A unit specifier is required. If the keyword `UNIT` is omitted, the scalar integer expression must be the first item in the position specifier list. A unit number identifies one and only one external unit in all program units in a Fortran program.

There must be a file connected to the unit, and the unit must be connected for sequential access.

1.8.4.2 `IOSTAT=` specifier

The format of the `IOSTAT=` specifier is as follows:

```
IOSTAT= scalar_default_int_variable
```

The I/O library returns an integer value in *scalar_default_int_variable*. If *scalar_default_int_variable* > `0`, an error condition occurred. If *scalar_default_int_variable* = `0`, no error condition occurred. Note that the value cannot be negative.

The `IOSTAT=` specifier applies to the execution of the file positioning statement itself.

1.8.4.3 `ERR=` specifier

The format of the `ERR=` specifier is as follows:

```
ERR= label
```

The program branches to the label in the `ERR=` specifier if there is an error in the execution of the particular file positioning statement itself. The label must be the label of a branch target statement in the same scoping unit as the file positioning statement.

If an error condition occurs, the position of the file becomes indeterminate.

If an `IOSTAT=` specifier is present and an error condition occurs, the `IOSTAT` variable specified becomes defined with a positive value.

1.9 Restrictions on I/O specifiers, list items, and statements

Any function reference appearing in an I/O statement specifier or in an I/O list must not cause the execution of another I/O statement on the same file or unit. Note that such function references also must not have side effects that change any object in the same statement.

```
WRITE (10, FMT = "(10I5)", REC = FCN(I)) &
      X(FCN(J)), I, J
```

The function FCN must not contain an I/O statement using unit 10 and must not change its argument because I and J are also output list items.

A unit or file might not have all of the properties (for example, all access methods or all forms) required for it by execution of certain I/O statements. If this is the case, such I/O statements must not refer to files or units limited in this way. For example, if unit 5 cannot support unformatted sequential files, the following OPEN statement must not appear in a program:

```
OPEN (UNIT = 5, IOSTAT = IERR,  &
      ACCESS = "SEQUENTIAL", FORM = "UNFORMATTED")
```

Input and Output (I/O) Editing [2]

Usually, data is stored in memory as the values of variables in some binary form. On the other hand, formatted data records in a file consist of characters. Thus, when data is read from a formatted record, it must be converted from characters to the internal representation. When data is written to a formatted record, it must be converted from the internal representation into a string of characters.

A *format specification* provides the information necessary to determine how these conversions are to be performed. The format specification is basically a list of *edit descriptors*, of which there are three general types: data edit descriptors, control edit descriptors, and string edit descriptors. There is a data edit descriptor for each data value in the input/output (I/O) list of the data transfer statement. Control edit descriptors specify the spacing and position within a record, new records, interpretation of blanks, and plus sign suppression. String edit descriptors transfer strings of characters represented in format specifications to output records.

The format reference that indicates where to find the format can be a statement label that identifies a `FORMAT` statement, or it can be a character expression giving the format directly. Using either method is called *explicit formatting*.

There are two other cases where formatting of a different sort applies. These are list-directed and namelist formatting. Formatting (that is, conversion) occurs without specifically providing the editing information usually contained in a format specification. In these cases, the editing or formatting is implicit; that is, the details about the width of fields, forms of output values, and location of output fields within the records is determined by the type of each data value in the I/O list.

This chapter describes the following information:

- The two methods of specifying explicit formatting.
- The three general types of edit descriptors and an algorithm that describes the correspondence of the data edit descriptors and items in the data item list of the data transfer statement.
- The two methods of implicit formatting: list-directed and namelist.

Table 6, page 84, Table 7, page 84, and Table 8, page 85, list the control, data, and string edit descriptors and provide a brief description of each.

Table 6. Summary of control edit descriptors

Descriptor	Description
BN	Ignore nonleading blanks in numeric input fields
BZ	Treat nonleading blanks in numeric input fields as zeros
S	Do not print optional plus sign
SP	Print plus sign
SS	Do not print plus sign
T	Tab to specified position
TL	Tab left the specified number of positions
TR	Tab right the specified number of positions
X	Tab right the specified number of positions
$	Suppress carriage control
/	End current record and move to beginning of next record
:	Stop format processing if no further input/output list items exist
P	Interpret certain real numbers with a specified scale factor

ANSI/ISO: The Fortran 90 standard does not specify the $ control edit descriptor.

Table 7. Summary of data edit descriptors

Descriptor	Description
A	Convert data of type character
B	Convert data of type integer to/from a binary base
D	Convert data of type real; same as E edit descriptor
E	Convert data of type real with an exponent
EN	Convert data of type real to engineering notation
ES	Convert data of type real to scientific notation
F	Convert data of type real with no exponent on output

Descriptor	Description
`G`	Convert data of all intrinsic types
`I`	Convert data of type integer
`L`	Convert data of type logical
`O`	Convert data of type integer to/from an octal base
`Z`	Convert data of type integer to/from a hexadecimal base

Table 8. Summary of string edit descriptors

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

The CF90 and MIPSpro 7 Fortran 90 compilers extend the use of `B`, `O`, and `Z` edit descriptors to type complex (`KIND=8`), logical, and real (`KIND=8`).

Table 9, page 85, and Table 10, page 86, show the usage of the CF90 and MIPSpro 7 Fortran 90 compilers' edit descriptors with all intrinsic data types. In these tables:

- NA indicates illegal usage that is not allowed.
- I,O indicates legal usage for input and output.
- O indicates legal usage for output.

Table 9. Edit descriptors with data types

Data types	I	F	E	D	G	L	A	B	O	Z	R
Character	NA	NA	NA	NA	NA	NA	I,O	NA	NA	NA	NA
Complex	NA	I,O	I,O	I,O	I,O	NA	I,O	I,O	I,O	I,O	I,O
Double precision	NA	I,O	I,O	I,O	I,O	NA	NA	NA	O	NA	NA

Data types	I	F	E	D	G	L	A	B	O	Z	R
Integer	I,O	NA	NA	NA	NA	NA	I,O	I,O	I,O	I,O	I,O
Logical	NA	NA	NA	NA	NA	I,O	I,O	I,O	I,O	I,O	I,O
Real	NA	I,O	I,O	I,O	I,O	NA	I,O	I,O	I,O	I,O	I,O

Table 10 shows the format restrictions for integer, logical, and real variables that are valid when you use SEGLDR or `cld`(1) and the `EQUIV` directive. To change the limitations for read and write operations, specify `EQUIV=$RNOCHK($RCHK)` or `EQUIV=$WNOCHK($WCHK)`, respectively. Both of these `EQUIV` statements must be specified if changes are desired.

Note: SEGLDR and `cld`(1) are not supported on IRIX systems.

Table 10. Descriptors and data types when the SEGLDR or `cld`(1) `EQUIV` directive is used (UNICOS and UNICOS/mk systems)

Data types	I	F	E	D	G	L	A	B	O	Z	R
Integer	I,O	I,O	I,O	NA	I,O	I,O	I,O	I,O	I,O	I,O	I,O
Logical	I,O	I,O	I,O	NA	I,O	I,O	I,O	I,O	I,O	I,O	I,O
Real	I,O	I,O	I,O	I,O	I,O	I,O	I,O	I,O	I,O	I,O	I,O

2.1 Explicit formatting

Explicit formatting information can be provided in one of the following ways:

- Contained in a `FORMAT` statement, as follows:

```
    WRITE (6, 100) LIGHT, AND, HEAVY
100 FORMAT (F10.2, I5, E16.8)
```

- Given as the value of a character expression, as follows:

```
WRITE (6, ’(F10.2, I5, E16.8)’ ) LIGHT, AND, HEAVY
```

2.1.1 `FORMAT` statement

The `FORMAT` statement is defined as follows:

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

As the following sample format shows, the `FORMAT` statement must be labeled. The label is used in the I/O statement to reference a particular `FORMAT` statement.

label `FORMAT (`[*format_item_list*]`)`

Each I/O statement in a program can reference a unique `FORMAT` statement, or a `FORMAT` statement can be used in more than one I/O statement.

2.1.2 Character expression format specifications

A character expression can be used in the I/O statement as a format specification. The leading part of the character expression must be a valid format specification including the parentheses; that is, the value of the expression must be such that the first nonblank character is a left parenthesis, followed by a list of valid format items, followed by a right parenthesis.

All variables in the character expression must be defined when the I/O statement is executed.

Characters can appear following the last right parenthesis in the character expression; they have no effect.

If the expression is a character array, the format is scanned in array element order. For example, the following format specification is valid (where `A` is a character array of length at least 8 and size at least 2):

```
A(1) = '(1X,I3,'
A(2) = ' I7, I9)'
PRINT  A, MUTT, AND, JEFF
```

If the expression is an array element, the format must be entirely contained within that element.

If the expression is a character variable, it or any part of it must not be redefined or become undefined during the execution of the I/O statement.

If the expression is a character constant delimited by apostrophes, two apostrophes must be written to represent each apostrophe in the format specification. If a format specification contains, in turn, a character constant delimited by apostrophes, there must be two apostrophes for each of the apostrophe delimiters, and each apostrophe within the character constant must be represented by four apostrophes (see the example below). If quotes are used for the string delimiters and quotes are used within the string, a similar doubling of the quote marks is required. One way to avoid nested delimiter problems is to use delimiters different from the characters within the format specification, if possible. Another way to avoid the problem is to put the character expression in the I/O list instead of the format specification. This is shown in the second line of the following example, where `A16` is a character edit descriptor specifying a field width of 16 positions:

```
PRINT '(''I can''''t hear you'')'
PRINT "(A16)", "I can't hear you"
```

The preceding example can be written without a field width (character count), as in the following example:

```
PRINT "(A)", "I can't hear you"
```

When a character expression is used as a format specification, errors in the format specification might not be detected until the program is executed because the format specification may not be complete or known until the data transfer statement is executed.

2.2 Format specifications

Each item in the format item list of a format specification is an *edit descriptor*, which can be a data edit descriptor, control edit descriptor, or character string edit descriptor. Each data list item must have a corresponding data edit descriptor.

Blanks can be used freely in format specifications without affecting the interpretation of the edit descriptors, both in the free and fixed source forms. Named constants are not allowed in format specifications because they would create ambiguities in the interpretation of the format specifications. For example, if `N12` were a named integer constant with value 15, the engineering format edit descriptor `E N12.4` could be interpreted as the edit descriptor `EN12.4` or `E15.4`.

The *format_item* is defined as follows:

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

In the preceding format, *r* must be a default integer literal constant and is called a *repeat factor*. If a repeat factor is not present, it is as if it were present with the value of 1.

r must not have a kind value specified for it. *r* must be a positive integer.

The comma between edit descriptors can be omitted in the following cases:

- Between the scale factor (`P`) and the numeric edit descriptors `F`, `E`, `EN`, `ES`, `D`, or `G`
- Before a new record indicated by a slash when there is no repeat factor present
- After the slash for a new record
- Before or after the colon edit descriptor

Blanks can be used in the following cases:

- Before the first left parenthesis.
- Anywhere in the format specification. Blanks within a character edit descriptor are significant. Blanks outside of a character edit descriptor are not significant.

Edit descriptors can be nested within parentheses, or an edit descriptor can be preceded by a repeat factor indicating that the edit descriptor is repeated. A parenthesized list of edit descriptors can also be preceded by a repeat factor; this indicates that the entire list is to be repeated.

The following examples illustrate many of the edit descriptors that are described in the following sections:

```
100 FORMAT (2(5E10.1, I10) / (1X, SP, I7, ES15.2))
110 FORMAT (I10, F14.1, EN10.2)
120 FORMAT (TR4, L4, 15X, A20)
130 FORMAT ('MORE SNOW')
140 FORMAT (9X, 3A5, 7/ 10X, 3L4)
```

2.2.1 Data edit descriptor form

Data edit descriptors specify the conversion of values to and from the internal representation to the character representation in the formatted record of a file. The data edit descriptors are defined as follows:

R1005	*data_edit_desc*	**is**	`I`*w*[`.`*m*]
		or	`B`*w*[`.`*m*]
		or	`O`*w*[`.`*m*]
		or	`Z`*w*[`.`*m*]
		or	`F`*w*`.`*d*
		or	`E`*w*`.`*d*[`E`*e*]
		or	`EN`*w*`.`*d*[`E`*e*]
		or	`ES`*w*`.`*d*[`E`*e*]
		or	`G`*w*`.`*d*[`E`*e*]
		or	`L`*w*
		or	`A`[*w*]
		or	`D`*w*`.`*d*
EXT		**or**	`D`*w*`.`*d*`E`*e*
EXT		**or**	`R`*w*
R1007	*m*	**is**	*int_literal_constant*
R1008	*d*	**is**	*int_literal_constant*
R1009	*e*	**is**	*int_literal_constant*

ANSI/ISO: The Fortran 90 standard does not specify the `D`*w*`.`*d*`E`*e* or `R`*w* data edit descriptors.

The *w*, *m*, *d*, and *e* descriptors must be default integer literal constants. They have the following meanings:

Descriptor	Specification
w	The width of the field
m	The least number of digits in the field
d	The number of fractional digits in the field
e	The number of digits in the exponent

The *w*, *m*, *d*, and *e* descriptors must not have a kind value specified for them.

The *w* and *e* descriptors must be positive.

The `I`, `B`, `O`, `Z`, `F`, `E`, `EN`, `ES`, `G`, `L`, `A`, and `D` edit descriptors indicate the manner of editing.

The meanings of the data edit descriptors are described in Section 2.5, page 98, through Section 2.7, page 111.

2.2.2 Control edit descriptor form

Control edit descriptors determine the position, form, layout, and interpretation of characters transferred to and from formatted records in a file. The control edit descriptors are defined as follows:

R1010	*control_edit_desc*	**is**	*position_edit_desc*
		or	`[ r ] /`
		or	`:`
		or	*sign_edit_desc*
		or	*k* `P`
		or	*blank_interp_edit_desc*
R1011	*k*	**is**	*signed_int_literal_constant*
R1012	*position_edit_desc*	**is**	`T` *n*
		or	`TL` *n*
		or	`TR` *n*
		or	*n* `X`
EXT		**or**	`$`
R1013	*n*	**is**	*int_literal_constant*
R1014	*sign_edit_desc*	**is**	`S`
		or	`SP`
		or	`SS`
R1015	*blank_interp_edit_desc*	**is**	`BN`
		or	`BZ`

The *n* and *r* descriptors must be default integer literal constants. *k* must be a signed default integer literal constant. *n*, *k*, and *r* must not have a kind value specified for them. *k*, *n*, and *r* have the following meanings:

Descriptor	Specification
k	A scale factor.
n	A position in the record to move, relative to the left tab limit, for descriptor `T`. Specifies the number of spaces to move for descriptors `X`, `TR`, and `TL`. *n* must be positive.
r	The repeat factor.

The control edit descriptors `T`, `TL`, `TR`, `X`, and `$` are called *position edit descriptors* (R1012). The control edit descriptors `S`, `SP`, and `SS` are called *sign edit descriptors* (R1014). The control edit descriptors `BN` and `BZ` are called *blank interpretation edit descriptors* (R1015).

ANSI/ISO: The Fortran 90 standard does not describe the `$` as a position edit descriptor.

In *k* `P`, *k* is called the *scale factor.*

`T`, `TL`, `TR`, `X`, `$`, slash, colon, `S`, `SP`, `SS`, `P`, `BN`, and `BZ` indicate the manner of editing and are described in detail in Section 2.8, page 112.

2.3 Character string edit descriptor form

Character string edit descriptors specify character strings to be transmitted to the formatted output record of a file. The character string edit descriptors are defined as follows:

R1016	*char_string_edit_desc*	**is**	*char_literal_constant*
		is	*c* `H` *rep_char* [*rep_char*] ...
R1017	*c*	**or**	*int_literal_constant*

c must be a default integer literal constant and is a character count. *c* must not have a kind value specified for it. *c* must be positive.

The second form of the character edit descriptor is obsolescent.

The character string edit descriptors are described in detail in Section 2.10, page 119.

2.4 Formatted data transfer

The format specification indicates how data is transferred by READ, WRITE, and PRINT statements. The data transfer typically involves a conversion of a data value. The particular conversion depends on the next data input or output item, along with the current edit descriptor in the format specification.

```
    READ (*, '(A7, I10, E16.8)' )  X, Y, Z
    WRITE (*, 100)  X, Y, Z
100 FORMAT (A7, I10, E16.3)
```

An empty format specification ( ) is restricted to I/O statements with no items in the I/O data item list or a list of items all of which have zero size. A zero-length character string requires an A edit descriptor.

The effect on I/O of an empty format specification depends on whether the data transfer is advancing or nonadvancing, and on whether there is a current record. The effect is described by the following cases.

When the data transfer is advancing:

- If there is no current record, then:
  - On input, skip the next record
  - On output, write an empty record
- If there is a current record, then:
  - On input, skip to the end of the current record
  - On output, terminate the current record

When the data transfer is nonadvancing:

- If there is no current record, then:
  - On input, move to the initial point of the next record
  - On output, create an empty record and move to its initial point
- If there is a current record, then:
  - On input, there is no effect
  - On output, there is no effect

The following example program segment reads N character strings, each of length M, from a single record and then advances to the beginning of the next record:

```
DO I = 1, N
   READ (5, ’(A1)’, ADVANCE=’NO’) &
      (CHARS(I)(J:J), J = 1, M)
ENDDO
READ (5, ’()’, ADVANCE = ’YES’)
```

The data and the edit descriptors are converted in a left-to-right fashion, except for repeated items, which are repeated until either the data items are exhausted or the repeat number is reached. A complex data item requires two data edit descriptors for data items of type real; that is, two of the edit descriptors E, F, D, ES, EN, or G. The two edit descriptors may be different.

Control edit descriptors and character edit descriptors do not require a corresponding data item in the list. The effect is directly on the record transferred. When the data items are completed, no further change is made to record on output, and no change is made to the position in the file on input.

2.4.1 Parentheses usage

The effect of parentheses in a format specification depends on the nesting level of the parentheses.

When the rightmost right parenthesis of a complete format specification is encountered and there are no more data items, the I/O data transfer terminates. Remember that the format specification can be given by a character string expression. In such a case, the right parenthesis matching the leftmost left parenthesis can be followed by any characters, including parentheses. None of these trailing characters are relevant to the rules and restrictions in this chapter. For example, the following character string can be used as a format specification in a character string, and the part after the first right parenthesis is ignored:

```
’(I5,E16.8,A5)  (This part is ignored)’
```

If there are more data items when the rightmost right parenthesis is encountered, format control continues beginning at the left parenthesis corresponding to the last preceding right parenthesis in the specification, if there is one, with an implied slash (/) to cause a new record to begin. If there is no preceding right parenthesis, the reversion is to the beginning of the format.

If there is a repeat factor encountered when reverting, the repeat before the parenthesis is reused.

Reversion does not affect the scale factors, the sign control edit descriptor, or blank interpretation. These remain in effect for the duration of the format action. Example:

```
CHR_FMT = '(I5, 4(3F10.2, 10X), E20.4)'
```

In the previous example, if the character string were used in a formatted output data transfer statement, the first output data item must be an integer. The remaining items must be of type real (or complex); 13 real values are written on the first record after the integer, and the next real values are written in each new record, 13 at a time, until the data items are exhausted. All but the last record will have 13 real values written, 3 real values using the `F10.2` edit descriptor, 10 blanks, followed by 3 more real values and 10 blanks repeated 4 times in total, followed by a real value using the `E20.4` edit descriptor. This behavior is described in more detail in the next section.

2.4.2 Correspondence between a data-edit descriptor and a list item

The best way to describe how this correspondence is determined is to think of two markers, one beginning at the first item of the I/O data item list and the other beginning at the first left parenthesis of the format specification. Before describing how each marker proceeds through each list, the I/O data item list is considered to be expanded by writing out each element of an array, each component of a structure, each part (real and imaginary) of each item of type complex, and each iteration of each implied-`DO` list. The expanded item list is called the *effective data item list*, and each item in the list is called an *effective item*. Note that zero-sized arrays yield no effective items, but zero-length character objects yield effective items. Also, the format specification is considered expanded for each repeat factor preceding any data or slash edit descriptor but not a parenthesized format item list. If the data item list is nonempty, there must be at least one data edit descriptor in the format specification. Given the effective data item list and expanded format specification, the markers proceed as follows:

1. The marker proceeds through the format specification until the first data edit descriptor or right parenthesis is encountered. Any control edit descriptor or string edit descriptor encountered before the first data edit descriptor is encountered is interpreted according to its definition, each possibly changing the position within the record or the position within the file, or changing the interpretation of data in the record or conversion of data to the record.

2. If a data edit descriptor is encountered first, the effective data item pointed to by the marker in the data item list is transferred and converted according to the data edit descriptor, and the marker in the data item list proceeds to the next effective data item.

3. If a right parenthesis is encountered and the right parenthesis is not the outermost one of the format specification, the repeat factor in front of the matching left parenthesis is reduced by one. If the reduced factor is nonzero, the marker scans right from this left parenthesis, looking for a data edit descriptor as above. If the repeat factor becomes zero, the format specification marker then proceeds right from the right parenthesis, again looking for a data edit descriptor. If the right parenthesis is the outermost right parenthesis, the marker reverts to the left parenthesis corresponding to the last preceding right parenthesis, if there is one; if there is no preceding right parenthesis, it reverts to the first left parenthesis of the format specification. Upon reversion, a slash edit descriptor is interpreted implicitly, and the format marker proceeds right from this left parenthesis, honoring any repeat factor in front of it.

4. If no effective data item remains when a data edit descriptor is encountered or when a colon edit descriptor is encountered, the I/O operation terminates.

To illustrate how this works, consider the following example:

```
INTEGER  A(3)
COMPLEX  C
TYPE RATIONAL
   INTEGER  N, D
END TYPE
TYPE(RATIONAL)  R
   . . .
WRITE (*, &
     "('A and C appear on line 1, R appears on line 2' &
      / (1X, 3I5, 2F5.2) )" )  A, C, R
```

The data item list is first expanded as described above. The expanded data item list becomes the following:

```
A(1), A(2), A(3), REAL(C), AIMAG(C), R%N, R%D
```

The format specification is also expanded and becomes the following:

```
('A and C appear on line 1, R appears on line 2' &
   / (1X, I5, I5, I5, F5.2, F5.2) )
```

A marker is established in the data item list, which initially points at the item A(1). A marker is also established in the format specification and initially points to the first left parenthesis. The marker in the format specification proceeds right to the first edit descriptor, which is the first I5. In so doing, it sees the string edit descriptor (which is transferred to the output record), the slash edit descriptor (which causes the previous record to terminate and to begin a new record), and the position edit descriptor (which positions the record at the second character, blank filling the first character). The item A(1) is then converted according to the I5 specification and the converted value is transferred to the output record. The marker in the data item list is moved to A(2). The format specification marker is moved right to the second I5 edit descriptor, and A(2) is converted and transferred to the output record. Similarly, A(3), the real part of C, and the imaginary part of C are converted and transferred to the output record.

At this point, the data item list marker is pointing at R%N, and the format specification marker begins scanning after the second F5.2 edit descriptor looking for the next edit descriptor. The first right parenthesis is encountered and the scan reverts back to the corresponding left parenthesis. The repeat factor in front of this parenthesis is 1 by default and is reduced by 1 to 0. The marker in the format specification proceeds right from the first right parenthesis, encountering the outermost right parenthesis and then reverts to the left parenthesis before the edit descriptor 1X. As a result, an implicit slash edit descriptor is interpreted, causing the previous output record to be completed and a new record to be started. The format specification marker scans right looking for a data edit descriptor, which is the first I5. In the process of the scan right, the position edit descriptor is interpreted, which positions the file at the second character of the next record (and blank fills the skipped character).

Finally, the N and D components of R are converted and transferred to the output record, using the first two I5 edit descriptors. The data item list marker finds no further items, so the output operation terminates.

The following is an example of writing a zero-sized array and zero-length character string using formatted output data transfer:

```
REAL  A(10)
CHARACTER(4)  CHR
   . . .
WRITE(6, '()')  A(1:0)
WRITE(6, '(A4)')  CHR(4:3)
```

An empty format specification is allowed for the first WRITE statement, because the array to be printed is a zero-sized array section. The format specification in the second `WRITE` statement is required to have at least one `A` edit descriptor because the effective data item is a zero-length character string, not a zero-sized array. In the first case, an empty record is written, and in the second case, a record consisting of four blank characters is written.

2.5 File positioning by format control

Assume that there is a current record being processed. After each data edit descriptor is used, the file position within that record is following the last character read or written by the particular edit descriptor. On output, after a string edit descriptor is used, the file is positioned within that record following the last character written. (See the description of the control edit descriptors `T`, `TL`, `TR`, `X`, and `$` for any special positioning within the current record; see the description of the slash edit descriptor for special positioning within the file.)

The remaining control edit descriptors do not affect the position within a record or within the file; they affect only the interpretation of the input characters or the form of the output character string or how subsequent edit descriptors are interpreted. The interpretation of the edit descriptors is not affected by whether the operation is an advancing or nonadvancing I/O operation.

2.6 Numeric editing

There are seven edit descriptors that cover numeric editing: `I`, `F`, `E`, `EN`, `ES`, `D`, and `G`. The following rules apply to all of them:

On input:

- Leading blanks are never significant.
- Plus signs can be omitted in the input data.
- A blank field is considered to be zero, regardless of the `BN` edit descriptor or the `BLANK=` specifier in effect.
- Within a field, blanks are interpreted in a manner that depends on the `BLANK=` specifier default for preconnected files, the `BLANK=` specifier provided in an `OPEN` statement for the unit, or if there is a `BN` or `BZ` blank edit descriptor in effect.
- In numeric fields that have a decimal point and correspond to `F`, `E`, `EN`, `ES`, `D`, or `G` edit descriptors, the decimal point in the input field overrides the placement of the decimal point specified by the edit descriptor specification.

- Data input is permitted to have more digits of significance than can be used to represent a number.
- The lowercase exponent letters `e` and `d` are equivalent to the corresponding uppercase exponent letters.

On output:

- A positive or zero value may have a plus sign, depending on the sign edit descriptors used.
- Negative values have a negative sign unless the printed value would be zero (because of the conversion rules). Negative zero is never produced.

  For example, suppose variable `SMALL` has the value `-0.000314`. On output, the characters transferred to output unit 6 by the following statements will not contain a negative sign:

  ```
      WRITE(6,10)  SMALL
  10  FORMAT(F5.2)
  ```

  The field will be as follows because a negative zero cannot be printed:

  b`0.00`

- The number is right justified in the field. Leading blanks may be inserted.
- If the number or the exponent is too large for the field width specified in the edit descriptor, the entire output field is filled with asterisks.
- Asterisks are not produced when the optional characters can be omitted and the output character string fits in the output field.

2.6.1 Integer editing

The Fortran 90 standard states that the I/O list item corresponding to an integer edit descriptor must be of type integer, except for the `G` edit descriptor. The integer edit descriptors are as follows:

```
Iw.m

Bw[.m]

Ow[.m]

Zw[.m]

Gw.d[Ee]
```

w, *m*, *d*, and *e* have the following meanings:

Descriptor	Specification
w	The width of the field
m	The minimum number of digits in the field
d	The number of fractional digits in the field
e	The number of digits in the exponent

For both input and output:

- The value of *m* must not exceed the value of *w*.
- For an integer I/O list item, the edit descriptor `G`*w*`.`*d*[`E`*e*] follows the same rules as the `I` *w* edit descriptor for the given value of *w*.

On input:

- *m* has no effect on an input field.
- For the `I` edit descriptor, the character string in the file must be an optionally signed integer constant, but it cannot have an underscore or a kind parameter value.
- For the `B`, `O`, or `Z` edit descriptors, the character string must be a string of blanks, a sign, and digits of binary, octal, or hexadecimal base, respectively. For example, the character string corresponding to a `B` edit descriptor must not contain digits 2 through 9. The character string corresponding to an `O` edit descriptor must not contain the digits 8 or 9. The character string corresponding to a `Z` edit descriptor may consist of the digits 0 through 9, the letters `A` through `F`, or the letters `a` through `f`.

ANSI/ISO: The Fortran 90 standard allows only binary, octal, or hexadecimal digits and blanks as input.

On output:

- If *m* is not present, each edit descriptor behaves as if it were present with the value 1.
- For the `I`*w*`.`*m* edit descriptor with *m* positive, the field is *w* characters wide and consists of zero or more leading blanks, followed by an optional sign, followed by an unsigned integer consisting of at least *m* digits. Leading zeros pad an integer field until there are *m* digits.
- For the `I`*w*`.0` edit descriptor, if the output list item has the value `0`, the field consists entirely of blanks.
- If *m* is not present and the value of the output list item is nonzero, the first digit must be nonzero; otherwise, the field consists of only one `0` digit with no sign character. Negative zero is never produced.
- For the `B`, `O`, or `Z` edit descriptors, the rules for forming the output field for the values *w* and *m* are the same as for the `I` edit descriptor except that the signed integer must consist of digits from the binary, octal, or hexadecimal base, respectively. A negative value is indicated in the encoding of the digits written; that is, a minus sign is not written.

ANSI/ISO: The Fortran 90 standard allows only an unsigned integer as an output list item.

2.6.2 Real editing

The `F`, `E`, `EN`, `ES`, and `D` edit descriptors specify editing for real and complex I/O list items. The `G` edit descriptor may also be used for real and complex items. Two such edit descriptors are required for each complex data item.

The forms of the edit descriptors for real values are as follows:

```
Fw.d

Ew.d[Ee]

ENw.d[Ee]

ESw.d[Ee]

Dw.d

Gw.d[Ee]

Dw.dEe
```

w, *d*, and *e* have the following meanings:

Descriptor	Specification
w	The width of the field
d	The number of fractional digits in the field
e	The number of digits in the exponent

2.6.2.1 F editing

The `F`*w*`.`*d* editing converts to or from a string occupying *w* positions.

For both input and output:

- *d* must not exceed *w*.
- The value in the input field or the value transferred to the output field may be signed.

On input:

- *d* specifies the number of decimal places in the input value if a decimal point is not present in the input field.
- The input field may be one of the following:
  - A signed integer or real literal constant but without an underscore and kind parameter value

  – A signed digit string followed by a sign followed by an unsigned digit string treated as an exponent
  – A signed digit string containing a decimal point followed by a sign followed by an unsigned digit string treated as an exponent

  Blanks may be freely inserted anywhere in the input field.

- If the input field contains a decimal point, the value of *d* has no effect.
- If there is no decimal point, a decimal point is inserted in front of the rightmost *d* digits of the nonexponent part, treating blanks as 0 digits or as if they were not present, according to the BLANK= specifier or the BZ or BN edit descriptor currently in effect.

  Consider the format specification F5.1. The following input data item is treated as the real number 19.9, if the BLANK= specifier is NULL or the BN edit descriptor is in effect, and is treated as the real number 1009.9 if the BLANK= specifier is ZERO or the BZ edit descriptor is in effect:

  1*bb*99

- There might be more digits in the number than the internal representation can contain.
- The number can contain an E or D indicating an exponent value; a field with a D exponent letter is processed identically to the same field with an E exponent letter. If there is no exponent field on input, the assumption is that the character string is followed by an exponent with the value -*k* where *k* is the scale factor established by a previous *k*P edit descriptor.

On output:

- *d* specifies the number of digits after the decimal point.
- The form of the output field consists of *w* positions comprised of leading blanks, if necessary, and an optionally signed real constant with a decimal point, rounded to *d* digits after the decimal point but with no exponent, underscore, or kind parameter value.
- Leading zeros are not written unless the number is less than 1.
- At least one zero is written if no other digits would appear.
- The scale factor has no effect on F editing on output.
- Negative zero is never produced.

Example:

```
    READ (5, 100)  X, Y
100 FORMAT (F10.2, F10.3)
```

Assume the following input field:

bbbb`6.42181234567890`

The values assigned to `X` and `Y` are `6.4218` and `1234567.89`, respectively. The value of *d* is ignored for `X` because the input field contains a decimal point.

2.6.2.2 E and D editing

The `E`*w*`.`*d*[`E`*e*] , `D`*w*`.`*d*`E`*e*, and `D`*w*`.`*d* edit descriptors convert to and from a string occupying *w* positions. For the edit descriptors `E`*w*`.`*d*[`E`*e*], `D`*w*`.`*d*`E`*e*, and `D`*w*`.`*d*, the field representing the floating point number contains *w* characters, including an exponent.

For both input and output:

- *w* is the field width, *d* is the number of places after the decimal, and *e* is the exponent width.
- *d* and *e* must not exceed *w*.

On input:

- The forms `E`*w*`.`*d*[`E`*e*], `D`*w*`.`*d*`E`*e*, and `D`*w*`.`*d* are the same as for `F`*w*`.`*d* editing, where either `E` or `D` in the input data record may indicate an exponent. *e* has no effect on input.

On output:

- The form of the output field for a scale factor of zero is as follows:

  `[±] [0] .` $x_1 x_2$ `. . .` x_d *exp*

  The ± symbol signifies a plus or a minus.

  x_1 x_2... x_d are the *d* most significant digits of the data value after rounding.

  exp is a decimal exponent having one of the forms specified in Table 11, where each z_i is a decimal digit.

Table 11. Forms for the exponent *exp* in `E` and `D` editing

Edit descriptor	Absolute value of exponent	Form of exponent
`E`*w*`.`*d*	$\mid exp \mid \leq 99$	`E` $\pm z_1 z_2$
	$99 < \mid exp \mid \leq 999$	$\pm z_1 z_2 z_3$
	$999 < \mid exp \mid \leq 2466$	$\pm z_1 z_2 z_3 z_4$
`E`*w*`.`*d*`E`*e*	$\mid exp \mid \leq 10^e - 1$	`E`$\pm z_1 z_2 . . . z_e$
`D`*w*`.`*d*	$\mid exp \mid \leq 99$	`E`$\pm z_1 z_2$
	$99 < \mid exp \mid \leq 999$	$\pm z_1 z_2 z_3$
	$999 < \mid exp \mid \leq 2466$	$\pm z_1 z_2 z_3 z_4$
`D`*w*`.`*d*`E`*e*	$\mid exp \mid \leq 10^e - 1$	`E`$\pm z_1 z_2 . . . z_e$

ANSI/ISO: The Fortran 90 standard restricts the use of `E`*w*`.`*d* and `D`*w*`.`*d* to an exponent less than or equal to 999. The `E`*w*`.`*d*`E`*e* form must be used.

- The sign in the exponent is always written.
- Plus is used for zero exponents.
- A scale factor *k*`P` can be used to specify the number of digits to the left of the decimal point, with the exponent adjusted accordingly; that is, the scale factor *k* controls the decimal normalization. If $-d < k \leq 0$, the output field contains the decimal point, exactly $\mid k \mid$ leading zeros, and $d - \mid k \mid$ significant digits. If $0 < k < d < + 2$, the output field contains exactly *k* significant digits to the left of the decimal point and $d - k + 1$ significant digits to the right of the decimal point. Other values of *k* are not permitted; that is, those values of *k* that will produce no digits to the left of the decimal point or specify fewer than zero digits to the right of the decimal point. The output field will contain *w* asterisks.
- The form of zero on output always contains a decimal point, *d* zero digits, and an exponent of 4 characters whose digits are zero.

Consider the following example:

```
    WRITE (6, 105)  Y, Z
105 FORMAT (E15.3,4PD15.3)
```

If the values of `Y` and `Z` are `-21.2` and `26542.1232` respectively, the output record produced is as follows:

```
bbbbb-0.212E+02bbb2654.212E+01
```

2.6.2.3 Engineering edit descriptor EN

The EN edit descriptor converts to or from a string using engineering notation for a value occupying *w* positions.

On input:

- The form EN*w*.*d*[E*e*] is the same as for F*w*.*d* editing.

On output:

- The output of the number is in the form of engineering notation, where the exponent is divisible by 3 and the absolute value of the significand is $1000 > | \textit{significand} | \geq 1$. This is the same form as the E edit descriptor, except for these exponent and significand differences.

  ```
      WRITE (6, 110)  B
  110 FORMAT (EN13.3)
  ```

  If the value of B is 0.212, the output record produced is as follows:

  ```
  bb212.000E-03
  ```

- The form of zero is the same as noted for the E edit descriptor except that there is exactly one zero digit before the decimal point.

- The form of the output field is as follows:

  $[\pm]\ yyy.x_1x_2\ \ldots\ x_d\ exp$

The ± symbol signifies a plus or a minus.

yyy are the 1 to 3 decimal digits representing the most significant digits of the value of the data after rounding (*yyy* is an integer such that $1 \leq yyy < 1000$ or, if the output value is zero, $yyy = 0$).

$x_1x_2 \ldots x_d$ *exp* are the *d* next most significant digits of the value of the data after rounding. If the output value is zero, the x_i are all 0.

exp is a decimal integer, divisible by 3, representing the exponent and must have one of the forms shown in Table 12, where each z_i is a decimal digit.

Table 12. Forms for the exponent *exp* in EN editing

Edit descriptor	Absolute value of exponent	Form of exponent
EN*w*.*d*	$\mid exp \mid \leq 99$	$E\pm z_1 z_2$
	$99 < \mid exp \mid \leq 999$	$\pm z_1 z_2 z_3$
	$999 < \mid exp \mid \leq 2466$	$\pm z_1 z_2 z_3 z_4$
EN*w*.*d*E*e*	$\mid exp \mid \leq 10^e - 1$	$E\pm z_1 z_2 . \; . \; . \; z_e$

- The sign in the exponent is always written. A plus sign is written if the exponent value is zero. The form EN*w*.*d*E*e* must be used with a sufficiently large value of *e* if $|exp| > 999$.

Internal value	Output field using SS, EN12.3
6.421	6.421E+00
–.5	–500.000E–03
.00217	2.170E–03
4721.3	4.721E+03

2.6.2.4 Scientific edit descriptor ES

The scientific edit descriptor ES converts to or from a string using scientific notation for a value occupying *w* positions.

On input:

- The form ES*w*.*d*[E*e*] is the same as for F*w*.*d* editing.

On output:

- The output of the number is in the form of scientific notation, where the absolute value of the significand is $10 > \mid significand \mid \geq 1$. This is the same form as the E edit descriptor, except for the significand difference. Example:

```
    WRITE (6, 110)  B
110 FORMAT (ES12.3)
```

If the value of B is 0.12345678, the output record produced is as follows:

*bbb*1.235E-01

- The form of zero is the same as noted for the EN edit descriptor.

- The form of the output field is as follows:

  [±] $y.x_1x_2 \; ... \; x_d \; exp$

  The ± symbol signifies a plus or a minus.

  y is a decimal digit representing the most significant digit of the value of the data after rounding (y is an integer such that $1 \le y < 10$ or, if the output value is zero, $y = 0$).

  $x_1x_2 \dots x_d$ are the d next most significant decimal digits of the value of the data after rounding. If the output value is zero, the x_i are all 0.

  exp is a decimal exponent of one of the forms, given in Table 13, where each z_i is a decimal digit.

Table 13. Forms for the exponent *exp* in ES editing

Edit descriptor	Absolute value of exponent	Form of exponent
ES*w*.*d*	$\mid exp \mid \le 99$	$E\pm z_1z_2$
	$99 < \mid exp \mid \le 999$	$\pm z_1z_2z_3$
	$999 < \mid exp \mid \le 2466$	$\pm z_1z_2z_3z_4$
ES*w*.*d*E*e*	$\mid exp \mid \le 10^e - 1$	$E\pm z_1z_2 . . . z_e$

- The sign in the exponent is always written.
- A plus sign is written if the exponent value is zero.
- The form ES*w*.*d*E*e* must be used with a sufficiently large value of *e* if $|exp| > 999$.

Internal value	Output field using SS, ES12.3
6.421	6.421E+00
–.5	–5.000E–01
.00217	2.170E–03

4721.3	4.721E+03

2.6.2.5 Complex editing

Complex editing follows the rules for numeric editing. Editing of complex numbers requires two real edit descriptors, the first one for the real part and the second one for the imaginary part. Different edit descriptors may be used for the two parts. Control and character string edit descriptors can be inserted between the edit descriptors for the real and imaginary parts. Example:

```
COMPLEX CM(2)
READ(5, "(4E7.2)") (CM(I), I = 1, 2)
```

Assume that the input record is as follows:

bb`55511`*bbb*`2146`*bbbb*`100`*bbbb*`621`

The values assigned to `CM(1)` and `CM(2)` are `555.11 + 21.46`*i* and `1.0 + 6.21`*i*, respectively.

2.6.2.6 Generalized editing of real data

`G`*w*`.`*d*[`E`*e*] converts to or from a string using generalized editing. The form for generalized editing is determined by the magnitude of the value of the number.

On input:

- The `G`*w*`.`*d*[`E`*e*] edit descriptor is the same as the `F`*w*`.`*d* edit descriptor.

On output:

- Assume that *N* is the magnitude of a number to be printed using a `G` edit descriptor. If *N* `= 0` and *d* is not zero, or if *N* is approximately between `0.1` and `10` Table 14, page 110, specifies the form of the output, where *n* is `4` for `G`*w*`.`*d* and *n* is *e* `+ 2` for `G`*w*`.`*d*`E`*e*. A *k*`P` scale factor has no effect.

- If *N* is outside this range, or if *N* is identically zero and *d* is zero, output editing with the edit descriptor *k*`PG`*w*`.`*d* [`E`*e*] is the same as that with *k*`PE`*w*`.`*d*[`E`*e*] .

Table 14. The form of the output using a `G` edit descriptor for a number of magnitude N

Magnitude of data[1]	s	t
$N = 0$	$w - n$[2]	$d - 1$
$0.1 - 0.5 \times 10^{-d-1} \leq N < 1 - 0.5 \times 10^{-d}$	$w - n$	d
$1 - 0.5 \times 10^{-d} \leq N < 10 - 0.5 \times 10^{-d+1}$	$w - n$	$d - 1$
$10 - 0.5 \times 10^{-d+1} \leq N < 100 - 0.5 \times 10^{-d+2}$	$w - n$	$d - 2$
...	...	
$10^{d-2} - 0.5 \times 10^{-2} \leq N < 10^{d-1} - 0.5 \times 10^{-1}$	$w - n$	1
$10^{d-1} - 0.5 \times 10^{-1} \leq N < 10^{d} - 0.5$	$w - n$	0

Example 1:

```
PRINT "(G10.1)", 8.76E1
```

The preceding statement produces the following output:

bbb`0.9E+02`

The magnitude of N is outside the range for an `F` edit descriptor; it yields the format `E10.1`.

Example 2:

```
PRINT "(G10.3)", 8.76E1
```

The preceding statement produces the following output:

bb`87.6`*bbbb*

With the `G10.3` edit descriptor, n is 4, and the format reduces to `F6.1,4X` (the fourth line in Table 14, page 110) because of the magnitude of the number.

Example 3:

```
PRINT "(G10.3E1)", 8.76E1
```

The preceding statement produces the following output because n is `3 (=1+2)`, and the format reduces to `F7.1,3X`:

1 Assume that the equivalent conversion for all is `F`*s*`.`*t*, *n*`X`, where `X` indicates blanks.

2 n is 4 for `G`*w*`.`*d* edit descriptors, and n is $e + 2$ for `G`*w*`.`*d*`E`*e* edit descriptors.

```
bbb87.6bbbb
```

2.7 Logical editing

The logical edit descriptors convert to or from a string representing a logical value that is true or false. The edit descriptors used for logical editing are as follows:

```
Lw

Gw.d[Ee]
```

w, *d*, and *e* have the following meanings:

Descriptor	Specification
w	The width of the field
d	The number of fractional digits in the field
e	The number of digits in the exponent

For both input and output:

- Generalized logical editing `G`*w*`.`*d*[`E`*e*] follows the rules for `L`*w* editing.

On input:

- The input field for a logical value consists of any number of blanks, followed by an optional period, followed by `T` or `F`, for a true or false value respectively, followed by any representable characters.
- The `T` or `F` and any following letters can be in lowercase. They are treated the same as the uppercase letters.

  For example, if you use the following `READ` statement:

  ```
  READ(5, "(2L8)")  L1, L2
  ```

  to read the following input record:

  ```
  .TRUE.bb.Falseb
  ```

  `L1` and `L2` will have the values true and false, respectively. The result would be the same if the input record were as follows:

TUESDAY*b*FRIDAY*bb*

On output:

- The output field consists of *w* `- 1` leading blanks, followed by `T` or `F`, for a true or false value, respectively.

  ```
  WRITE(6, "(2L7)")  L1, L2
  ```

  If `L1` and `L2` are true and false, respectively, the output record will be as follows:

  *bbbbbb*T*bbbbbb*F

2.8 Character editing

Character editing converts to or from a string of characters. The edit descriptors for character editing are as follows:

```
A[w]

Gw.d[Ee]
```

w, *d*, and *e* have the following meanings:

Descriptor	Specification
w	The width of the field
d	The number of fractional digits in the field. *d* is ignored when `G` is used for character editing.
e	The number of digits in the exponent. *e* is ignored when `G` is used for character editing.

For both input and output:

- *w* is the field width measured in characters.
- A `G`*w*`.`*d*[`E`*e*] general edit descriptor is the same as an `A`*w* edit descriptor for character data.
- If *w* is omitted from the `A` format, the length of the character data object being transferred is used as the field width.

On input:

- If *w* is greater than or equal to the length *len* of the character data read, *len* rightmost characters of the input field are read.
- If *w* is less than the length *len* of the character data read, the *w* characters of the character data will be read from the input field and placed left justified in the character list item followed by *len*-*w* trailing blanks.

On output:

- If *w* exceeds the length *len* of the character data written, *w*-*len* blank padding characters are written followed by *len* characters of the character data.
- If *w* is less than or equal to the length *len* of the character data written, the *w* leftmost characters of the character data will appear in the output field.

  ```
  CHARACTER(LEN = 14), PARAMETER :: &amp;
        SLOGAN = "SAVE THE RIVER"
  WRITE (*, "(A)")  SLOGAN
  ```

The preceding lines produce the following output record:

```
SAVE THE RIVER
```

2.9 Control edit descriptors

No data is transferred or converted with the control edit descriptors. Control edit descriptors affect skipping, tabbing, scale factors, and printing of optional signs. These edit descriptors may affect how the data is input or output using the subsequent data edit descriptors in the format specification.

2.9.1 Position editing

Position edit descriptors control relative tabbing left or right in the record before the next list item is processed. The edit descriptors for tabbing, where *n* is a positive integer, are as follows:

Descriptor	Meaning
T*n*	Tab to position *n*
TL*n*	Tab left *n* positions
TR*n*	Tab right *n* positions
*n*X	Tab right *n* positions

$	Carriage control

ANSI/ISO: The Fortran 90 standard does not specify the dollar sign ($) edit descriptor.

The tabbing operations to the left are limited by a position called the *left tabbing limit.* This position is normally the first position of the current record but, if the previous operation on the file was a nonadvancing formatted data transfer, the left tabbing limit is the current position within the record before the data transfer begins. If the file is positioned to another record during the data transfer, the left tabbing limit changes to the first position of the new record.

The T*n* edit descriptor positions the record just before the character in position *n* relative to the left tabbing limit. TR*n* and *n*X move right *n* characters from the current position. TL*n* moves left *n* characters from the current position, but is limited by the left tabbing limit.

For both input and output:

- *n* must be a positive integer constant with no kind parameter value specified for it.
- Left tabbing is always limited so that even if left tabbing specifies a position to the left of the left tabbing limit, the record position is set to the left tabbing limit in the record.
- The left tabbing limit in the record is determined by the position in the record before any data transfer begins for a particular data transfer statement.
- If a file is positioned to another record during a particular data transfer statement, the left tabbing limit is the first position of the record.

On input:

- The T descriptor might move the position within a record either left or right from the current position.
- Moving to a position left of the current position allows input to be processed twice, provided the same input statement is performing the processing for advancing input, or provided nonadvancing input is in effect.
- The X descriptor always moves the position to the right and skips characters.

On output:

- The positioning does not transmit characters, and does not by itself cause the record to be shorter or longer.
- Positions that are skipped and have not been filled previously behave as if they are blank filled.
- Positions previously filled can be replaced with new characters, but are not blank filled when they are skipped using any of the position edit descriptors.

For example, assume that `DISTANCE` and `VELOCITY` have the values `12.66` and `-8654.123`.

```
    PRINT 100, DISTANCE, VELOCITY
100 FORMAT (F9.2, 6X, F9.3)
```

The preceding code produces the following record:

bbbb`12.66`*bbbbbb*`-8654.123`

Continue to assume that `DISTANCE` and `VELOCITY` have the values `12.66` and `-8654.123`.

```
    PRINT 100, DISTANCE, VELOCITY
100 FORMAT(F9.2, T7, F9.3)
```

The preceding code produces the following record because `T7` specifies the first position for `VELOCITY` as the seventh character in the record:

bbbb`12-8654.123`

The dollar sign character (`$`) in a format specification modifies the carriage control specified by the first character of the record. In an output statement, the `$` descriptor suppresses the carriage return or line feed. In an input statement, the `$` descriptor is ignored. It is intended primarily for interactive I/O.

2.9.2 Slash editing

The slash edit descriptor consists of the single slash character (`/`). The current record is ended when a slash is encountered in a format specification.

On input:

- If the file is connected for sequential access, the file is positioned at the beginning of the next record. The effect is to skip the remainder of the current record.

- For direct access, the record number is increased by one, and the file is positioned at the beginning of the record with this increased record number, if it exists; it becomes the current record.
- A record can be skipped entirely on input.

On output:

- If the file is connected for sequential access, the current record is written. The current record can be an empty record.
- For direct access, the current record is blank filled, if necessary, the record number is increased by one, and this record becomes the current record.
- For an internal file that is a scalar, the current record is blank-filled, if necessary. No other action is taken until a check of the contents of the format is done; that is, the format determines what happens next.
- For an internal file that is an array, the current record is blank filled, and the file is positioned at the beginning of the next array element.

Assume in the following code that `ALTER`, `POSITION`, and `CHANGE` have the values `1.1`, `2.2`, and `3.3`, respectively:

```
PRINT "(F5.1, /, 2F6.1)", ALTER, POSITION, CHANGE
```

The preceding code produces the following two records:

```
bb1.1
bbb2.2bbb3.3
```

2.9.3 Colon editing

The colon edit descriptor consists of the character colon (`:`). If the list of items in a formatted `READ` or `WRITE` statement is exhausted, a colon stops format processing at that point. If the list is not exhausted, the colon edit descriptor has no effect.

For example, assume in the following code that `ALTER`, `POSITION`, and `CHANGE` have the values `1.1`, `2.2`, and `3.3`, respectively:

```
WRITE(6, 100)  ALTER, POSITION, CHANGE
100 FORMAT(3F5.2, :, "STOP")
```

The preceding code produces the following:

```
bb1.1bb2.2bb3.3
```

The characters `STOP` are not printed because the output list is exhausted when the colon edit descriptor is processed. If the colon edit descriptor were not present in the above format, the string `"STOP"` would be printed.

2.9.4 Sign editing

Sign editing applies to the output data transfer of positive integer and real values only. It controls the writing of the optional plus sign when the edit descriptor `I`, `F`, `E`, `EN`, `ES`, `D`, or `G` is used. The sign edit descriptors are as follows:

Descriptor	Meaning
`SP`	The plus sign is always written
`S`, `SS`	The plus sign is not written

The descriptors remain in effect until another sign edit descriptor is encountered in the format specification. The descriptors have no effect during formatted input data transfers.

For example, assume in the following code that `SPEED(1)` and `SPEED(2)` are `1.46` and `2.3412` respectively:

```
    WRITE(6, 110)  (SPEED(K), K = 1, 2)
110 FORMAT(SP, 2F10.2)
```

The preceding code produces the following:

bbbbb`+1.46`*bbb*`+234.12`

2.9.5 Scale factors

The *k* `P` edit descriptor indicates scaling, where the scale factor *k* is a signed integer literal constant.

The scale factor is zero at the beginning of a formatted I/O statement. When a *k*`P` descriptor occurs, the scale factor becomes *k*, and all succeeding numeric fields processed with an `F`, `E`, `EN`, `ES`, `D`, or `G` edit descriptor may be affected by this scale factor until another *k*`P` edit descriptor occurs.

On input:

- If the input field has no exponent, the scale factor effect is that the external number equals the internal number multiplied by a scale factor 10^k.

- The scale factor has no effect if the input field has an exponent.

Input example:

Assume that the input record contains `10.12`:

```
    READ (5,100) MASS
100 FORMAT (3PF15.3)
```

The preceding code gives `MASS` the value `10120.0`.

On output:

- For the `F` edit descriptor, the value is multiplied by 10^k.
- For the `E` and `D` edit descriptors, the significand of the value is multiplied by 10^k and the exponent is reduced by k.
- The `G` edit descriptor is not affected by the scale factor if the number will print correctly with the appropriate `F` edit descriptor as described in Table 14, page 110. Otherwise, the scale factor for the `G` edit descriptor has the same effect as for the `E` edit descriptor.
- `EN` and `ES` edit descriptors are not affected by a scale factor.

Assume that `TREE` has the value `12.96`:

```
    WRITE(6,200) TREE
200 FORMAT(2PG10.1)
```

The preceding code produces the following:

b`1296.E-02`

2.9.6 Blanks in numeric fields

Blanks other than leading blanks are ignored or interpreted as zero characters in numeric input fields as determined by the blank edit descriptors:

Descriptor	Meaning
`BN`	Treat nonleading blanks in numeric input fields as nonexistent
`BZ`	Treat nonleading blanks in numeric input fields as zeros

The interpretation is for input fields only when the field is processed using an `I`, `B`, `O`, `Z`, `F`, `E`, `EN`, `ES`, `D`, or `G` edit descriptor; output fields are not affected.

The BLANK= specifier in the OPEN statement affects the interpretation of blanks if no BN or BZ descriptor is used.

On input:

- The BLANK= specifier for an internal file is NULL.
- If the BLANK= specifier is NULL, or a BN edit descriptor is in effect, the nonleading blanks are ignored in succeeding numeric fields.
- If the BLANK= specifier is ZERO, or a BZ edit descriptor is in effect, the nonleading blanks are interpreted as zeros in succeeding numeric fields.
- The BN and BZ edit descriptors override the effect of the BLANK= specifier during the execution of a particular input data transfer statement.

Example:

```
    READ (5, 100)  N1, N2
100 FORMAT (I5, BZ, I5)
```

Considering the preceding code, if the input record is as follows, and unit 5 has been opened with a BLANK= specifier equal to NULL, the values assigned to N1 and N2 are 99 and 90909, respectively:

*b*9*b*9*b*9*b*9*b*9

2.10 Character string edit descriptors

Character string edit descriptors are used to transfer characters to an output record. They must not be used on input. The character string edit descriptors are apostrophe, quote, and Hollerith. They have the following format:

'*characters*'

"*characters*"

*c*H*characters*

On output:

- The apostrophe and quote edit descriptors have the form of literal character constants with no kind parameter values. They cause those constants to be placed in the output.

- The Hollerith descriptor *c*`H . . .` may be used to print the *c* characters following the `H`. *c* must be a positive integer and must not have a kind value specified for it.
- To write a quote in the output field when a quote is the delimiting character, use two consecutive quotes; to write an apostrophe in the output field when an apostrophe is the delimiting character, use two consecutive apostrophes.
- The field width is the length of the character constant, but doubled apostrophes or quotes count as one character.

For example, assume in the following code that `TEMP` has the value `32.120001`:

```
    WRITE (6, 120)  TEMP
120 FORMAT (’ TEMPERATURE = ’, F13.6)
```

The preceding code produces the following record:

b`TEMPERATURE`*b*`=`*bbbbb*`32.120001`

2.11 List-directed formatting

List-directed formatting is one of the implicit formatting methods in Fortran. Conversion from characters in `READ` statements to characters in `PRINT` and `WRITE` statements does not use an explicit format specification. The editing occurs based on the type of the list item. Data is separated by commas or blanks. The I/O statement uses an asterisk (*) instead of an explicit format specification, as follows:

```
READ (5, *)  HOT, COLD, WARM
```

For both input and output:

- A list-directed record consists of values and value separators.
- The values allowed in a list-directed input record are as follows:

Value	Content
null	A null value. Specified, for example, by two consecutive commas (`,,`).
c	Either 1) A literal constant or 2) a nondelimited character string with no embedded blanks. See Section 2.11.1.1, page 123, for the requirements for nondelimited character constants.

*r*c*	*r* repetitions of the constant *c*. *r* must be an unsigned, nonzero integer.
*r**	*r* repetitions of the null value, where *r* is nonzero. *r* must be an unsigned, nonzero integer.

Embedded blanks are allowed only within values as specified for each data type. For example, embedded blanks are allowed within a delimited character constant or at certain positions within a complex value.

- The value separators allowed in a list-directed input record are as follows:

Separator	Meaning
`,`	A comma, optionally preceded or followed by contiguous blanks.
`/`	A slash, optionally preceded or followed by contiguous blanks.
<blank>	One or more contiguous blanks between two nonblank values or following the last nonblank value, where a nonblank value is a constant, an *r*c* form, or an *r** form.

- List-directed formatting must not be specified for direct access or nonadvancing sequential data transfer.
- If there are no list items and there is no current record, an input record is skipped or an output record that is empty is written. If there are no list items and there is a current record, the current record is skipped (the file is positioned at the end of the current record) or the current record is terminated at the current position. Recall that a current record exists only if the previous I/O data transfer to the unit was nonadvancing. An empty record is either a blank-filled record or a record with no characters in it, depending on the file structure used. If you are using UNICOS or UNICOS/mk systems, see the *Application Programmer's I/O Guide*, publication SG–2168, for more information on file structures.
- The end of a record has the same effect as a blank, unless it occurs within a delimited character literal constant. Similarly, a sequence of two or more consecutive blanks is treated as a single blank.

2.11.1 List-directed input

Input values are generally accepted as list-directed input if they are the same as those required for explicit formatting with an edit descriptor. The exceptions are as follows:

- When the data list item is of type integer, the constant must be of a form suitable for the `I` edit descriptor. The CF90 and MIPSpro 7 Fortran 90 compilers permit binary, octal, and hexadecimal based values in a list-directed input record to correspond to `I` edit descriptors.

  ANSI/ISO: The Fortran 90 standard specifies that binary, octal, and hexadecimal values must not be used in a list-directed input record.

- When the data list item is of type real, the constant must be of a form suitable for the `F` edit descriptor. If no decimal point appears in the constant, the constant has no fractional digits specified for it.
- Blanks are never zeros.
- Embedded blanks are allowed within a delimited character constant. Values of type complex include the parentheses for a complex constant. Blanks may occur before or after the comma and before or after the parentheses. The following are input examples:

  ```
  "NICE DAY"
  (1.2, 5.666 )
  TODAY
  ```

- Logical items must not use value separators as the optional characters following the `T` or `F`. `TUESDAY` is allowed. Specifying `T,TOO` is not allowed because the value `TOO` will be used as the next input value.
- An end-of-record cannot occur within a constant, except a complex constant or a delimited character constant. For a complex constant, the end of record can occur between the real part and the comma, or between the comma and the imaginary part. For a character constant, the end-of-record can occur anywhere in the constant except between any consecutive (doubled) quotes or apostrophes in the constant. The end-of-record does not cause a blank or any other character to become part of the character value. A complex or character constant can be continued on as many records as needed.
- Value separators may appear in any delimited character constant. They are, however, not interpreted as value separators, but are characters in the delimited character constant.
- Assume that *len* is the length of the corresponding input list item and *w* is the number of effective characters in the character value. If $len \le w$, the leftmost *len* characters of the constant are used. If $len > w$, the *w* characters of the constant are left justified in the input list item and the list item is blank filled on the right.

For example, consider the following code:

```
CHARACTER (2) NAME
   . . .
READ (5,*)  NAME
```

Assume that the input record is as follows:

```
JONES
```

After the READ statement, the value in NAME is JO, because *len* (=2) is less than *w* (=5).

2.11.1.1 Requirements for nondelimited character strings as values

In certain cases, the delimiters are not required for character values on input. However, nondelimited character strings impose the following requirements:

- The corresponding data list item must be of type character.
- The character string must not contain any value separator.
- The character string must not be continued across a record boundary.
- The first nonblank character is neither an apostrophe (') nor a quote (").
- The leading characters are not a string of digits followed immediately by an asterisk.

In any of these cases, the character constant represented by the character string is terminated by the first value separator or end-of-record, and apostrophes (') and quotes (") are not doubled.

2.11.1.2 Null values

Null values are used to specify no change of the items in the input item list. Null values have the following forms:

- No value between separators, such as `,,`
- A nonblank value separator as the first entity in the record; for example, a record beginning with slash as the first nonblank character represents a null value, as in `/4.56`.
- *r** followed by a value separator as in the following:

  ```
  7*,'TODAY'
  ```

An end-of-record does not signify a null value.

The null value does not affect the definition status or value of the corresponding list item.

For a complex constant, the entire constant can be null, but not one of the parts.

If a slash terminates input, the remaining characters in the record are ignored, and the remaining list items are treated as though null values had been read. This applies to any remaining items in an implied-`DO` or to any remaining elements of an array. Example:

```
REAL AVERAGE(2)
READ (5, *) NUMBER, AVERAGE
```

If the input record is the following, the result is that `NUMBER = 6`, `AVERAGE(1)` is unchanged, and `AVERAGE(2) = 2.418`:

```
b6,,2.418
```

2.11.2 List-directed output

List-directed output uses similar conventions to those used for list-directed input. The following rules and restrictions apply to both input and output:

- Except for nondelimited character values, a comma, optionally preceded or followed by a blank, is used as a separator.
- New records are begun as needed, at any point in the list of output items. A new record does not begin in the middle of a value, except as noted below for character values. Each new record begins with a blank for carriage control, except for delimited character constants.
- Slashes and null values are not written.
- If two or more consecutive values from an array are identical, they can be written using the repeat factor *r***c*.

There are a few exceptions. These are noted as follows for each of the intrinsic types:

- Integer. The effect is as though an `I`*w* edit descriptor were used, using a suitable value for *w*.

- Real. The effect is as though an `0PF`*w*`.`*d* or an `1PE`*w*`.`*d*`E`*e* edit descriptor were used, using suitable values for *w*, *d*, and *e*. The edit descriptor chosen depends on the magnitude of the number written.
- Complex. The real and imaginary parts are enclosed in parentheses and separated by a comma. A complex value must fit within one record.
- Logical. List-directed output prints `T` or `F` depending on the value of the logical data item.
- Character. The form of the output for character values depends on the value of the `DELIM=` specifier in the `OPEN` statement for that unit.
  - The following are true if there is no `DELIM=` specifier, if the value of the `DELIM=` specifier is `NONE`, or if the file is an internal file:
    - Character values are not delimited
    - Character values are not surrounded by value separators
    - Only one quote or apostrophe is needed for each quote or apostrophe in the string transferred
    - A blank is inserted at the beginning of new records for a continued character value
  - The following are true if the `DELIM=` specifier is `QUOTE` or `APOSTROPHE`:
    - Character values are delimited with the specified delimiter
    - All values are surrounded by value separators
    - A character that is the same as the specified delimiter is doubled when written to the output record
    - No blank is inserted at the beginning of a continued record for carriage control in the case of a character value continued between records

List-directed output of real values uses either an `F` or an `E` format with a number of decimal digits of precision that assures full-precision printing of the real values. This allows later formatted or list-directed input of real values to result in the generation of a bit-identical binary floating-point representation, which allows a value to be written and reread without changing the stored value.

You can set the `LISTIO_PRECISION` environment variable to control the number of digits of precision printed by list-directed output. The following values can be assigned to `LISTIO_PRECISION`:

Value	Result
`FULL`	Prints full precision (default).
`PRECISION`	Prints *x* or *x* `+1` decimal digits, where *x* is value of the `PRECISION` intrinsic function for a given real value. This is a smaller number of digits, which usually ensures that the last decimal digit is accurate to within 1 unit. This number of digits is usually insufficient to assure that subsequent input will restore a bit-identical floating-point value.
`YMP80`	Causes list-directed and namelist output of real values to be of the format used in UNICOS 8.0 and previous library versions on CRAY Y-MP systems. The `YMP80` specification is available in UNICOS 8.0 libraries, but this functionality will be removed in a later UNICOS release.

ANSI/ISO: The Fortran 90 standard does not specify full-precision printing or environment variables.

Example:

```
REAL                     :: TEMPERATURE = -7.6
INTEGER                  :: COUNT = 3
CHARACTER(*), PARAMETER :: PHRASE = "This isn’t so"
OPEN(10, DELIM = ’NONE’)
WRITE(10, *)  TEMPERATURE, COUNT, PHRASE
```

The output record on unit 10 would be:

```
-7.6    3     This isn’t so
```

2.12 Namelist formatting

In some programs, it is convenient to create a list of variables that can be read or written by referencing the name of the list. The term *namelist* denotes this kind of data transfer. Before input or output can begin using this facility, the `NAMELIST` specification statement is used to define the list and give it a *group name.* In the following example, the `NAMELIST` statement defines the group name `MEETING` made up of the data objects `JAKE`, `JOE`, and `JANE`:

```
NAMELIST /MEETING/ JOE, JAKE, JANE
```

The namelist input and output records consist of an ampersand (&) followed by a namelist group name followed by a sequence of name-value pairs followed by a slash (`/`). A *name-value pair* is a name or a subobject designator, an equal sign, and one or more values separated by value separators; that is, *name*=*value* or *name*=*value*,*value*, ... The *name* in a name-value pair must appear in the `NAMELIST` statement specifying the namelist group name. The name-value pairs provide a convenient form of documentation.

For example, consider the following input data transfer statement:

```
READ (*, NML = MEETING)
```

The preceding statement sets the variables `JAKE` and `JOE` when the input record is as follows:

```
&MEETING  JAKE = 3505,  JOE = 1 /
```

It does not change the value of the variable `JANE`.

Namelist output is convenient for debugging or writing values in a form that can later be read by a `READ` statement referencing the same namelist group name. All variables in the namelist group name appear in the output record and they must all be defined when the output statement is executed. For example, consider the following `WRITE` statement:

```
WRITE (*, NML = MEETING)
```

The preceding statement creates the following output record (assuming `JANE` is defined with the value `0`):

```
&MEETING  JAKE = 3505, JOE = 1, JANE = 0 /
```

In namelist I/O records, blanks can appear before the ampersand, after the namelist group name, before or after the equal sign in the name-value pairs, or before the slash terminating the namelist records.

The rules and restrictions for a blank and an end-of-record in a character constant are the same as for delimited character constants in list-directed formatting. Nondelimited character strings are not permitted in namelist input records.

2.12.1 Namelist input

Namelist input consists of the following items, in order:

- An ampersand (&).
- The group name.
- One or more blanks.

- A sequence of zero or more name-value pairs whose names are in the namelist group and whose values are described in the following sections. These must be separated by value separators and followed by a slash, which terminates the namelist input.

The name-value pairs are separated by value separators that are composed of the following characters:

- A comma, optionally preceded or followed by contiguous blanks
- A slash, optionally preceded or followed by contiguous blanks
- One or more contiguous blanks between two name-value pairs

Blanks can precede the ampersand (&) or the slash (`/`).

When the name in the name-value pair is a subobject designator, it must not be a zero-sized array, zero-sized array section, or a zero-length character string.

A lowercase letter is the same as an uppercase letter and vice versa when used in the group name.

An example of namelist input is as follows:

```
READ (*, NML = MEETING)
```

The input record for the preceding statement might be as follows:

```
&MEETING JAKE = 3500, JOE = 100, JANE = 0/
```

2.12.1.1 Names in name-value pairs

This section describes the rules and restrictions for the names used in the name-value pairs in the namelist records.

- The name-value pairs can appear in any order in the input records.
- The name-value pairs are evaluated serially, in left-to-right order.
- A name in the namelist group can be omitted.
- Each name must correspond with a name in the designated namelist group; a component name, if any, must also be the name of a component of a structure named in the namelist group.
- Optionally-signed integer literal constants with no kind parameter values must be used in all expressions that appear in subscripts, section designators, or substring designators.

- The name of a structure or a subobject designator can be the name in a name-value pair.
- A name in an input record must not contain embedded blanks. A name in the name-value pair can be preceded or followed by one or more blanks.
- A lowercase letter is the same as an uppercase letter and vice versa when used in the name of a name-value pair.
- A namelist group object name or subobject designator can appear in more than one name-value pair in a sequence.

Recall that each name must not be the name of an array dummy argument with nonconstant bounds, an automatic object, a character variable with nonconstant length, a pointer, or a structure variable of a type with a component that is a pointer, or an allocatable array.

2.12.1.2 Values in name-value pairs

The value in a name-value pair must be in a form acceptable for a format specification for the type of the name, except for restrictions noted below.

Null values can have the following forms:

- No value between value separators
- No value between the equal sign and the first value separator
- The *r** form, followed by one or more blanks

Null values do not change the value of the named item or its definition status. An entire complex constant can be null; neither of the parts can be. The end of a record following a value separator does not specify a null value.

Each value is a null value or one of the following forms; in these forms, *r* is assumed to be a repeat factor and is nonzero:

Value	Representation
c	Indicates a literal constant, which must be unsigned
*r*c*	Indicates *r* successive literal constants *c*
*r**	Indicates *r* successive null values

The form of a value must be acceptable to a format specification for an entity of the type of the corresponding list item, except as noted below; for example, the value *c* corresponding to a list item of type real can be of the following forms:

```
1
1.0E0
```

The value *c* corresponding to a list item of type real cannot be of the following forms:

```
(1.0,0.0)
A0
1.0EN-3
2.2_QUAD
```

Blanks are never treated as zeros, and embedded blanks must not appear in numeric or logical constants. The exception is that a blank can appear as a character in a character constant, or preceding or following the real or imaginary parts of a complex constant.

The number of values following the equal sign must not be larger than the number of elements of the array when the name in the name-value pair is an array, or must not be larger than the ultimate number of components when the name in the name-value pair is that of a structure. Any array or component that is an array is filled in array element order.

Consider the following example:

```
TYPE PERSON
     INTEGER LEN
     CHARACTER(10) :: NAME
END TYPE PERSON
TYPE(PERSON) PRESIDENT, VICE_PRES
NAMELIST /PERSON_LIST/ PRESIDENT, VICE_PRES
READ (5, NML = PERSON_LIST)
```

The input record for the preceding code might be as follows:

```
&PERSON_LIST PRESIDENT%LEN=10,
PRESIDENT%NAME="WASHINGTON",
             VICE_PRES%LEN=5, VICE_PRES%NAME="ADAMS"/
```

If there are fewer values in the expanded sequence than array elements or structure components, null values are supplied for the missing values.

If a slash occurs in the input, it is as if null values were supplied for the remaining list items, and the namelist input data transfer is terminated. The remaining values after the slash within the current record are ignored. The namelist input record can be terminated by the characters `&END`.

An integer value is interpreted as if the data edit descriptor were `I`*w* for a suitable value of *w*.

A complex value consists of a pair of parentheses surrounding the real and imaginary parts, separated by a comma. Blanks can appear before and after these values.

A logical value must not contain slashes, commas, or blanks as part of the optional characters permitted for logical values. The required characters for logical values include the first few characters, such as the initial:

- `.T`
- `.F`
- `T`
- `F`

The optional characters on logical values, which include `ALSE.`, `RUE.`, and so on, cannot include slashes, commas, or blanks.

A character literal constant may contain slashes, commas, or blanks as part of the constant. On namelist input, the `DELIM=` specifier is ignored.

2.12.1.3 Blanks

Blanks are part of the value separator except when used in the following situations:

- In a character constant
- Before or after the parts of a complex constant
- Before or after an equal sign, unless after the equal sign the blanks are followed immediately by a comma or slash
- Before the ampersand indicating the namelist group name and after the namelist group name

2.12.1.4 Use of namelist input

Namelist input requires the namelist group name, preceded by an ampersand, to be on the first nonblank record read by the namelist `READ` statement.

The following is an example of namelist input:

```
REAL A(3), B(3)
CHARACTER(LEN = 3) CHAR
COMPLEX X
LOGICAL LL
NAMELIST /TOKEN/ I, A, CHAR, X, LL, B
READ (*, NML = TOKEN)
```

Assume that the input record for the preceding is as follows:

```
&TOKEN A(1:2) = 2*1.0  CHAR = "NOP" B = ,3.13,,
       X = (2.4,0.0)  LL = T /
```

The results of the `READ` statement are as follows:

Name	Value
`I`	Unchanged
`A(1)`	1.0
`A(2)`	1.0
`A(3)`	Unchanged
`B(1)`	Unchanged
`B(2)`	3.13
`B(3)`	Unchanged
`CHAR`	`"NOP"`
`X`	(2.4, 0.0)
`LL`	True

2.12.2 Namelist output

Value separators can be blanks, commas, or a combination of blanks and commas. A new record can begin anywhere, except within a name or value, unless the value is a character constant or a complex constant; a record can begin anywhere within a character constant, or can begin before or after the comma, or left or right parenthesis of a complex constant. A blank can occur anywhere, except in a name or a noncharacter value.

2.12.2.1 Use of namelist output

A number of rules, similar to those for list-directed formatting, apply for namelist output. They are as follows:

- Namelist output consists of a series of records. The first nonblank record begins with an ampersand, followed by the namelist group name, followed by a sequence of name-value pairs, one pair for each variable name in the namelist group object list of the `NAMELIST` statement, and ends with a slash.
- A logical value is either `T` or `F`.
- An integer value is one that would be produced by an `I` *w* edit descriptor using a suitable value of *w*.
- For real output, the rules for list-directed output are followed using reasonable values for the *w*, *e*, and *d* that appear in real data edit descriptors and are appropriate for the output value.
- Parentheses enclose the value of a complex constant, and the parts are separated by a comma. A complex value must fit within one record. Blanks can be embedded after the comma and before the end of the record.
- Character constants follow the rules for list-directed output.
- Repeat factors of the form *r***c* are allowed on output for successive identical values.
- An output record does not contain null values.
- Each record begins with a blank for carriage control.
- No values are written for zero-sized arrays.
- For zero-length character strings, the name is written, followed by an equal sign, followed by a zero-length character string, and followed by a value separator or slash.

```
       NAMELIST /CALC/ DEPTH, PRESSURE
       DIMENSION DEPTH(3), PRESSURE(3)
       FLOAT1 = 0.2
       FLOAT2 = 3.0
       DO 10 I=1,3
         DEPTH(I) = FLOAT1 + FLOAT(I)
         PRESSURE(I) = FLOAT2 + (FLOAT(I-1) * 0.1)
10     CONTINUE
       WRITE(6,NML=CALC)
       END
```

This program produces the following output:

```
&CALC  DEPTH = 1.2, 2.2, 3.2,  PRESSURE = 3., 3.1, 3.2 /
```

2.12.2.2 DELIM= specifier for character constants

The form of the output for character values depends on the value of the DELIM= specifier in the OPEN statement for that unit.

If there is no DELIM= specifier or if the value of the DELIM= specifier is NONE, the following actions occur:

- Character values are not delimited.
- Character values are not surrounded by value separators.
- Only one quote or apostrophe is needed for each quote or apostrophe in the string transferred.
- A blank is inserted in new records for a continued character value to allow for carriage control.

If the DELIM= specifier is QUOTE or APOSTROPHE, the following actions occur:

- Character values are delimited with the specified delimiter.
- A character that is the same as the specified delimiter is doubled when written to the output record.

For example, in the following code, assume LEFT and RIGHT with values "SOUTH" and "NORTH". The program has a DELIM= specifier of QUOTE for unit 10:

```
CHARACTER(5) LEFT, RIGHT
NAMELIST /TURN/ LEFT,RIGHT
LEFT = ’SOUTH’
RIGHT = ’NORTH’
WRITE(10,NML=TURN)
END
```

The preceding code produces the following output record:

```
&TURN  LEFT = SOUTH, RIGHT = NORTH /
```

Note that if the DELIM= specifier is NONE in an OPEN statement, namelist output might not be usable as namelist input when character values are transferred, because namelist input of character values requires delimited character values.

2.12.3 Namelist distinctions

The CF90 and MIPSpro 7 Fortran 90 compilers have extended the namelist feature. The following additional rules govern namelist processing:

- An ampersand (&) or dollar sign ($) can precede the namelist group name or terminate namelist group input. If an ampersand precedes the namelist group name, either the slash (/) or the ampersand must terminate the namelist group input. If the dollar sign precedes the namelist group name, either the slash or the dollar sign must terminate the namelist group input.
- Octal and hexadecimal constants are allowed as input to integer and single-precision real namelist group items. An error is generated if octal and hexadecimal constants are specified as input to character, complex, or double-precision real namelist group items.

  Octal constants must be of the following form:

  – `O"123"`

  – `O'123'`

  – `o"123"`

  – `o'123'`

  Hexadecimal constants must be of the following form:

  – `Z"1a3"`

  – `Z'1a3'`

  – `z"1a3"`

  – `z'1a3'`

ANSI/ISO: The Fortran 90 standard does not specify the namelist distinctions described in this section.

Program Units [3]

There are several kinds of executable and nonexecutable program units in Fortran. Each of these program units provides unique functionality. The executable program units are the main program and procedure subprograms. The nonexecutable program units are block data units and modules, which provide definitions used by other program units. This chapter describes each of these units as well as the closely related concepts of host association and use association.

3.1 Overview

A Fortran program is a collection of program units. One and only one of these units must be a main program. In all but the simplest programs, the individual tasks are typically organized into a collection of function and subroutine subprograms and module program units. The program can be organized so that the main program drives (or manages) the collection of program units making up the executable program, but other program organizations can work as well.

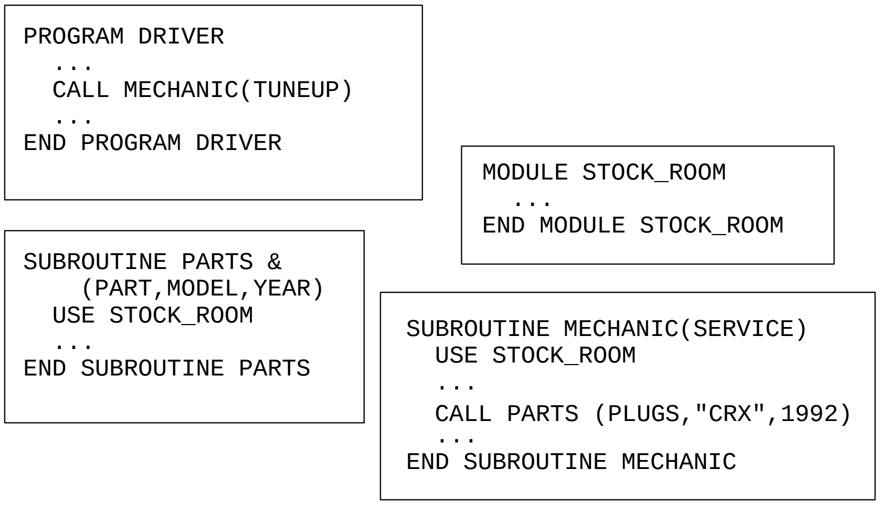
Each program unit is an ordered set of constructs and statements. The heading statement identifies the kind of program unit it is, such as a subroutine or a module; it is optional in a main program. An ending statement marks the end of the unit. The principal kinds of program units are as follows:

- Main program
- External function subprogram
- External subroutine subprogram
- Module program unit
- Block data program unit

The module program unit is new in Fortran 90 and is intended to help organize elements of the program. A module itself is not executable, but it can contain data declarations, derived-type definitions, procedure interface information, and subprogram definitions used by other program units. Block data program units are also nonexecutable and are used only to specify initial values for variables in named common blocks.

Program execution begins with the first executable statement in the main program. The *Fortran Language Reference Manual, Volume 1*, publication SR–3902, explains the high-level syntax of Fortran and how to put a Fortran program together. It is a good place to review the ways statements can be combined to form a program unit.

The Fortran program in Figure 6 is an example of a program that contains four program units: a main program, a module, and two subroutines.



```
PROGRAM DRIVER
  ...
  CALL MECHANIC(TUNEUP)
  ...
END PROGRAM DRIVER
```

```
MODULE STOCK_ROOM
  ...
END MODULE STOCK_ROOM
```

```
SUBROUTINE PARTS &
    (PART,MODEL,YEAR)
  USE STOCK_ROOM
  ...
END SUBROUTINE PARTS
```

```
SUBROUTINE MECHANIC(SERVICE)
  USE STOCK_ROOM
  ...
  CALL PARTS (PLUGS,"CRX",1992)
  ...
END SUBROUTINE MECHANIC
```

a10755

Figure 6. Four program units

Module STOCK_ROOM contains data and procedure information used by subroutines MECHANIC and PARTS. The main program DRIVER invokes the task represented by subroutine MECHANIC, but DRIVER does not itself need the information in STOCK_ROOM.

3.2 Main program

The main program specifies the overall logic of a Fortran program and is where execution of the program begins. A main program is similar to the other program units, particularly external subprograms, and has the following principal parts:

- The specification part, which defines the data environment of the program
- The execution part, in which execution begins and program logic is described

- The internal procedure part, which is included if the main program contains internal procedures

The principal ways of stopping the execution are as follows:

- Executing a `STOP` statement anywhere in the program (in any program unit in the program)
- Reaching the end of the main program

3.2.1 Main program organization

A main program is defined as follows:

R1101	*main_program*	**is**	[*program_stmt*] [*specification_part*] [*execution_part*] [*internal_subprogram_part*] *end_program_stmt*
R1102	*program_stmt*	**is**	`PROGRAM` *program_name* [(*arg_list*)]
EXT	*arg*	**or**	Any character in the CF90 character set. The CF90 compiler ignores any *arg*s specified after *program_name*.
R1103	*end_program_stmt*	**is**	`END` [`PROGRAM` [*program_name*]]

ANSI/ISO: The Fortran 90 standard does not specify the use of a parenthesized list of *args* at the end of a `PROGRAM` statement.

The preceding BNF definition results in the following general format to use for constructing a main program:

```
[ PROGRAM program_name ]
   [ specification_part ]
   [ execution_part ]
   [ internal_subprogram_part ]
   END [ PROGRAM [ program_name ] ]
```

The simplest of all programs is as follows:

```
END
```

The following simple program is more interesting:

```
PROGRAM  SIMPLE
   PRINT *, 'Hello, world.'
END
```

The PROGRAM statement is optional for a main program.

The program name on the END statement, if present, must be the same as the name on the PROGRAM statement and must be preceded by the keyword PROGRAM.

A main program has no provisions for dummy arguments.

A main program must not be referenced anywhere; that is, a main program must not be recursive (either directly or indirectly).

A main program must not contain RETURN or ENTRY statements, but internal procedures in a main program can have RETURN statements.

3.2.2 The specification part

The principal purpose of the specification part is to describe the nature of the data environment of the program: the arrays, types and attributes of variables, initial values, and so forth. The complete list of specification part statements is given in the *Fortran Language Reference Manual, Volume 1*, publication SR–3902.

A summary of the specification statements that are valid in a main program (R204) is as follows. The statements that are new in Fortran 90 appear in the following list with a note about their usage.

- ALLOCATABLE. The ALLOCATABLE attribute and statement, which are new in Fortran 90, specify dynamic arrays.
- AUTOMATIC.
- COMMON.
- DATA.
- DIMENSION.
- EQUIVALENCE.
- EXTERNAL.
- FORMAT.

- IMPLICIT.
- INTRINSIC.
- NAMELIST.
- PARAMETER.
- POINTER. The POINTER attribute and statement, which are new in Fortran 90, specify dynamic objects. The Cray pointer is an extension to Fortran 90 and is different from the Fortran 90 pointer. They both use the POINTER keyword, but they are specified such that the compiler can differentiate them.
- SAVE.
- TARGET. The TARGET attribute and statement, which are new in Fortran 90, specify a target for a Fortran 90 pointer.
- USE. USE statements, which are new in Fortran 90, provide access to entities packaged in modules.
- Derived-type definitions and declarations are new in Fortran 90.
- Interface blocks. Procedure interface blocks, which are new in Fortran 90, make procedure interfaces explicit.
- Statement function statement.
- Type declaration statement.

The entity-oriented style of declaration is new in Fortran 90. In these, all attributes of an entity may be declared in the same statement.

OPTIONAL and INTENT attributes or statements cannot appear in the specification part of a main program; they are applicable only to dummy arguments.

The accessibility attributes or statements, PUBLIC and PRIVATE, cannot appear in a main program; they are applicable only within modules.

Automatic objects are not permitted in main programs.

The SAVE attribute or statement can appear in a program, but it has no effect in a main program.

3.2.3 Execution part

The complete list of execution part statements (R208) is given in the *Fortran Language Reference Manual, Volume 1*, publication SR–3902. The following list includes the statements that are valid in a main program:

- `ALLOCATE` and `DEALLOCATE`. These statements, which are new in Fortran 90, are used for dynamically allocatable objects.
- `ASSIGN`.
- Assignment statement.
- Pointer assignment statement. New in Fortran 90. This feature should not be confused with Cray pointers and Cray character pointers.
- `BACKSPACE`.
- `BUFFER IN` and `BUFFER OUT`. These are extensions to Fortran 90.
- `CALL`.
- `CASE` construct. The `CASE` control construct, which includes the `CASE` statement, is new in Fortran 90.
- `CLOSE`.
- `CONTINUE`.
- `CYCLE`. New in Fortran 90.
- `DATA`.
- `DO` construct. Some forms of the `DO` construct are new in Fortran 90.
- `END`.
- `ENDFILE`.
- `ENTRY`.
- `EXIT`. New in Fortran 90.
- `FORMAT`.
- Assigned `GO TO`.
- Computed `GO TO`.
- `GO TO`.

- Arithmetic `IF`.
- `IF` statement.
- `IF` construct.
- `INQUIRE`.
- `NULLIFY`. New in Fortran 90.
- `OPEN`.
- `PAUSE`.
- `PRINT`.
- `READ`.
- `REWIND`.
- `STOP`.
- `WHERE` statement and `WHERE` construct. New in Fortran 90.
- `WRITE`.

3.2.4 The internal subprogram part

A set of internal procedures comprises the internal subprogram part. Internal procedures are described in the following section.

3.3 Internal procedures

Internal procedures are very much like external procedures, except that they are packaged inside a main program or other procedure subprograms. This makes their names local, rather than global like external procedures, so an internal procedure can be referenced only within the program unit that contains its definition.

The format of the internal procedure part (R210) of the host is as follows:

```
CONTAINS
    internal_subprogram
    [ internal_subprogram ] ...
```

Each internal procedure is either a function (R1215) or subroutine (R1219), as follows:

```
function_statement
   [ specification_part ]
   [ execution_part ]
   END FUNCTION  [ function_name ]
```

```
subroutine_statement
   [ specification_part ]
   [ execution_part ]
   END SUBROUTINE [ subroutine_name ]
```

The following is an example of an internal procedure:

```
PROGRAM WEATHER
   . . .
CONTAINS
   FUNCTION STORM(CLOUD)
      . . .
   END FUNCTION STORM
END
```

Internal procedures must not themselves contain internal procedures; that is, internal procedures must not be nested.

Internal procedures must not contain ENTRY statements.

Internal procedures must not contain PUBLIC or PRIVATE attributes or statements.

Internal procedures must not be passed as actual arguments.

The specification part of an internal procedure may contain the same statements as the specification part of a main program, plus the INTENT statement and the OPTIONAL statement.

The execution part of an internal procedure may contain the same statements as the execution part of a main program, plus the RETURN statement.

There must be at least one internal subprogram after the CONTAINS statement.

An internal procedure can be referenced in the execution part of its host (for example, the main program that contains it) and in the execution part of any internal procedure contained in the same host. This includes itself, so internal procedures may be referenced recursively, either directly or indirectly.

An internal procedure name is a local name in its host and therefore is subject to the rules governing such names. An internal procedure name has the following characteristics:

- It gives the internal procedure precedence over any external procedure or intrinsic procedure with the same name
- It must be different from the names of other internal procedures in the same host and different from the imported names of any module procedures either imported into the host or into the internal procedure itself
- It must be different from any other local name in the host or itself and from names made accessible by a USE statement

The next section describes the rules governing other names that appear in the host and/or the internal procedure. The host association rules apply to a module procedure and its host module, as well as to an internal procedure and its host (such as a main program).

3.4 Host association

The program unit containing an internal procedure is called the *host* of the internal procedure. The program unit (which must be a module) that contains a module procedure is called the *host* of the module procedure. As shown in Figure 7, page 146, an important property of internal and module procedures is that the data environment of the host is available to the procedure. When data in the host are available within the contained procedure, they are said to be accessible by *host association*. Because the internal (or module) procedure also has a local data environment, rules are needed to determine whether a given reference inside that procedure identifies a host entity or one local to the procedure.

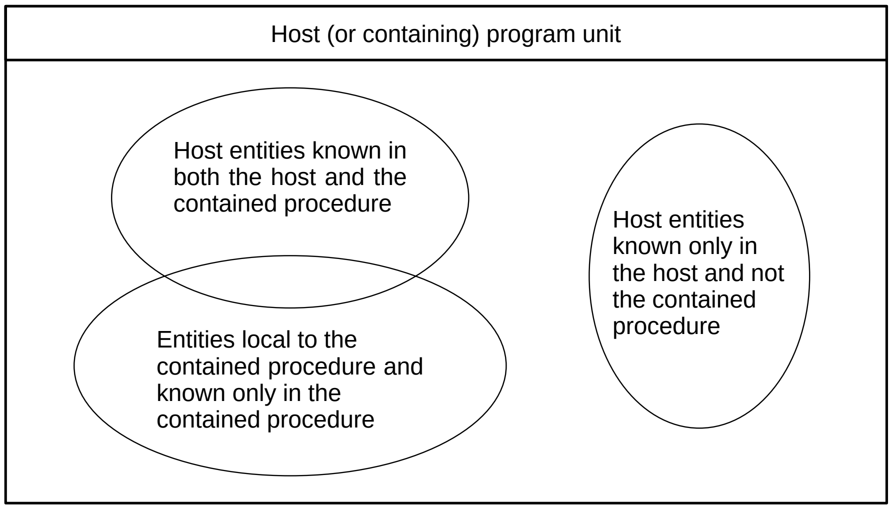

Host (or containing) program unit
Host entities known in both the host and the contained procedure
Entities local to the contained procedure and known only in the contained procedure
Host entities known only in the host and not the contained procedure
a10756

Figure 7. Host association

In a language in which the attributes of all entities must be declared explicitly, local declarations typically override host declarations. The host declarations that are not overridden are available in the contained procedure. Fundamentally these are the rules used in Fortran 90, and this situation can be simulated by using `IMPLICIT NONE` in both the host and the contained procedure. `IMPLICIT NONE` forces explicit declaration of all entities.

However, Fortran allows implicit declarations (the use of an entity name in the execution part without an explicit declaration of that name in the specification part). Suppose the variable `TOTAL` is referenced in an internal procedure, and neither the internal procedure nor its host explicitly declares `TOTAL`. Is `TOTAL` a host or local entity? Or, suppose that `TOTAL` is used in two internal procedures in the same host, without declaration anywhere. Are they (it) the same `TOTAL`? The possibilities are shown in Figure 8, page 147.

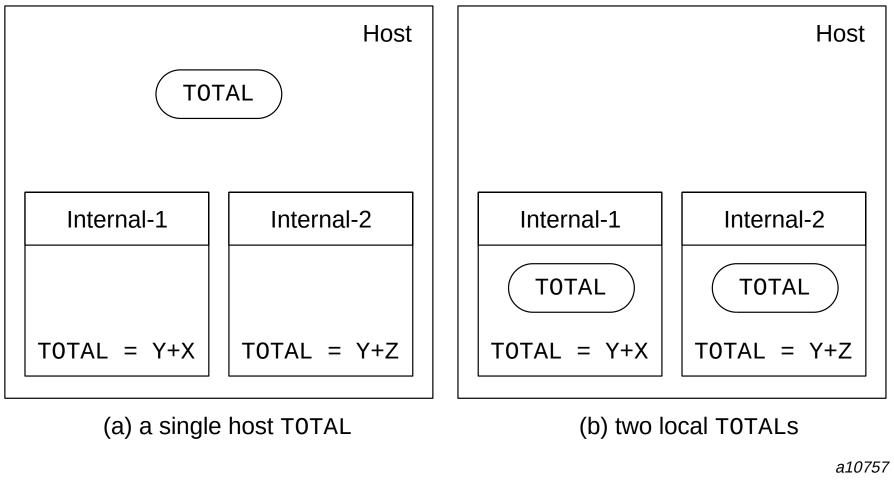


Figure 8. Is there one `TOTAL` in the host or two local `TOTAL`s?

The answer to both of these questions is case (b) in Figure 8, unless `TOTAL` is also referenced in the host, in which case (a) applies. If `TOTAL` is referenced in the host, it becomes declared implicitly there and is therefore a host entity. In this case, any internal procedure use of `TOTAL` accesses the host entity. The situation is the same (`TOTAL` is a host entity) if it is declared but not referenced in the host and not declared in the internal procedure. If `TOTAL` is declared in the internal procedure, then case (b) applies (`TOTAL` is local) regardless of whether `TOTAL` is declared or referenced in the host.

Implicit declarations are governed by the implicit typing rules and the use of the `IMPLICIT` statement. The rules governing implicit typing in hosts and contained procedures are given in the *Fortran Language Reference Manual, Volume 1*, publication SR–3902, and the rules governing host association are given in Section 6.2.1.3, page 275, page Section 6.2.1.3, page 275. These rules are combined and summarized in the following paragraphs.

A name is local if it is declared explicitly in the contained procedure, regardless of any declarations in the host. A dummy argument in a contained procedure is an explicit local declaration, even though the name can be implicitly typed. A dummy argument, if not explicitly typed, is typed according to the implicit typing rules of the contained procedure.

An entity not declared explicitly in a contained procedure is nevertheless local (via implicit declaration) if and only if it is neither explicitly nor implicitly declared in the host.

If it is not local, based on the previous paragraphs, the entity is host associated.

The default implicit rules (the implicit typing rules in the absence of IMPLICIT statements) in a contained procedure are the implicit typing rules of the host, as established by the default implicit rules in the host and modified by any IMPLICIT statements in the host.

IMPLICIT statements in the contained procedure, if any, modify the implicit typing rules inherited from the host. Note that these modified rules apply to implicitly typed dummy arguments of the contained procedure.

The following is a summary of the implicit typing rules:

host implicit typing rules	=	host default implicit rules	+	host IMPLICIT statements
contained procedure typing rules	=	host implicit typing rules	+	contained procedure IMPLICIT statements

In the expression X = A+B+P+Q+Y in the following program example, the operands in the expression are declared in different scoping units (see Figure 9, page 150):

```
PROGRAM HOST
   USE GLOBAL_DATA      ! Accesses integer X and real Y.
   IMPLICIT LOGICAL (E-J)
   ! implicit typing: A-D real
   !                  E-J logical
   !                  K-N integer
   !                  O-Z real
   REAL  A, B
      . . .
   READ *, P            ! This reference declares P
                        ! implicitly here.
      . . .
   CALL CALC(Z)         ! This reference also implicitly
      . . .             ! declares Z here.
CONTAINS
      . . .
   ! X declared explicitly in internal procedure CALC.
   SUBROUTINE CALC(X)
      IMPLICIT REAL(G-I)
```

```
         ! implicit typing: A-D real
         !                  E-F logical
         !                  G-I real
         !                    J logical
         !                  K-N integer
         !                  O-Z real
         REAL  B
            . . .
         X = A + B + P + Q + Y
         ! In subroutine CALC (all are type real):
         !    X is local (dummy argument)
         !    A is host associated
         !    B is local (explicitly declared)
         !    P is host associated
         !    Q is local (implicitly declared)
         !    Y host associated with Y in HOST,
         !      and that Y is is use associated (from the module)
            . . .
      END SUBROUTINE CALC
      . . .
   END PROGRAM HOST
```

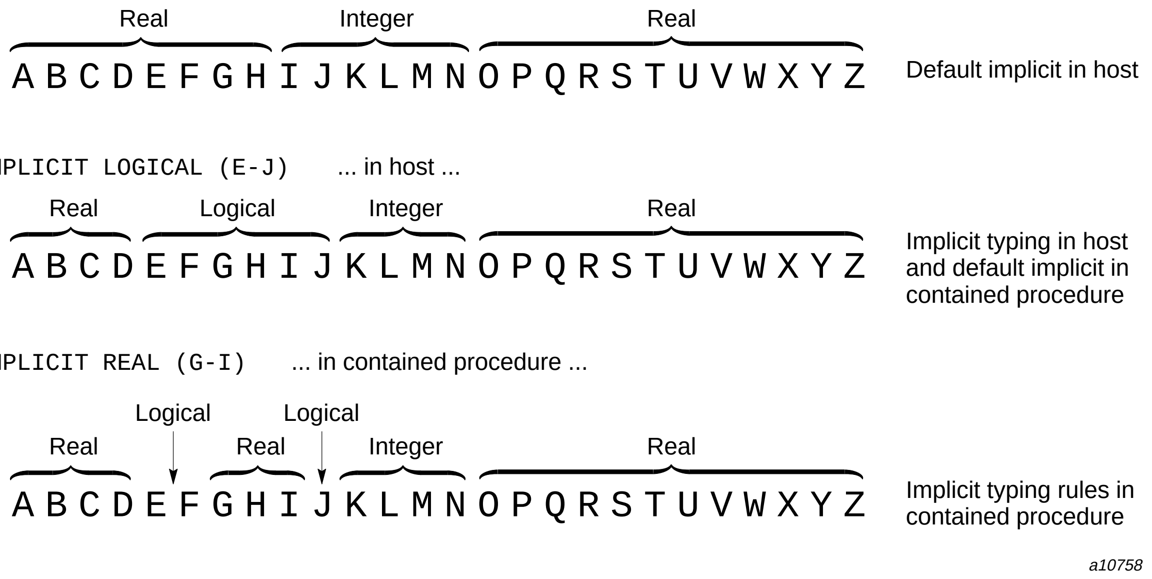

Real
Integer
Real
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Default implicit in host
IMPLICIT LOGICAL (E-J)
... in host ...
Real
Logical
Integer
Real
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Implicit typing in host and default implicit in contained procedure
IMPLICIT REAL (G-I)
... in contained procedure ...
Logical
Logical
Real
Real
Integer
Real
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Implicit typing rules in contained procedure
a10758

Figure 9. How the mapping of implicit typing progresses from host to contained procedure

A particularly interesting case of the host associated implicit rules is when the host has `IMPLICIT NONE`. With `IMPLICIT NONE`, no other implicit statements are allowed in that scoping unit, and explicit typing is required for all data objects in the host. `IMPLICIT NONE` is therefore the default in the contained procedure, although this can be modified by `IMPLICIT` statements in the contained procedure. This can result in some of the letters having implicit types in the contained procedure and some not. For example, suppose that the host has `IMPLICIT NONE` and the contained procedure has the following `IMPLICIT` statements:

```
IMPLICIT COMPLEX (C, Z)
IMPLICIT LOGICAL (J-L)
```

Data objects in the contained procedure with names starting with `C` or `Z` may be declared implicitly of type complex; data objects with names starting with `J`, `K`, or `L` may be declared implicitly of type logical. `IMPLICIT NONE` continues to apply to letters `A-B`, `D-I`, and `M-Y`, and data object names beginning with these letters must be explicitly declared.

3.5 External subprograms

External subprograms are global to the Fortran program; they can be referenced or called anywhere. An internal procedure, on the other hand, is known only within its host.

The major difference between external procedures and internal (and module) procedures is not syntactic; it is the fact that an external procedure interface is not known at the point of procedure reference. Also, internal (and module) procedures are compiled with their hosts, whereas external procedures usually are compiled separately. In these respects external procedures are the same as in the FORTRAN 77 standard. For internal (and module) procedures, on the other hand, interface information is available at the point of procedure reference. This is a very significant difference and a major practical advantage of internal and module procedures; Section 4.8, page 221, describes the benefits of explicit interfaces, which come automatically with internal and module procedures, but must be provided for external procedures.

Another difference between internal and external procedures is that external procedures can contain internal procedures; internal procedures cannot.

The organization of external subprograms is very much like that of main programs. As the following formats show, external subprograms (R203) are of two types, functions (R1215) and subroutines (R1219):

```
function_statement
   [ specification_part ]
   [ execution_part ]
   [ internal_subprogram_part ]
   END [ FUNCTION [ function_name ] ]
```

```
subroutine_statement
   [ specification_part ]
   [ execution_part ]
   [ internal_subprogram_part ]
   END  [ SUBROUTINE [ subroutine_name ] ]
```

The following are examples of external procedures:

```
FUNCTION FOOTBALL(GAME)
   INTEGER FOOTBALL
   FOOTBALL = N_PLAYERS
      . . .
```

```
END FUNCTION FOOTBALL
SUBROUTINE SATURDAY(SEVEN)
   X = . . .
END
```

Unlike the main program, the program unit heading (FUNCTION or SUBROUTINE statement) is required in an external subprogram. For more information on the FUNCTION statement, see Section 4.3.1, page 183. For more information on the SUBROUTINE statement, see Section 4.2.1, page 178.

The procedure name, if present on the END statement, must be the same as that in the heading statement.

OPTIONAL and INTENT attributes and statements for dummy arguments are allowed in the specification part of an external subprogram, but only for dummy arguments.

The specification and execution parts of an external subprogram can contain ENTRY statements and the execution part may contain RETURN statements.

External subprograms must not contain PUBLIC or PRIVATE attributes or statements.

External procedures can be directly or indirectly recursive, in which case the RECURSIVE keyword is required on the heading statement.

An external subprogram is the host to any internal procedures defined within it.

An external procedure name can be used as an actual argument in a procedure reference, corresponding to a dummy procedure argument in the procedure referenced.

Procedures, including internal, external, and module procedures, are described in detail in section Chapter 4, page 171.

3.6 Modules

The module program unit is a new feature in Fortran 90. A module allows you to package data specifications and procedures in one place for use in any computational task in the program.

Anything required by more than one program unit can be packaged in modules and made available where needed. A module is not itself executable, though the procedures it contains can be individually referenced in the execution part of other program units. The number of modules is not restricted, and a module may use any number of other modules as long as the access path does not lead back to itself. Modules are powerful tools for managing program organization and simplifying program design.

3.6.1 Module organization

A module is defined as follows:

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

The preceding BNF definition results in the following general format to use for constructing a module:

```
MODULE module_name
   [ specification_part ]
   [ module_subprogram_part ]
   END [ MODULE [ module_name ] ]
```

The module name, if present on the `END` statement, must be the same as on the `MODULE` statement.

3.6.2 The specification part

The form of the specification part (R204) of a module is similar to that for other program units. The statements it can contain are as follows:

- `ALLOCATABLE`
- `COMMON`
- `DATA`
- `DIMENSION`

- EQUIVALENCE
- EXTERNAL
- IMPLICIT
- INTRINSIC
- NAMELIST
- PARAMETER
- POINTER
- PRIVATE
- PUBLIC
- SAVE
- TARGET
- USE
- Derived-type definition statements
- Interface blocks
- Type declaration statements

The following paragraphs describe the rules and restrictions that apply to the specification part of a module. The specification parts of the module procedures have the same rules as those for external procedures, which are described in the previous section.

The following types of statements are not allowed: OPTIONAL or INTENT attributes or statements, ENTRY statements, FORMAT statements, automatic objects, and statement function statements.

PUBLIC and PRIVATE attributes and statements are allowed.

The SAVE attribute and statement can be used in the specification part of a module to ensure that module data object values remain intact. Without SAVE, module data objects remain defined as long as any program unit using the module has initiated, but not yet completed, execution. However, when all such program units become inactive, any data objects in the module not having the SAVE attribute become undefined. SAVE can be used to specify that module objects continue to be defined under these conditions.

The following is an example of a simple module for providing global data:

```
! This module declares three scalar variables (A,
! KA, and X) and two arrays (Y and Z).  X is given
! an initial value.  These five variables can be
! considered to be global variables that can
! selectively be made available to other program! units.
!
MODULE T_DATA
   INTEGER  ::  A, KA
   REAL     ::  X = 7.14
   REAL     ::  Y(10,10), Z(20,20)
END MODULE T_DATA
!
! The USE statement makes A, KA, X, Y, and Z
! available to subroutine TASK_2!
SUBROUTINE TASK_2
   USE T_DATA
   . . .
END SUBROUTINE TASK_2
```

3.6.3 Module subprogram part

The module subprogram part is similar to the internal procedure part of a main program or external subprogram. It is a collection of procedures local to the module and sharing its data environment through host association. The two principal differences between module subprograms and internal subprograms are as follows:

- The organization, rules, and restrictions of module procedures are those of external procedures rather than internal procedures. For example, module procedures can contain internal procedures.
- Module procedures are not strictly local to the host module, nor are they global to the program. Only program units using the module can access the module's procedures not specified to be PRIVATE.

The form of the module subprogram part (R212) is as follows:

CONTAINS
   module_subprogram
   [*module_subprogram*] ...

There must be at least one module subprogram after the CONTAINS statement.

Each module subprogram is a function (R1215) or subroutine (R1219) and has one of the following formats:

```
function_statement
    [ specification_part ]
    [ execution_part ]
    [ internal_subprogram_part ]
    END FUNCTION [function_name ]
```

```
subroutine_statement
    [ specification_part ]
    [ execution_part ]
    [ internal_subprogram_part ]
    END SUBROUTINE [ subroutine_name ]
```

An example of a module procedure is as follows:

```
MODULE INTERNAL
   . . .
CONTAINS
   FUNCTION SET_INTERNAL(KEY)
      . . .
   END FUNCTION
END
```

The rules for host association and implicit typing in a module procedure are the same as those described for internal procedures in Section 3.4, page 145. A module procedure acquires access to entities in its host module through host association, but not to entities in a program unit that uses the module.

3.6.4 Using modules

A program unit can use the specifications and definitions in a module by referencing (using) the module. This is accomplished with a USE statement in the program unit requiring access to the specifications and definitions of that module. Such access causes an association between named objects in the module and the using program unit and is called *use association*. USE statements must immediately follow the program unit heading.

Each entity in a module has the PUBLIC or PRIVATE attribute, which determines the accessibility of that entity in a program unit using the module. A PRIVATE entity is not accessible (that is, it is hidden) from program units using the module. A PUBLIC entity is accessible, although its accessibility may be further limited by the USE statement itself. Figure 10 depicts these phenomena:

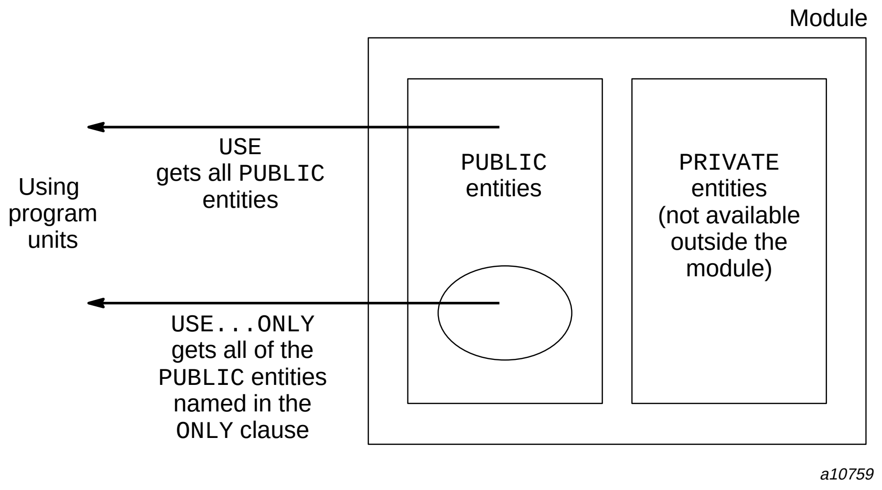


Figure 10. Public and private entities in a module

3.6.4.1 Accessing all public entities in a module

The USE statement gives the program unit access to public entities in the module and is defined as follows:

R1107	*use_stmt*	**is**	USE *module_name* [, *rename_list*]
		or	USE *module_name*, ONLY: [*only_list*]
R1108	*rename*	**is**	*local_name* => *use_name*
R1109	*only*	**is**	*access_id*
		or	[*local_name* =>] *use_name*
R522	*access_id*	**is**	*use_name*
		or	*generic_spec*

Specifying a USE statement with a *rename_list* allows any of the public entities in the module to be renamed to avoid name conflicts or to blend with the readability flavor in the using program unit.

Examples:

```
USE FOURIER
USE S_LIB, PRESSURE => X_PRES
```

With both USE statements in the preceding example, all public entities in the respective modules are made accessible. In the case of FOURIER, the names are those specified in the module. In the case of S_LIB, the entity named X_PRES is renamed PRESSURE in the program unit using the module. The other entities accessed from S_LIB have the same name in the using program unit as in the module.

3.6.4.2 Accessing only part of the public entities

To restrict the entities accessed from a module, specify ONLY on the USE statement, as follows:

```
USE module_name, ONLY: only_list
```

In this case, the using program unit has access only to those entities explicitly identified in the ONLY clause of the USE statement. All items in this list must identify public entities in the module. As with the unrestricted form of the USE statement, named accessed entities may be renamed for local purposes. The possible forms of each item in the *only_list* are as follows:

```
   [ local_name => ] module_entity_name
   OPERATOR(defined_operator)
   ASSIGNMENT(=)
```

The *local_name*, if present, specifies the name of the module entity in the using program unit.

```
USE MTD, ONLY:  X, Y, OPERATOR( .ROTATE. )
USE MONTHS, ONLY: JANUARY => JAN, MAY, JUNE => JUN
```

In the case of MTD, only X, Y, and the defined operator .ROTATE. are accessed from the module, with no renaming. In the case of MONTHS, only JAN, MAY, and JUN are accessed. JAN is renamed JANUARY and JUN is renamed JUNE.

In addition, specifying the statement on the left is equivalent to specifying the two statements on the right:

```
USE MTD, ONLY:  X, Y          USE MTD, ONLY:  X
                              USE MTD, ONLY:  Y
```

3.6.4.3 Entities accessible from a module

The following items can be defined, declared, or specified in a module, and may be public. They are accessed through the USE statement by other program units. Any public entity, except a defined operator or assignment interface, can be renamed in the using program unit.

- Declared variables
- Named constants
- Derived-type definitions
- Procedure interfaces
- Module and intrinsic procedures
- Generic identifiers
- Namelist groups

Note that the preceding list does not contain the implicit type rules of the module; these are not accessible through a USE statement.

A common block can be declared in a module. Because a common block name is global, however, it is not one of the name categories that is made available to another program unit via a USE statement. The names of the members of the common block in a module, though, are made available through the USE statement; these member names can be given local names through the USE statement renaming clauses. Consider, for example, the following module:

```
MODULE definer
COMMON /def/ i, j, r
END MODULE
```

All three of the members of common block DEF are made accessible to a program unit containing the following USE statement because all three variables are public entities of the module:

```
USE definer
```

The following USE statement limits access to only common block member R and gives it the local name MY_R:

```
USE def, ONLY: MY_R => R
```

The default accessibility for all entities in a module is PUBLIC unless this default has been changed by a PRIVATE statement with an empty entity list. If the default has been turned to PRIVATE, an entity can be made PUBLIC by its appearance in a PUBLIC statement or in a type declaration that contains the PUBLIC attribute. An entity can be specified to be PRIVATE in a PRIVATE statement or in a type declaration statement that contains the PRIVATE attribute.

Each named entity in a module is classified as either public or private. Regardless of this classification, all module entities can be used freely within the module, including within module procedures in the module; within a module procedure a module entity is governed only by the rules of host association. Outside the module, however, only the public entities are accessible (via the USE statement). Figure 11 illustrates these rules:

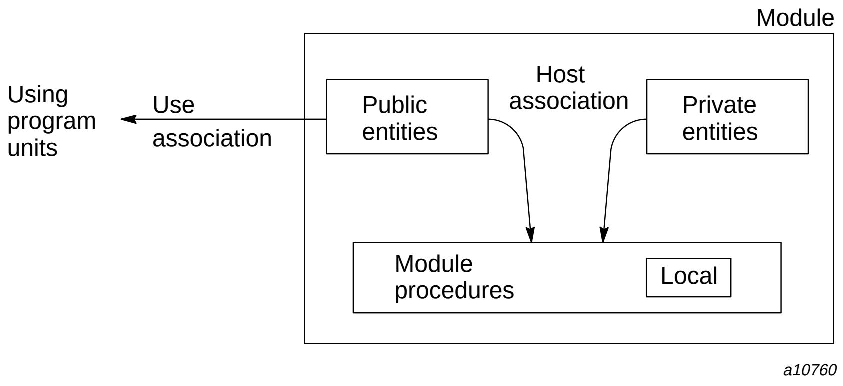


Figure 11. Use of public and private module entities

3.6.4.4 Name conflicts when using modules

When using modules, name conflicts can occur in the following two ways:

- A public entity in a module might have the same name as a local entity in the using program.
- Two modules being used might each have a public entity with the same name. Such a name conflict is allowed if and only if that name is never

referenced in the using program. If a name is to be referenced in the using program, potential conflicts involving that name must be prevented through use of the rename or ONLY facilities of the USE statement. This is the case even if the using program is another module.

Example:

```
MODULE BLUE
   INTEGER  A, B, C, D
END MODULE BLUE
MODULE GREEN
   USE BLUE, ONLY : AX => A
   REAL  B, C
END MODULE GREEN
! in program RED:
! integer A is accessed as AX or A
! integer B is accessed as B
!   real B is accessed as BX
! neither C is accessible, because
!   there is a name conflict
PROGRAM RED
   USE BLUE            ! accesses A, B, C, and D
   USE GREEN, BX => B  ! accesses A as AX, B as
                       ! BX, and C
   REAL D              ! Illegal; D cannot be
   . . .               ! redeclared locally.
END
```

3.6.4.5 Use association

The USE statement gives a program unit access to other entities not defined or specified locally within the using program. The association between a module entity and a local entity in the using program unit is termed *use association*. Host association is analogous, but host association applies only to a module and its module procedures and to internal procedures and their hosts. There are many similarities between use association and host association. Their rules, however, are different in the following two ways:

- The implicit typing rules of a module have no effect on the environment of a using program unit
- Entities accessed through a USE statement must not be respecified locally

The only exception to the second rule is that if the using program unit is another module, then the using module can specify an entity from the used module to be PRIVATE in the using module, rather than maintaining public accessibility. This is perhaps best illustrated with an example. In the following code, program units using module M2, defined as follows, can access X but not Y, even though Y is a public entity of M1:

```
MODULE M2
   USE M1, ONLY: X, Y
   PRIVATE  Y
      . . .
END MODULE M2
```

The prohibition on respecifying entities accessed via use association includes the use of module data objects in locally specified COMMON and EQUIVALENCE specifications.

While a name accessed from a module must not be respecified locally, the same name can be imported from another module under either of the following conditions:

- Both accesses are to the same entity. For example, if a program unit uses both M1 and M2 in the preceding example, both give access to the same X. This is allowed.
- The accesses are to different entities, but the using program unit makes no reference to that name.

3.6.5 Typical applications of modules

Some Fortran applications may be easier to write and understand when using modules. Modules provide a way of packaging the following types of data:

- Global data, previously packaged in common blocks
- User-defined operators
- Software libraries
- Data abstraction

These uses for modules are summarized in the following sections.

3.6.5.1 Global data

A module provides an easy way of making type definitions and data declarations global in a program. Data in a module does not have an implied storage association or an assumption of any form of sequence or any order of appearance, unless it is a sequence structure or in a common block. Global data in a module can be of any type or combination of types, as follows:

```
MODULE MODELS
   COMPLEX             :: GTX(100, 6)
   REAL                :: X(100)
   REAL, ALLOCATABLE :: Y(:), Z(:, :)
   INTEGER                CRX, GT, MR2
END MODULE
```

There are alternative ways to use the preceding module. The following statement makes all the data (and their attributes) of the module available:

```
USE MODELS
```

The following statement makes only the data named X and Y and their attributes available to the program using the module:

```
USE MODELS, ONLY: X, Y
```

The following statement makes the data object named Z available, but it is renamed to T for that particular application. In addition, it makes the other public entities of the module MODELS available with the same names they have in the module:

```
USE MODELS, T => Z
```

3.6.5.2 Common blocks in a module

One way of packaging common blocks is by putting them in a module. This makes migration of FORTRAN 77 programs that use common blocks easier. Consider the following program:

```
MODULE LATITUDE
   COMMON . . .
   COMMON . . .
   COMMON /BLOCK1/ . . .
END MODULE
   . . .
PROGRAM NAVIGATE
```

```
USE LATITUDE
   . . .
END
```

The USE statement in the preceding example makes all of the variables in the common blocks in the module available to the program NAVIGATE. These common blocks can be made available to other program units in the same way.

Unless there is a reason for variables to have their storage association in a particular order, there is no need to include a common block in a module. The data in a module is already global.

3.6.5.3 Global user-defined types

A derived type defined in a module is a user-defined type that can be made accessible to other program units. The same type definition can be referenced through a USE statement by more than one program unit. Consider the following code:

```
MODULE NEW_TYPE
   TYPE TAX_PAYER
      INTEGER SSN
      CHARACTER(20) NAME
   END TYPE TAX_PAYER
END MODULE NEW_TYPE
```

In the preceding example, the module NEW_TYPE contains the definition of a new type called TAX_PAYER. Procedures using the module NEW_TYPE may declare objects of type TAX_PAYER.

3.6.5.4 Operator extensions

An interface block can declare new operators or give additional meanings to the intrinsic ones, such as +, .EQ., .OR., and //. The assignment symbol = also can be given additional meanings and may be redefined for derived-type intrinsic assignment. (Derived-type assignment is the only instance of an intrinsic operation or assignment that can be redefined.) These extensions require that the OPERATOR or ASSIGNMENT option be on the interface block, the details of which appear in Section 4.8, page 221.

A simple example of an OPERATOR interface for matrix inversion requires a function and an interface block defining the new operator. In the following example, which normally (but not necessarily) would be in a module, the function INVERSE defines the desired operation, and the operator .INVERSE. can be used in an expression to reference the function:

```
INTERFACE OPERATOR(.INVERSE.)
   FUNCTION INVERSE(MATRIX_1)
      TYPE(MATRIX), INTENT(IN) :: MATRIX_1
      TYPE(MATRIX) :: INVERSE
   END FUNCTION INVERSE
END INTERFACE
```

An example of its use might be as follows (assuming + also has been extended to add a real value and a MATRIX):

```
1.0 + (.INVERSE. A)
```

3.6.5.5 Data abstraction

Data type definitions and operations can be packaged together in a module. Program units using this module will have the convenience of a new data type specific to a particular application. The following is a simple example:

```
MODULE POLAR_COORDINATES
   TYPE POLAR
      PRIVATE
      REAL RHO, THETA
   END TYPE POLAR
   INTERFACE OPERATOR(*)
      MODULE PROCEDURE POLAR_MULT
   END INTERFACE
CONTAINS
   FUNCTION POLAR_MULT(P1, P2)
      TYPE(POLAR) P1, P2, POLAR_MULT
      POLAR_MULT = &
            POLAR(P1%RHO   * P2%RHO,  &
                  P1%THETA + P2%THETA)
   END FUNCTION POLAR_MULT
      . . .
END MODULE POLAR_COORDINATES
```

In the function POLAR_MULT, the structure constructor POLAR computes a value that represents the result of multiplication of two arguments in polar coordinates. Any program unit using the module POLAR_COORDINATES has access to both the type POLAR and the extended intrinsic operator * for polar multiplication.

3.6.5.6 Procedure libraries

A module can contain a collection of interface blocks for related procedures. Argument keywords, as well as optional arguments, can be used to differentiate various applications using these procedures. The following is a simple example:

```
MODULE ENG_LIBRARY
   INTERFACE
      FUNCTION FOURIER(X, Y)
          . . .
      END
      SUBROUTINE INPUT(A, B, C, L)
         OPTIONAL C
         . . .
      END SUBROUTINE INPUT
   END INTERFACE
END MODULE ENG_LIBRARY
```

The following example shows that an input routine can be called using optional or keyword arguments:

```
CALL INPUT (AXX, L = LXX, B = BXX)
```

A collection of related procedures that need to access the same type definitions and data declarations can be placed in a module. The following example shows this:

```
MODULE BOOKKEEPING
   TYPE, PRIVATE :: ID_DATA
      INTEGER ID_NUMBER
      CHARACTER(20) NAME, ADDRESS(3)
      REAL BALANCE_OR_SALARY
   END TYPE ID_DATA
   REAL, PRIVATE :: GROSS_INCOME, EXPENSES,  &
                    PROFIT, LOSS
   INTEGER, PARAMETER :: NUM_CUST = 1000,  &
                         NUM_SUPP = 100,  &
                         NUM_EMP  = 10
CONTAINS
   SUBROUTINE ACCTS_RECEIVABLE(CUST_ID, AMOUNT)
      . . .
   END SUBROUTINE ACCTS_RECEIVABLE
   SUBROUTINE ACCTS_PAYABLE(CUST_ID, AMOUNT)
      . . .
   END SUBROUTINE ACCTS_PAYABLE
   SUBROUTINE PAYROLL(EMP_ID, AMOUNT)
      . . .
   END SUBROUTINE PAYROLL
   FUNCTION BOTTOM_LINE(AMOUNT)
      . . .
   END FUNCTION BOTTOM_LINE
END MODULE
```

3.6.6 Independent compilation

Independent compilation is the practice of compiling or processing subprograms separately and then using the compiled program units in a number of applications without the inconvenience or cost of recompiling those units. In FORTRAN 66 and FORTRAN 77, each program unit was entirely independent of other units.

The Fortran 90 `INCLUDE` facility behaves as if the source text from another file were inserted in place of the `INCLUDE` line prior to compilation. This departs from pure independent compilation in some respects because the program unit in which the `INCLUDE` line is contained now depends on material from elsewhere.

The use of a module is also a departure from pure independent compilation in that a program unit being compiled depends on information from a module.

If the program unit contains a reference to a module, the module must have been previously compiled and available to the using program unit; either through a module search path or appearing previously in the file being compiled. However, if no modules or `INCLUDE` lines are used, program unit compilation is completely independent of other sources of information.

Because a module is a complete program unit, it can be compiled independently. An advantage is that the module contents can be put into a precompiled form that can be incorporated efficiently during the program units' compilation.

Although there are frequently some dependencies using modules, it is often possible to put together a self-contained package consisting of certain modules and the program units that use them. This package is independent of other packages that might be part of the Fortran program; packages may be used in the same way as independent compilation has been used in the past. For example, such a module package may be compiled independently of the main program and external procedures, both of which may be compiled independently of the module package as long as these external procedures do not use the module package. In cases where program units use the module package, such program units are required to be compiled after the module package is compiled.

3.7 Block data program units

A block data program unit initializes data values in a named common block. The block data program unit contains data specifications and initial data values. Executable statements are not allowed in a block data program unit and the block data program unit is referenced only in EXTERNAL statements in other program units; its only purpose is to initialize data.

The block data program unit is defined as follows:

R1110	*block_data*	**is**	*block_data_stmt* [*specification_part*] *end_block_data_stmt*
R1111	*block_data_stmt*	**is**	BLOCK DATA [*block_data_name*]
R1112	*end_block_data_stmt*	**is**	END [BLOCK DATA [*block_data_name*]]

The preceding BNF definition results in the following general format to use for constructing a block data program unit:

```
BLOCK DATA [ block_data_name ]
   [ specification_part ]
   END [ BLOCK DATA [ block_data_name ] ]
```

The following is an example of a block data program unit:

```
BLOCK DATA SUMMER
   COMMON /BLOCK_2/ X, Y
   DATA X /1.0/,  Y /0.0/
END BLOCK DATA SUMMER
```

The name SUMMER appears on the BLOCK DATA statement and the END statement. X and Y are initialized in a DATA statement; both variables are in named common block BLOCK_2.

The CF90 and MIPSpro 7 Fortran 90 compilers allow you to include 26 unnamed block data program units.

ANSI/ISO: The Fortran 90 standard specifies that there can be at most one unnamed block data program unit in an executable program.

The block data name on the `END` statement, if present, must be the same as on the `BLOCK DATA` statement.

The *specification_part* can contain any of the following statements or attributes. Other statements are prohibited.

- `COMMON`
- `DATA`
- `DIMENSION`
- `EQUIVALENCE`
- `IMPLICIT`
- `INTRINSIC`
- `PARAMETER`
- `POINTER`
- `SAVE`
- `TARGET`
- `USE`
- Derived-type definition
- Type declaration

A `USE` statement in a block data program unit can give access to a limited set of objects, such as named constants, sequence derived types, and variables used only as arguments of inquiry functions. Most uses are disallowed by the restrictions on variables in block data programs.

The block data program unit can initialize more than one named common block.

It is not necessary to initialize an entire named common block.

The CF90 and MIPSpro 7 Fortran 90 compilers permit named common blocks to appear in more than one block data program unit.

ANSI/ISO: The Fortran 90 standard permits a given named common block to appear in only one block data program unit.

Using Procedures [4]

Procedures help to structure a problem solution into understandable segments. This chapter describes the details for constructing and using procedures in Fortran 90.

4.1 Procedure terms and concepts

This section describes some of the basic terms and concepts associated with Fortran procedures.

4.1.1 Procedure terms

A procedure can take two basic forms in a Fortran program: a subroutine and a function. These two forms are very similar except in the way they are invoked.

4.1.1.1 Subroutines

A *subroutine* is a procedure whose purpose is to perform an action, such as modifying a set of arguments and/or global variables, or performing input/output (I/O). Typically, a subroutine is invoked with a `CALL` statement, but Fortran 90 also provides an additional form of subroutine reference: the defined assignment. A subroutine can be used to define a new form of assignment, one that is different from those intrinsic to Fortran. Such subroutines are invoked with assignment syntax (using the = symbol) rather than with the `CALL` statement.

4.1.1.2 Functions

The purpose of a *function* is to provide a value needed in an expression. A function is invoked as an expression operand, and the result is used as the value of that operand. In addition, in Fortran 90, a function can be used to define a new operator or extend the meaning of an intrinsic operator symbol; such a function is invoked by the appearance of the new or extended operator in the expression along with the appropriate operand(s). For example, an interpretation for the operator + can be defined for logical operands. This extends the + operation's intrinsic definition because the intrinsic definition of + involves only numeric operands.

4.1.1.3 Function results

The main difference between a subroutine and a function is that there is a *function result* value associated with a function. More precisely, there is a result value associated with any particular execution or call to a function. This result can be of any type, including derived type, and it can be array-valued. The RESULT option in the FUNCTION statement can be used to give the result a different name than the function name. The RESULT clause is not required for a recursive function that calls itself directly unless the function result is array-valued.

ANSI/ISO: If a function invokes itself directly, the Fortran 90 standard requires that a RESULT clause be specified.

4.1.1.4 External procedures

External procedures are stand-alone subroutines and functions that are not part of any other program unit. They can share information, such as data and procedures through argument lists, modules, and common blocks, but otherwise they do not share information with any other program unit. They can be developed, compiled, and used completely independently of other procedures and program units. In fact they need not even be written in Fortran.

4.1.1.5 Intrinsic procedures

Fortran 90 contains intrinsic procedures such as sine and cosine. Intrinsic procedures are available to any Fortran program unit automatically. There are over 100 intrinsic procedures in Fortran 90, and the CF90 and MIPSpro 7 Fortran 90 compilers include several additional procedures as extensions. These procedures are described in Chapter 5, page 237.

Many of the intrinsic procedures are generic. The generic properties (when two or more procedures share the same name) of an intrinsic procedure can be extended and the automatic availability of an intrinsic procedure can be overridden explicitly by an EXTERNAL statement or a procedure interface block.

Many intrinsic procedures can be called *elementally*. In these cases an array is supplied instead of a scalar for an actual argument. The computation is applied element-by-element to those arguments and returns a conformable array result. User-defined procedures cannot be called elementally.

4.1.1.6 Internal procedures

Internal procedures are defined within other procedures. The procedure that contains an internal procedure is called the *host* of the internal procedure. An internal procedure can be either a subroutine or a function and appears between the CONTAINS and END statements of its host. An internal procedure is local to its host and inherits the host's environment through host association.

4.1.1.7 Module procedures

Module procedures are defined within module program units. A module procedure can be either a subroutine or a function and appears between the CONTAINS and END statements of its host module. A module procedure inherits the host module's environment through host association. A module procedure can be PRIVATE to the module, and hence available only within the module, or it can be PUBLIC.

4.1.1.8 Statement functions

Statement functions are one-statement function definitions in the specification part of a program unit other than a module or a block data program unit. Their functionality is extended and essentially superseded by internal functions in Fortran 90.

4.1.1.9 Procedure entry

Normally, one procedure is associated with a procedure subprogram. However, a procedure subprogram, which is a syntactic entity, can define any number of conceptual procedures. The name of the procedure subprogram identifies one procedure associated with that subprogram. An ENTRY statement can be used to specify and identify an additional procedure associated with that subprogram. These statements are *procedure entries*, and each defines an additional procedure. This technique is often used in external procedure subprograms to define different actions involving the data environment of the subprogram. The classic example is the use of the same subprogram to define both the SIN and COS functions, because COS(*x*)= SIN (π /2-*x*). This sort of data sharing is provided by host association for internal and module procedures; therefore, procedure entries are not needed for internal and module procedures. Procedure entries are not permitted for internal procedures, although they are permitted in module procedures.

4.1.1.10 Procedure reference

Procedure reference is the term given to the appearance of a procedure name in a program in such a way that causes the procedure to be executed. This is also termed *calling* or *invoking* the procedure. In most cases *reference* is used in this chapter, although occasionally *call* or *invoke* is used. These terms are used for both functions and subroutines. When a procedure is invoked, it suspends execution of the program making the call while the procedure is executed. When procedure execution is completed, the invoking program resumes execution.

A *subroutine reference* is a standalone action in the form of a CALL statement. In some cases, the call can take the form of an assignment statement.

A *function reference* occurs as part of an expression when the name of the function and its argument list appear as a primary in the expression. In some cases, a function reference can take the form of a unary or binary operation involving an operator with the arguments as its operands.

4.1.1.11 Actual arguments

Actual arguments appear in a procedure reference and specify the actual entities to be used by the procedure during its execution. These can be variables, for example, with different values for each reference. Some arguments are used as input values, others are variables that receive results from the procedure execution, and some can be both.

4.1.1.12 Dummy arguments

Dummy arguments are the names by which the actual arguments are known inside a procedure. You must specify these names when the procedure is defined and use them to represent arguments in computations in the procedure. When the procedure is referenced during program execution, the actual arguments in the reference become associated with the dummy arguments through argument association. If the procedure interface is explicit, a call to the procedure can use the dummy argument names as actual argument keywords.

4.1.1.13 Alternate return

Alternate returns are special arguments allowed only in subroutines. They permit control to branch immediately to some spot other than the statement following the call. The actual argument in an alternate return is the label of the statement to which control should be transferred. This is frequently used in accommodating error exits from the subroutine. Alternate returns are an obsolescent feature, so control structures, such as `IF` and `CASE` constructs, should be used to achieve the desired control.

4.1.1.14 Dummy procedures

Dummy argument names can be treated within the procedure definition as procedure names. That is, they can be used as procedure names in procedure references; this accommodates procedure passing. The associated actual argument for a dummy procedure must be the name of an actual procedure. The exceptions are that internal procedures, statement functions, and generic intrinsic names cannot be used as actual arguments.

4.1.1.15 Non-Fortran procedures

Procedure definitions can be written in a language other than Fortran. As long as all references to such a procedure are consistent in terms of the properties of the interface to this procedure, the calling program remains standard conforming. It may not be portable, however, because the non-Fortran procedure or its method of argument communication might differ across implementations. The only way to guarantee consistent interfaces across implementations is to write all procedures in standard Fortran.

4.1.2 Argument association

The term *argument association* refers to the matching up of data across procedure boundaries; that is, matching data sources being passed from the calling side with the appropriate receivers in the called procedure. One helpful image of this matching up is the plug/socket analogy in Section 4.7.1, page 201.

The term *argument association* refers first to the overall concept that such a matching up (or association) must take place in order to use procedures and second to the rules governing the matchups as described in Section 4.7, page 200.

4.1.3 Recursion

Fortran 90 procedures can be recursive. A procedure involved in either direct or indirect recursion must have the keyword `RECURSIVE` added to the `FUNCTION` or `SUBROUTINE` statement of the procedure definition.

Data initialization implies use of the SAVE attribute. For recursive procedures that have local data-initialized variables, each layer of recursion shares a single copy of the variable and initialization takes place only once, at the outset of program execution.

4.1.4 Host and use association

A procedure can access information specified outside its own scope of definition in the following ways:

- Argument association
- Common blocks
- Host association
- Use association

Argument association is fundamentally the same as in FORTRAN 77 and is extended naturally for the new features in Fortran 90. Argument association is described in Section 4.7, page 200. Common blocks, extended with data structures, are the same as in FORTRAN 77 and are described in the *Fortran Language Reference Manual, Volume 1*, publication SR–3902. Host and use association are new in Fortran 90 and are described in Section 3.4, page 145, and Section 3.6.4, page 156.

Host association applies to a procedure defined (contained) within another (host) program unit. A host can be a main program, module program unit, external procedure, or module procedure. Data and procedure entities specified in or accessible to the host are accessible to the contained procedure through host association. The rules for host association are very similar to the scoping rules of typical block-structured languages, such as Pascal. The main difference is that the Fortran 90 host association rules must take into account implicit as well as explicit declarations. See Section 3.4, page 145, for a complete description of these rules.

Use association applies to procedures and program units containing `USE` statements. All public entities of a module are available to a using procedure through use association, although the `USE...ONLY` mechanism can limit accessibility as the programmer desires. Use association allows shared data and procedure entities to be gathered together in a central place, with selective access to and hiding of these entities as appropriate in the specific situation. See Section 3.6.4.5, page 161, for a complete description of use association.

4.1.5 Implicit and explicit interfaces

The interface to a procedure is the collection of names and attributes of the procedure and its arguments. When this information is not made available explicitly to the calling program, the interface is said to be *implicit* to the calling program. In this case the interface information is assumed by the calling program from the properties of the procedure name and actual arguments in the procedure call. With implicit interfaces, the compiler assumes that the programmer has specified a valid procedure call and has correctly matched actual argument and dummy argument data types, and so on. For array arguments, element sequence association is assumed. For pointer arguments, the target is passed. External procedures and statement functions have implicit interfaces. Note that an implicit interface prohibits the use of generic references and argument keywords because there are no such mechanisms to describe these characteristics.

A procedure interface is said to be *explicit* if the interface information is known at the point of call and does not have to be assumed. In this case the compiler can check and guarantee the validity of the call. Intrinsic procedures have explicit interfaces. The explicit nature of the intrinsic procedure interfaces, for example, permits generic intrinsic functions and keyword calls. The compiler can, based on the type of the actual argument, generate a call to the correct specific intrinsic function, because the compiler has explicit interface information for all of the intrinsic functions.

The interface block allows optional explicit specification of external procedure interfaces and is described later in this chapter.

Several important new features in Fortran 90 require explicit interfaces in order to allow correct and efficient procedure calls. These include optional arguments, calls using argument keywords, user-defined operations, user-defined assignment, and user-defined generic procedures.

4.2 Subroutines

A subroutine defines a complete process and is self-contained. It has an initial `SUBROUTINE` statement, a specification part, an execution part that comprises the algorithm, any internal procedures that perform ancillary processes, and an `END` statement. When a subroutine is invoked, its execution begins with the first executable construct in the subroutine. Data objects and other entities may be communicated to and from the subroutine through argument association, host association, use association, or common storage association.

4.2.1 Subroutine definition

The general format of an external, module, or internal subroutine is as follows:

```
[ RECURSIVE ] SUBROUTINE subroutine_name [([ dummy_arg_list ])]
   [ specification_part ]
   [ execution_part ]
   [ internal_subprogram_part ]
   END [ SUBROUTINE [ subroutine_name ] ]
```

This format contains entities and statements that are defined as follows:

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

A *dummy_arg* is either a dummy argument name or an asterisk (*), where the asterisk designates an alternate return. When a subroutine is executed, the dummy arguments become associated with the actual arguments specified in the call (see Section 4.7, page 200, for more information on this).

The following examples show SUBROUTINE statements:

```
SUBROUTINE CAMP(SITE)
SUBROUTINE TASK()
SUBROUTINE INITIALIZE_DATABASE
SUBROUTINE LIGHT(INTENSITY, M, *)
RECURSIVE SUBROUTINE YKTE(Y, KE)
```

The following example shows a subroutine subprogram:

```
SUBROUTINE TROUT(STREAM, FLY)
   CHARACTER*10 STREAM
   INTENT(IN) FLY
   STREAM = . . .
      . . .
END SUBROUTINE TROUT
```

If the END statement contains a subroutine name, it must be the same name as that in the SUBROUTINE statement.

An internal subroutine must not contain an internal subprogram part. An internal subroutine must not contain ENTRY statements.

In an internal or module subroutine, the END statement must be of the following form because the keyword SUBROUTINE is required:

```
END SUBROUTINE [ subroutine_name ]
```

The use of alternate returns is an obsolescent feature.

The following rules apply to the subroutine *prefix*:

- The *prefix* can contain no more than one of each type of *prefix_spec*. For example, you cannot specify PURE twice in a *prefix*.
- If you specify ELEMENTAL or PURE, the subroutine itself must conform to the rules governing elemental or pure procedures. See Section 4.5, page 196, for information on pure procedures. See Section 4.6, page 199, for information on elemental procedures.
- When a subroutine calls itself either directly or indirectly, you must specify RECURSIVE.

ANSI/ISO: The Fortran 90 standard does not describe the PURE or ELEMENTAL keywords.

Dummy argument attributes can be specified explicitly in the body of the subroutine or may be declared implicitly. Each dummy argument is a local variable of the subroutine; therefore, its name must be different from that of any other local variable in the subroutine.

The `INTENT` attribute can be specified for the dummy arguments of the subroutine, but an `INTENT` attribute cannot be specified for a dummy pointer or a dummy procedure.

The `PRIVATE` and `PUBLIC` attributes must not be specified in a subroutine.

4.2.2 Subroutine reference

To use or invoke a subroutine, place a `CALL` statement or defined assignment at that point in a program where the process the subroutine performs is needed. A subroutine invocation specifies the arguments to be used and the name of the subroutine.

A subroutine reference is defined as follows:

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

Each *actual_arg* is associated with the corresponding dummy argument, as described in Section 4.7, page 200, by its position in the argument list or the name of its *keyword*. A *variable* is a special case of expression in the context of an *actual_arg*; a *variable* may be associated with dummy arguments used with any intent (`IN`, `OUT`, `INOUT`), whereas other forms of expressions must be associated either with dummy arguments with intent `IN` or dummy arguments that do not have an `INTENT` attribute.

Positional arguments must appear first in the argument list if both positional and *keyword* arguments are used in the same *actual_arg_spec_list*. Once the first *keyword* is used, the rest of the arguments must be keyword arguments.

Exactly one *actual_arg* is associated with each nonoptional dummy argument. For an optional dummy argument, the *actual_arg* may be omitted.

The *keyword* is the name of the dummy argument in the explicit interface for the subroutine. If a *keyword* is present, the *actual_arg* is associated with the dummy argument with that keyword name.

If the *keyword* is omitted, it must be omitted from all preceding *actual_arg* s in that argument list. If no *keyword* is used, the arguments all have a positional correspondence.

The *label* in the alternate return specifier must be a branch target in the same scoping unit as the CALL statement.

An *actual_arg* must not be the name of an internal procedure or statement function.

An *actual_arg* associated with a dummy procedure must be the specific name of a procedure. (There may be an identical generic name, but it is the procedure with that specific name that is passed.) Certain specific intrinsic function names must not be used as *actual_arg*s; see the *Intrinsic Procedures Reference Manual*, publication SR–2138, for more information on this.

Example of subroutine references:

```
      CALL TYR(2.0*A, *99) ! SUBROUTINE TYR (R, *)
      . . .
99    . . .                ! error recovery
      CALL TEST(X = 1.1, Y = 4.4) ! SUBROUTINE
                           !   TEST (Y, X)
```

In the first example, an alternate return to statement 99 in the calling program unit is the last argument. Keyword arguments are used for X and Y in the second CALL statement; therefore, the order of the actual arguments does not matter.

Another way to invoke or reference a subroutine is with user-defined assignment. A subroutine may define forms of assignment different from intrinsic assignment supplied by Fortran 90. Defined assignment is particularly useful with data structures. Defined assignment subroutines require an ASSIGNMENT interface as described in Section 4.7.4, page 213. They have exactly two arguments, arg_1 and arg_2, both required and the first with intent OUT or INOUT and the second with intent IN. Defined assignment is invoked with the following assignment syntax:

arg_1 = arg_2

The attributes of the arguments select the defined assignment. This facility, in effect, allows you to extend the generic properties of assignment.

Example of defined assignment:

```
MODULE POLAR_COORDINATES
    TYPE POLAR
       REAL  ::  RHO, THETA
    END TYPE POLAR
    INTERFACE ASSIGNMENT (=)
       MODULE PROCEDURE ASSIGN_POLAR_TO_COMPLEX
    END INTERFACE
        . . .
    CONTAINS
    SUBROUTINE ASSIGN_POLAR_TO_COMPLEX(C, P)
       COMPLEX, INTENT(OUT)       ::  C
       TYPE(POLAR), INTENT(IN)   ::  P
       C = CMPLX(P%RHO * COS(P%THETA),  &
                 P%RHO * SIN(P%THETA))
    END SUBROUTINE ASSIGN_POLAR_TO_COMPLEX
END MODULE POLAR_COORDINATES

PROGRAM CART
USE POLAR_COORDINATES
COMPLEX :: CARTESIAN
   . . .
CARTESIAN = POLAR(R, PI/6)
   . . .
END PROGRAM
```

This last assignment is equivalent to the following subroutine call:

```
CALL ASSIGN_POLAR_TO_COMPLEX(CARTESIAN, POLAR(R, PI/6))
```

The structure constructor POLAR constructs a value of type POLAR from R and PI/6 and assigns this value to CARTESIAN according to the computations specified in the subroutine.

4.3 Functions

A function is similar to a subroutine, except that its principal use is as a primary in an expression. Analogous to a subroutine, a function has an initial FUNCTION statement, a specification part, an execution part, possibly internal procedures, and an END statement. An argument list provides data communication with the function, but in this case arguments typically serve as input data for the function. The principal output is delivered as the function result to the expression invoking the function. Data objects also may be available to the function through host association, use association, and common storage association.

4.3.1 Function definition

The general format of an external, module, or internal function subprogram is as follows:

```
[ function_prefix ] function_statement [ RESULT (result_name) ]
   [ specification_part ]
   [ execution_part ]
   [ internal_subprogram_part ]
   END [ FUNCTION [ function_name ] ]
```

A function is defined as follows:

R1215	*function_subprogram*	**is**	*function_stmt* [*specification_part*] [*execution_part*] [*internal_subprogram_part*] *end_function_stmt*
R1216	*function_stmt*	**is**	[*prefix*] FUNCTION *function_name* ([*dummy_arg_name_list*]) [RESULT(*result_name*)]
R1217	*prefix*	**is**	*prefix_spec* [*prefix_spec*] ...
EXT	*prefix_spec*	**is**	*type_spec*
		or	RECURSIVE
		or	PURE
		or	ELEMENTAL
R1218	*end_function_stmt*	**is**	END [FUNCTION [*function_name*]]

The simplest function statement can take the following form:

```
FUNCTION function_name ([ dummy_arg_name_list ])
```

When a function is executed, the dummy arguments become associated with the actual arguments specified in the reference. See Section 4.7, page 200, for more information on argument association.

The following are some example `FUNCTION` statements:

```
FUNCTION HOSPITAL(PILLS)
REAL FUNCTION LASER(BEAM)
RECURSIVE CHARACTER*10 FUNCTION POLICE(STATION) &
       RESULT(ARREST)
```

The following rules apply to the function *prefix*:

- The *prefix* can contain no more than one of each type of *prefix_spec*. For example, you cannot specify `PURE` twice in a *prefix*.
- If you specify `ELEMENTAL`, you cannot also specify `PURE`. These specifications are mutually exclusive. If you specify `ELEMENTAL` or `PURE`, the function itself must conform to the rules governing elemental or pure procedures. See Section 4.5, page 196, for information on pure procedures. See Section 4.6, page 199, for information on elemental procedures.
- When a function is either directly or indirectly recursive, you must specify `RECURSIVE`.

ANSI/ISO: The Fortran 90 standard does not describe the `PURE` or `ELEMENTAL` keywords.

The type of the function can be specified in the `FUNCTION` statement or in a type declaration statement, but not both. If the type is not explicitly specified in this way, the default typing rules apply.

If the function result is array valued or a pointer, the declarations must state these attributes for the function result name. The function result name can be declared to be an explicit-shape or deferred-shape array. If the function is an explicit-shape array, the bounds can be nonconstant specification expressions.

Dummy argument attributes can be specified explicitly in the body of the function or may be declared implicitly. Each dummy argument is a local variable of the function; therefore, its name must be different from that of any other local variable in the function. For more information on this, see Section 6.1, page 264.

If the `END` statement contains the function name, it must be the same name used in the `FUNCTION` statement.

An internal function must not contain an internal subprogram part.

An internal function must not contain ENTRY statements.

In an internal or module function, the END statement must be of the following form because the keyword FUNCTION is required, as follows:

```
END FUNCTION [ function_name ]
```

If there is no RESULT clause, the function name is used as the result variable. If the function result is scalar and of type integer, real, complex, logical, or character, the function name followed by a left parenthesis is considered to be a recursive reference to the function. Otherwise, it is a reference to the result variable.

ANSI/ISO: The Fortran 90 standard does not specify direct recursion unless a RESULT clause is specified on the FUNCTION statement.

If there is a RESULT clause, the *result_name* is used as the result variable, and the function name must not be used as the result variable; in this case, all references to the function name are function references (that is, recursive calls).

The function name must not appear in specification statements if there is a RESULT clause.

If the result of a function is not a pointer, its value must be completely defined before the end of execution of the function. If the result is an array, all the elements must be defined; if the result is a structure, all of the components must be defined.

If the result of the function is an array or a pointer to an array, its shape must be determined before the end of execution of the function.

If the result is a pointer, its allocation status must be determined before the end of execution of the function; that is, a target must be associated with the pointer, or the pointer must have been explicitly disassociated from a target.

The INTENT attribute can be specified for the dummy arguments of the function, except that an INTENT attribute must not be specified for a dummy pointer or a dummy procedure.

The PRIVATE and PUBLIC attributes must not be specified in a function.

4.3.2 The RESULT option

As with subroutines, when a function is either directly or indirectly recursive, RECURSIVE must appear in the FUNCTION statement. The RESULT clause specifies a name different from the function name to hold the function result. The result name can be declared, defined, and referenced as an ordinary data object. The function name has the same attributes as the result name. Upon return from a function with a RESULT clause, the value of the function is the last value given to the result name.

If there is no RESULT clause, the function name is used as the result data object. If the function is both array-valued and directly recursive, however, a recursive reference to the function is indistinguishable from a reference to the array-valued result. The RESULT clause resolves this ambiguity by providing one name for the result value (the result name) and another name for recursive calls (the function name).

A simple example of a recursive function is REVERSE, which reverses the words in a given phrase:

```
RECURSIVE FUNCTION REVERSE(PHRASE) RESULT(FLIPPED)
   CHARACTER(*)           PHRASE
   CHARACTER(LEN(PHRASE)) FLIPPED
   L = LEN_TRIM(PHRASE)
   N = INDEX(PHRASE(1:L), " ", BACK = .TRUE.)
   IF (N == 0) THEN
      FLIPPED = PHRASE
   ELSE
      FLIPPED = PHRASE(N+1:L) // " "  &
            // REVERSE(PHRASE(1:N-1))
   END IF
END FUNCTION REVERSE
```

4.3.3 Function reference

One way a function can be referenced or invoked is by placing the function name with its actual arguments as an operand in an expression. The actual arguments are evaluated; argument association takes place in accordance with the rules in Section 4.7, page 200; and the statements in the body of the function are executed. The reference results in a value that is then used as the value of that primary in the expression.

In the following example expression, F is a function of one argument that delivers a numeric result. This result becomes the value of the right-hand operand of the expression:

```
A + F(B)
```

A function reference is defined as follows:

R1209	*function_reference*	**is**	*function_name* ([*actual_arg_spec_list*])
R1211	*actual_arg_spec*	**is**	[*keyword* =] *actual_arg*
R1212	*keyword*	**is**	*dummy_arg_name*
R1213	*actual_arg*	**is**	*expr*
		or	*variable*
		or	*procedure_name*

The only difference between subroutine and function argument lists is that a function argument list must not contain an alternate return. Otherwise, the rules and restrictions for actual and dummy arguments are the same for functions and subroutines as those described in Section 4.2.2, page 180.

A *keyword* is a dummy argument name in the function interface. As with subroutines, each actual argument is associated with the corresponding dummy argument by position or keyword. A variable is a special case of expression in the context of an actual argument; variables can be associated with dummy arguments used with any intent (IN, OUT, INOUT), whereas other forms of expressions must be associated either with dummy arguments with intent IN or dummy arguments that do not have an INTENT attribute. Note that (A), where A is a variable, is not a simple reference to the variable but is a more general expression.

The following are examples of function references:

```
Y = 2.3 * CAPS(4*12.0, K)! FUNCTION CAPS(SIZE, KK)
PRINT *, TIME(TODAYS_DATE)! FUNCTION TIME(DATE)
```

Another way to reference a function is with user-defined operators in an expression. A number of arithmetic, logical, relational, and character operators are predefined in Fortran; these are called *intrinsic operators*. These operators can be given additional meanings, and new operators can be defined. Functions define these operations and interface blocks associate them with the desired operator symbols, as described in Section 4.7.3, page 210. A function can be invoked by using its associated defined operator in an expression. The rules associated with operator functions are as follows:

- Functions of one argument are used to define unary operations; functions of two arguments are used to define binary operations.
- The arguments are required and must have intent IN.
- New operators must have the dot form, must contain only letters (underscores not allowed) between the dots, must have no more than 31 letters, and must not be the same as logical literal constants such as .TRUE. or .FALSE.. Examples are .FOURIER., .NEWPLUS., and .BLAHANDBLAAH..

If a defined operator is the same as an intrinsic operator (for example, +, *, .EQ., .AND.), it extends the generic properties of this operator, as described. In such an extension, the attributes of the arguments must not match exactly those of the operands associated with an intrinsic meaning of the operator. For more information on intrinsic operations, see the *Fortran Language Reference Manual, Volume 1,* publication SR–3902.

In the following example, the presence of .BETA. in the expression in the PRINT statement invokes the function BETA_OP, with X as the first actual argument and Y as the second actual argument. The function value is returned as the value of X .BETA. Y in the expression, as follows:

```
INTERFACE  OPERATOR (.BETA.)
    FUNCTION BETA_OP(A, B)
        . . . ! attributes of BETA_OP, A, and B
              ! (including INTENT(IN) for A and B)
    END FUNCTION
END INTERFACE
   . . .
PRINT *, X .BETA. Y
```

4.3.4 Statement functions

A statement function statement is a function definition that consists of only one Fortran statement. It is defined as follows:

R1226	*stmt_function_stmt*	**is**	*function_name* ([*dummy_arg_name_list*]) = *scalar_expr*

A statement function statement can be replaced (except within an internal procedure) with the following equivalent three-line internal function definition:

```
FUNCTION function_name(dummy_arg_name_list)
   function_name = scalar_expression
   END FUNCTION
```

The preceding format assumes that the function and its arguments are typed the same in both cases. Additional rules governing statement functions follow these three examples of statement functions:

Example 1:

```
CHARACTER(5)  ZIP_5     ! Notice these are scalar
CHARACTER(10) ZIP_CODE  ! character strings
ZIP_5(ZIP_CODE) = ZIP_CODE(1:5)
```

Example 2:

```
INTEGER TO_POST, MOVE
TO_POST(MOVE) = MOD(MOVE,10)
```

Example 3:

```
REAL FAST_ABS
COMPLEX Z
FAST_ABS(Z) = ABS(REAL(Z)) + ABS(AIMAG(Z))
```

Note that the function and all the dummy arguments are scalar.

The expression must contain only intrinsic operations and must be scalar valued. Note that this allows the expression to include references to scalar-valued functions having array arguments, such as `SUM(A+B)`, where `SUM` is the array reduction intrinsic function and `A` and `B` are conformable arrays.

Statement functions are defined in the specification part of a program unit, internal procedure, or module procedure. Any other statement function referenced in the expression must have been defined earlier in the specification part, and hence a statement function cannot be recursive (either directly or indirectly).

Named constants and variables used in the expression must have been declared earlier in the specification part or made available by use or host association.

If an array element is used in the expression, the parent array must have been declared earlier in the specification part.

The appearance of any entity in the expression that has not previously been typed explicitly constitutes an implicit type declaration and any subsequent explicit type declaration for that entity must be consistent with the implicit type.

Statement function dummy arguments have a scope of the statement function statement.

Statement function dummy arguments are assumed to have the `INTENT (IN)` attribute (that is, function references in the expression must not change the value of any dummy argument of the statement function).

A statement function must not be used as an actual argument.

A statement function is referenced in the same manner as any other function, except that the statement function interface is implicit and therefore the keyword form of actual arguments is not allowed; the argument association rules are the same.

Note that statement function interfaces are implicit, not explicit; see Section 4.7.1, page 201, for a discussion of explicit interfaces. Explicit interfaces are associated with those procedures that can have array-valued results, assumed-shape dummy arguments, pointer arguments, and keyword arguments, and can have various generic forms. Because none of these apply to statement functions, statement functions do not have and are not allowed to have explicit interfaces.

4.4 Procedure-related statements

Several procedure-related statements (for example, `RETURN`, `CONTAINS`, `ENTRY`, `EXTERNAL`, and `INTRINSIC`) are general in that they apply to both kinds of procedures (functions and subroutines) or to more than one form of procedure (for example, external, internal, module). The following sections describe these statements.

4.4.1 `RETURN` statement

A `RETURN` statement terminates execution of a procedure and returns control to the calling program. Often, however, it is not needed because the procedure `END` statement performs the same function as well as constituting the physical end of the procedure. It is occasionally convenient to use `RETURN` statements, however, because they may be placed anywhere in the execution part of the procedure.

The RETURN statement is defined as follows:

R1224	*return_stmt*	**is**	RETURN [*scalar_int_expr*]

The *scalar_int_expr* argument is applicable only to subroutines and is used in conjunction with alternate returns. This expression must be of type integer, and its value must be in the range 1 to *n*, where *n* is the number of alternate returns in the argument list; the value selects which alternate return, counting from left to right in the argument list, is to be used for this particular return from the procedure. The effect of an alternate return is the same as that produced by the following statements (where inside SUBR, the variable IRET is assigned the integer expression alternate return value prior to returning from SUBR):

```
CALL SUBR (..., IRET)
GO TO (label_list), IRET
```

The alternate return is an obsolescent feature.

4.4.2 CONTAINS statement

The CONTAINS statement separates the internal procedures from the specification and executable parts of the host, and it separates module procedures from the specification part of the module. It is defined as follows:

R1225	*contains_stmt*	**is**	CONTAINS

It is a nonexecutable statement that has the effect of making the execution sequence bypass everything following the CONTAINS statement up to the END statement of the program unit. Therefore, if it were executable, the CONTAINS statement would have the effect of a STOP statement in a main program and a RETURN statement in a procedure subprogram.

The CONTAINS statement serves only to delimit the procedure part of a program unit.

4.4.3 ENTRY statement

Section 4.1.1, page 171, describes procedure entry. A procedure entry is defined by the appearance of an ENTRY statement in the specification or execution part of the procedure subprogram.

The ENTRY statement is defined as follows:

R1223	*entry_stmt*	**is**	ENTRY *entry_name* [([*dummy_arg_list*])] [RESULT (*result_name*)]

The ENTRY statement can be thought of as providing auxiliary FUNCTION statements in function subprograms or SUBROUTINE statements in subroutine subprograms, each defining another procedure. The entry names must be different from one another and from the original function or subroutine name.

The following example illustrates a typical way of using the ENTRY statement to define several procedures in a single subprogram. In this example, the set of executable statements follows each ENTRY statement and precedes the next one. The last one is a RETURN statement, which represents the procedure corresponding to the entry. When the procedure represented by the entry is called, the procedure is entered and execution proceeds from this point. Execution continues in the procedure in the normal manner, ignoring any ENTRY statements subsequently encountered, until a RETURN statement is executed or the end of the procedure is reached.

```
SUBROUTINE name_1 (argument_list_1)
      . . .
   RETURN

   ENTRY name_2 (argument_list_2)
      . . .! This falls through past the next ENTRY statement

   ENTRY name_3 (argument_list_3)
      . . .
   RETURN
END
```

Often the computations in these entry bodies are similar, involving the same data and code.

All of the entries in a subroutine subprogram define subroutine procedures, and all of the entries in a function subprogram define function procedures. All of the entries in a function subprogram must be compatible with storage association. The RESULT option on an ENTRY statement has the same form and meaning as the RESULT option on a FUNCTION statement.

The following are examples of the ENTRY statement:

```
ENTRY FAST(CAR, TIRES)
ENTRY LYING(X, Y) RESULT(DOWN)
```

The following list enumerates rules regarding the ENTRY statement:

1. If an ENTRY statement appears in a function subprogram, the parentheses in the ENTRY statement surrounding the optional dummy argument list must be present.
2. An ENTRY statement can appear only in an external or module subprogram; an internal subprogram must not contain ENTRY statements.
3. An ENTRY statement must not appear in an executable construct (IF, DO, CASE, or WHERE constructs) or a nonblock DO loop.
4. An entry name must not be the same as any dummy argument name in the subprogram.
5. An entry name must not appear in an EXTERNAL statement, INTRINSIC statement, or procedure interface block in that subprogram.
6. The RESULT option applies only to function entries and thus may appear only in function subprograms.
7. If *result_name* is specified, it must not be the same as any entry name, the function name, or any other *result_name*. If *result_name* is specified, the *entry_name* must not appear in any specification statements in the subprogram; it inherits all of its attributes from *result_name*.
8. The keywords PURE, ELEMENTAL, and RECURSIVE cannot be used in an ENTRY statement. Instead the presence or absence of these keywords on the initial SUBROUTINE or FUNCTION statement of the subprogram applies to each entry in the procedure.
9. If each entry result in a function subprogram has the same type, kind, and shape as the function result, each of the entries identifies (is an alias for) the same result variable. In this case there is no restriction on the nature of the result. For example, the result could be of derived type, either scalar or array, and could have the POINTER attribute.

If all of the entries in a function subprogram (including the function result) are not the same type, kind, and shape, then they must all be scalar, without the POINTER attribute, and must be capable of being equivalenced. This means they all must be of type character with the same length or any mix of default integer, default real, default logical, double-precision real, or default complex. The reason for these rules is that all subprogram entries are storage associated with the function result.

10. A dummy argument must not appear in an executable statement before the ENTRY statement specifying that dummy argument. A dummy argument of the ENTRY statement must not appear in a statement function scalar expression before the ENTRY statement specifying that dummy argument, unless it is also a dummy argument of the statement function.

11. An executable statement or statement function depending on a dummy argument of the procedure that was entered, or upon a local data object depending on that dummy argument (such as a dynamic local array whose size depends on the dummy argument), may be executed only if the dummy argument appears in the ENTRY statement of the referenced procedure. In addition, an associated actual argument must be present if the dummy argument is optional.

For either a function or subroutine subprogram the order, number, types, kind type parameters, and names of the dummy arguments in an ENTRY statement may differ from those in the FUNCTION or SUBROUTINE statement or any other ENTRY statement in that subprogram. Note, however, that all of the entry result values of a function subprogram must be able to be equivalenced to the function result value, as described previously in item 9.

The interface to a procedure defined by an ENTRY statement in an external subprogram can be made explicit in another scoping unit (the calling scoping unit) by supplying an interface body for it in a procedure interface block. In this case the ENTRY statement appears as the first statement of the interface body, but the word ENTRY is replaced by the word FUNCTION or SUBROUTINE, whichever is appropriate. Such an interface body must include RECURSIVE if the subprogram is recursive and must correctly specify the dummy argument attributes and the attributes of the result if it is a function. Entry procedures defined in module procedures already have explicit interfaces in program units that use the module.

4.4.4 EXTERNAL statement

Consider the following program segment in a program unit that contains no declarations:

```
   . . .
A = X + Y
CALL B(X, Y)
CALL Q(A, B, C)
   . . .
```

It is clear that A is a variable, and B and Q are subroutines. But in this code fragment, C could be either a variable name or a procedure name. The other statements in the program may or may not resolve the mystery. In the cases where they do not, the argument is assumed to be a variable. But when the programmer wants it to be a procedure name, there must be some way to specify it. The means for doing this is the EXTERNAL statement, which is defined as follows:

R1207	*external_stmt*	**is**	EXTERNAL *external_name_list*

The EXTERNAL statement appears in the program unit in which the procedure in question is an actual argument; the procedure must be an external procedure or dummy procedure. Internal procedures, statement functions, and generic names must not appear as actual arguments. Use association takes care of module procedures, and intrinsic procedures are handled separately. An interface block for the external procedure has the same effect (as well as providing argument checking and other benefits) and, therefore, effectively replaces the EXTERNAL statement for this purpose.

An external procedure name must not appear in both the EXTERNAL statement and in an interface body heading statement. A name that appears in an EXTERNAL statement must not also appear as a specific procedure name in an interface block in the same scoping unit. Another minor use of the EXTERNAL statement is to identify the relevant block data program unit.

4.4.5 INTRINSIC statement

The INTRINSIC statement does for intrinsic procedures what the EXTERNAL statement does for external procedures (see the preceding section). The INTRINSIC statement is defined as follows:

R1208	*intrinsic_stmt*	**is**	INTRINSIC *intrinsic_procedure_name_list*

Note that an interface block cannot be provided for an intrinsic procedure because that would specify a duplicate explicit interface. If the generic intrinsic procedure is being extended with user-supplied subprograms, however, an interface block can have a generic name that is the same as a generic intrinsic procedure name.

For example, in the following procedure reference, if the intrinsic function SIN is intended for the third actual argument, SIN must be declared in an INTRINSIC statement if it is not otherwise known to be a procedure name in that scope. (SIN is both a specific and a generic procedure name. It is the specific name that is involved here.)

```
CALL Q(A, B, SIN)
```

4.5 Pure procedures

A *pure* procedure is one that is free from side effects. Such a procedure does not modify data that is visible outside of the procedure. The procedure does not alter global variables or variables accessible by host or use association. It does not perform I/O, and it does not save any variables. It is safe to reference a pure procedure in parallel processing situations and when there is no explicit order of evaluation.

The NOSIDEEFFECTS directive has a functionality that approaches the functionality of the PURE keyword. You can compare the information in this section to the information on the NOSIDEEFFECTS directive in the *CF90 Commands and Directives Reference Manual*, publication SR–3901, and the *MIPSpro 7 Fortran 90 Commands and Directives Reference Manual*, publication SR–3907.

ANSI/ISO: The Fortran 90 standard does not describe PURE procedures.

A pure procedure can be any of the following types of procedures:

- A pure intrinsic function or subroutine
- An object defined by a pure subprogram
- A pure user-defined function or subroutine

A user-defined pure subprogram must have the PURE keyword on the SUBROUTINE or FUNCTION statement.

Several rules govern pure subprograms. Basically, a pure subprogram cannot contain any operation that could *conceivably* result in an assignment or pointer assignment to a common variable or a variable accessed by use or host association. Nor can a pure subprogram contain any operation that could conceivably perform any external file I/O or a STOP operation.

The word *conceivably* is used in the preceding paragraph to indicate that it is not sufficient for a pure subprogram merely to be side-effect free in practice. For example, a function that contains an assignment to a global variable in a block that is not executed in any invocation of the function is, nevertheless, not a pure function. Functions like this are excluded to facilitate compile-time checks.

4.5.1 Rules for pure procedures

For a function to be pure, most dummy arguments must be specified with the `INTENT(IN)`attribute in the *specification_part*. The exception to this is that procedure dummy arguments and dummy arguments with the `POINTER` attribute cannot be declared as such.

For a subroutine to be pure, you must declare the intents of most dummy arguments in the *specification_part*. Dummy arguments for which you do not need to specify the intent are procedure arguments, alternate return indicators, and arguments with the `POINTER` attribute. The constraints for pure subroutines are similar to the constrains on pure functions, but side effects to `INTENT(OUT)`, `INTENT(INOUT)`, and pointer dummy arguments are permitted.

The following additional rules apply to subroutines or functions that you declare to be pure:

- A local variable declared in the *specification_part* or *internal_subprogram_part* of a pure subprogram cannot have the SAVE attribute. Note that when a variable is initialized in a *type_declaration_stmt* or a DATA statement, the SAVE attribute is implied, so initialization in this manner is also disallowed.
- You must declare all dummy arguments that are procedure dummy arguments to be PURE in the *specification_part* of a pure subprogram.
- If a procedure that is neither an intrinsic procedure nor a statement function is used in a context that requires it to be pure, then its interface must be explicit in the scope of that use. The interface must specify that the procedure is PURE.
- All internal subprograms in a pure subprogram must be pure.
- Any procedure referenced in a pure subprogram, including one referenced by a defined operation or assignment, must be pure.

- Pure subprograms cannot contain a `PRINT`, `OPEN`, `CLOSE`, `BACKSPACE`, `ENDFILE`, `REWIND`, or `INQUIRE` statement.
- Pure subprograms cannot contain `READ` or `WRITE` (including `PRINT`) statements that specify an external file unit nor can they contain `*` as an I/O unit.
- Pure subprograms cannot contain a `STOP` or a `PAUSE` statement.

4.5.2 Using variables in pure procedures

Certain rules apply to variables used in pure subprograms. These rules apply to the following types of variables:

- Variables in common blocks
- Variables accessed by host or use association
- Variables that are dummy arguments to a pure function
- Variables that are dummy arguments with `INTENT(IN)` to a pure subroutine

The following rules apply to both the types of variables described in the preceding list and to objects that are storage associated with any such variable. Specifically, these variables and objects cannot be used in the following contexts:

- As the *variable* of an assignment statement
- As a `DO` variable or implied-`DO` variable
- As an *input_item* in a `READ` statement from an internal file
- As an *internal_file_unit* in a `WRITE` statement
- As an `IOSTAT=` specifier in an input or output statement with an internal file
- As the *pointer_object* of a pointer assignment statement
- As the target of a pointer assignment statement
- As the *expr* of an assignment statement in which the *variable* is of a derived type if the derived type has a pointer component at any level of component selection
- As an *allocate_object* or *stat_variable* in an `ALLOCATE` or `DEALLOCATE` statement
- As a *pointer_object* in a `NULLIFY` statement

- As an actual argument associated with a dummy argument with `INTENT(OUT)` or `INTENT(INOUT)` or with the `POINTER` attribute.

4.6 Elemental procedures

An *elemental* procedure is an elemental intrinsic procedure or a procedure that is defined by a user-specified elemental subprogram. A user-defined elemental subprogram must have the `ELEMENTAL` keyword on the `SUBROUTINE` or `FUNCTION` statement.

Note: An elemental subprogram is a pure subprogram, and all of the constraints for pure subprograms also apply to elemental subprograms. See Section 4.5, page 196, for more information on pure subprograms.

An elemental subprogram is, by definition, also a pure subprogram, but the `PURE` keyword need not be present. The following additional rules apply to elemental subroutines and functions:

- All dummy arguments must be scalar and cannot have the `POINTER` attribute.
- A function must return a scalar result, and the result cannot have the `POINTER` attribute.
- A dummy argument, or a subobject of a dummy argument, cannot appear in a *specification_expr* except as an argument to the `BIT_SIZE`(3I), `KIND`(3I), or `LEN`(3I) intrinsic functions, or to one of the numeric inquiry functions.
- A dummy argument cannot be an asterisk (`*`).
- A dummy argument cannot be a dummy procedure.

Example:

```
ELEMENTAL REAL FUNCTION F(A)
  REAL, INTENT(IN) :: A
  REAL(SELECTED_REAL_KIND(PRECISION(A)*2)) :: WORK
  ...
END FUNCTION F
```

4.6.1 Elemental functions

If a generic name or a specific name is used to reference an elemental function, the shape of the result is the same as the shape of the actual argument with the greatest rank. If the actual arguments are all scalar, the result is scalar. For those elemental functions that have more than one argument, all actual arguments must be conformable. In the array-valued case, the values of the elements, if any, of the result are the same as would have been obtained if the scalar-valued function had been applied separately, in any order, to corresponding elements of each array actual argument.

The following is an example of an elemental reference to the MAX(3I) intrinsic function. Assume that X and Y are arrays of shape (M,N). The reference is as follows:

```
MAX(X, 0.0, Y)
```

The preceding statement is an array expression of shape (M,N) whose elements have the following values:

```
MAX (X(I, J),  0.0, Y(I, J))
```

In the preceding example, I has the values 1 to M, and J has the values 1 to N.

4.6.2 Elemental subroutines

An elemental subroutine is one that has only scalar dummy arguments, but it may have array actual arguments. In a reference to an elemental subroutine, one of the following situations must exist:

- All actual arguments must be scalar.
- All actual arguments associated with INTENT(OUT) and INTENT(INOUT) dummy arguments must be arrays of the same shape and the remaining actual arguments must be conformable with them. In this case, the values of the elements, if any, of the results are the same as would be obtained if the subroutine had been applied separately, in any order, to corresponding elements of each array actual argument.

In a reference to the intrinsic subroutine MVBITS(3I), the actual arguments corresponding to the TO and FROM dummy arguments may be the same variable. Apart from this, the actual arguments in a reference to an elemental subroutine must satisfy the restrictions of Section 4.7.2.3, page 208.

4.7 Argument association

When a procedure is referenced, the actual arguments supply the input data to be used for this execution of the procedure and specify the variables to receive any output data. Within the procedure, the dummy arguments assume the roles of these input and output data objects. Thus, during execution of a procedure reference, the appropriate linkage must be established between the actual arguments specified in the call and the dummy arguments defined within the procedure. This linkage is called *argument association*.

As shown in Figure 12, the fundamental form that a set of actual arguments takes in a reference is that of a sequence of expressions separated by commas. The set of names in a procedure definition after the procedure name is a list of dummy argument names. In each case this is called an argument list (an *actual argument list* in the former case and a *dummy argument list* in the latter case). The arguments are counted from left to right.

In Figure 12, the actual arguments are shown as solid boxes because these represent, or identify, actual data values or locations. The dummy arguments are shown as dotted boxes to indicate that they do not represent actual data.

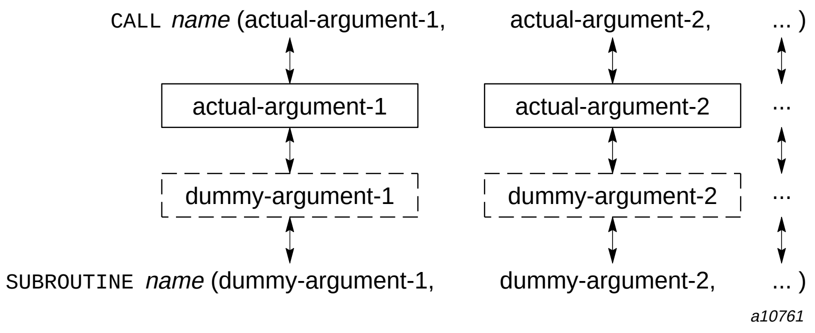


Figure 12. Actual and dummy argument lists

The principal argument association mechanism is *positional*; that is, arguments are associated according to their positions in the respective actual and dummy argument lists. The first dummy argument becomes associated with the first actual argument, the second dummy argument becomes associated with the second actual argument, and so on. The remainder of this chapter describes the rules governing this association mechanism and the forms it can take.

4.7.1 Type, kind, and rank matching

An actual argument, being a data object, has the usual set of data object attributes. These can be determined by the specification part of the calling program or, if the actual argument is an expression, by the rules governing expression results. The most important of these attributes, for argument association purposes, are the data type, kind type parameter, and rank of an object. This trio of attributes are referred to as the *TKR pattern* of the object. Thus, each actual argument, except for alternate returns and procedures as arguments, has a type, kind, rank (TKR) pattern.

Example 1: Suppose, for example, that an actual argument, `ERER`, has been specified by the following statement:

```
REAL ERER(100)
```

The TKR pattern of `ERER` is real, default kind, rank 1.

Example 2: Assume the following statement:

```
INTEGER(KIND=SELECTED_INT_KIND(1))(100, 100) IRIR
```

The TKR pattern of `IRIR` is integer, nondefault kind, rank 2.

Each dummy argument has a set of attributes just like any other data object, and this set includes a TKR pattern. The dummy argument attributes are specified, either explicitly or implicitly, in the procedure definition.

The most fundamental rule of argument association is that the TKR patterns of an actual argument and its associated dummy argument must be the same. Note that statement function references conform to these rules, except that dummy argument attributes are defined in the host.

The set of dummy arguments can be thought of as the procedure socket; that is, the means by which the procedure gets connected to the rest of the program. Each dummy argument is one of the holes in this socket. One can think of the TKR pattern of a dummy argument as determining the shape of that hole in the socket.

Similarly, the set of actual arguments in a reference to that procedure may be thought of as a plug that connects with the procedure socket (Figure 13, page 203), with each actual argument representing one prong of the plug. The TKR pattern of an actual argument determines the shape of that prong. For the connection to work properly, the shape of each plug prong must match the shape of the corresponding socket hole.

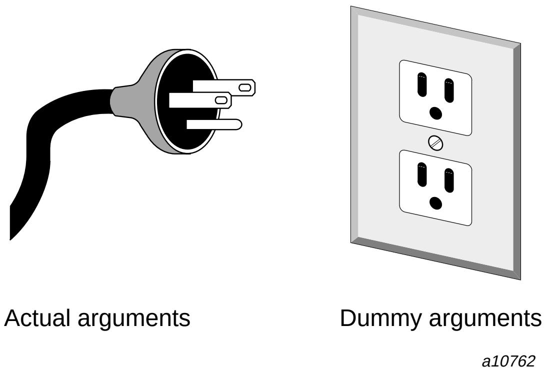


Figure 13. The plug and socket analogy for actual and dummy arguments

External procedures and their calling programs are usually compiled separately, so typically, there is no detection of TKR mismatches at compile time. Therefore, TKR argument matching is extremely prone to error, and such mismatches are among the most common and elusive errors in Fortran applications. Explicit procedure interfaces (see Section 4.8.1, page 221) solve this problem by enabling automatic detection of TKR argument mismatches.

Associated actual and dummy arguments must have the same data type, the same kind parameter for that type, and the same rank. This last part means that if one is a scalar, they both must be scalars; otherwise, they must both be arrays with the same number of dimensions. (See Section 4.7.2, page 204, for an exception to this general rule on rank matching.) Alternate returns are a special case, as are procedures used as arguments.

Argument associations involving arrays and pointers also have some special considerations; they are treated in Section 4.7.2, page 204, and Section 4.7.3, page 210, respectively. Scalar data objects without the `POINTER` attribute are discussed here. The rule is simple: the associated dummy argument for a scalar actual argument must be scalar and must have the same data type and kind type parameter value as the actual argument. The exception to this is explained in Section 4.7.3, page 210, in which an array element (which is a scalar) may be passed to a dummy array. Note that array elements, scalar-valued structure components, and substrings are valid scalar actual arguments. The only slightly complicated case involves arguments of type character because scalar character objects have an additional attribute: the character length.

The cleanest situation is when the associated actual and dummy argument character lengths are the same. This can be achieved by explicitly declaring the character length in each case, with the length value specified to be the same. In many cases this is an impossibly severe condition, however, because it prevents the development of general-purpose procedures (for example, procedures that can accept character input of any length). *Assumed length* dummy arguments alleviate this problem.

Assumed length can be specified only for dummy argument data objects. This is done by specifying an asterisk (*) for the character length. An assumed-length dummy argument does not have a length until it becomes associated with an actual argument. When it becomes associated, its length becomes the length of the actual argument. In effect, the length of the actual argument is passed as part of the actual argument and is picked up by the dummy argument in the course of argument association.

Explicit declared length is permitted for dummy arguments, however, so there must be rules governing those instances when the lengths of the actual argument and its associated dummy argument are different. For scalars, the rule is straightforward: the lengths of associated actual and dummy character arguments may be different only if the actual argument length is greater than the dummy argument length.

For arrays, the rules are somewhat complicated and are described in detail in Section 4.7.2.1, page 205.

4.7.2 Sequence association

For array arguments, the fundamental rule in Fortran 90 is that the shapes of an actual argument and its associated dummy argument must be the same. That is, they must have the same rank and the same extent (number of elements) in each dimension; thus, they also are the same size. To make this simple rule viable and to make passing array sections viable, the *assumed-shape* dummy argument was introduced in Fortran 90. Assumed-shape dummy arguments for arrays are analogous to assumed-length dummy arguments for character arguments in that assumed-shape dummy arguments assume their shape attributes from the actual argument upon association. If you use assumed-shape dummy arguments, you must ensure that the ranks of the actual and dummy arguments agree as well as the type and kind; the rest follows automatically, and association is on an element-by-corresponding-element basis. Thus, again, TKR is the only rule to observe when using assumed-shape dummy arguments. Assumed-shape dummy arguments are declared as described in Section 4.7.2.1, page 205.

In Fortran 90 an array is considered an object in and of itself, as well as a sequence of related but separate elements. Being array-element sequence oriented, the FORTRAN 77 array argument association mechanisms are geared towards associating array element sequences rather than associating array objects. These mechanisms are considerably more complicated than the simple TKR pattern matches described above, although they do offer somewhat more functionality (described below). Fortran 90, in order to be upward compatible with FORTRAN 77, provides these separate mechanisms, and much of the rest of this chapter is devoted to the description of array element sequence association.

An analogous need for object association for structures arises as well, but it is not related to FORTRAN 77 compatibility because FORTRAN 77 does not have structures. The need arises in order to accommodate the development of external programs that use structure arguments but do not have access to a module defining the type of the structure. As with arrays, the method of association is sequence association that relies on a form of storage layout, like storage association. For more information on structure sequence association, see Section 4.7.2.4, page 209.

4.7.2.1 Array element sequence association

If a dummy argument is declared as an explicit-shape array or an assumed-size array, then the ranks of the actual argument and its associated dummy argument do not have to be the same, although the types and kinds still have to match. In this case the actual argument is viewed as defining a sequence of objects, each an element of the actual array. The order of these objects is the array element order of the actual array. (Array element order is a linear sequence of the array elements obtained by varying the first subscript most rapidly through its range, then the second subscript, and so on.) The number of objects in this sequence is the size of the actual array.

Similarly, the dummy array is viewed as defining a linear sequence of dummy array elements, in array element order of the dummy array. The association of the dummy array and actual array is the association of corresponding elements in these sequences. To determine associated elements, the two sequences are superimposed with the initial elements of each corresponding with each other. This is illustrated in Figure 14, in which a three-dimensional actual array is associated with a two-dimensional dummy array. In this example, this causes the actual argument element `AA(1,2,2)` to become associated with dummy argument element `DA(4,2)`, for example. The only additional rule that needs to be observed is that the size of the dummy array cannot exceed the size of the actual array. An assumed-size dummy array extends to and cuts off at the end of the actual argument array sequence.

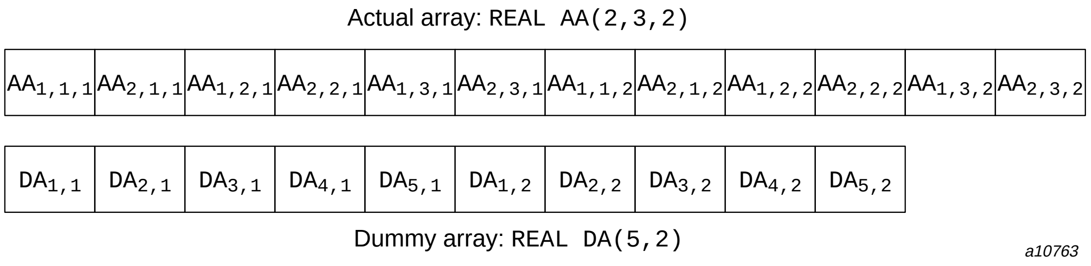

Actual array: REAL AA(2,3,2)
AA1,1,1 AA2,1,1 AA1,2,1 AA2,2,1 AA1,3,1 AA2,3,1 AA1,1,2 AA2,1,2 AA1,2,2 AA2,2,2 AA1,3,2 AA2,3,2
DA1,1 DA2,1 DA3,1 DA4,1 DA5,1 DA1,2 DA2,2 DA3,2 DA4,2 DA5,2
Dummy array: REAL DA(5,2)
a10763

Figure 14. Example of array element sequence association

For character arrays, character length is again an issue, because the array element length must be specified in this case. The lengths of the actual and dummy array elements can be different. Here, the actual and dummy arguments are viewed as sequences of characters. Each array element, in array element order, contributes a subsequence of characters the size of its length to the corresponding sequence of characters representing the argument. The argument association is then on a character-by-corresponding-character basis of these two character sequences. This can result in array element boundary crossing between the actual and dummy arguments, as illustrated in Figure 15. In this case, the size rule is that the number of characters in the dummy array cannot exceed the number of characters in the actual array. Using the example in Figure 15, these rules cause the dummy argument element `DA(4)` to be associated with the last character of the actual argument element `AA(2,1)` and the first two characters of actual argument element `AA(1,2)`.

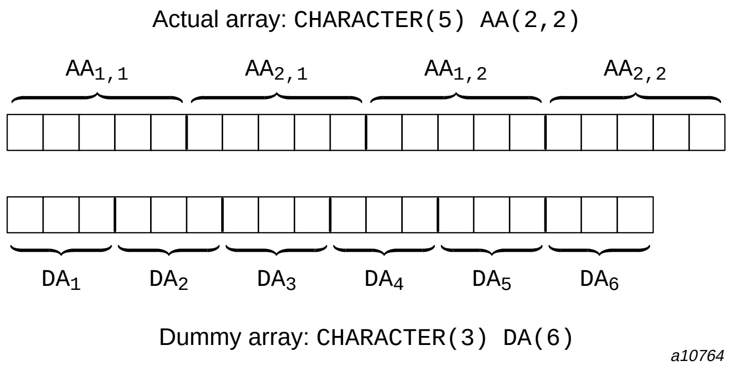

Actual array: CHARACTER(5) AA(2,2)
AA1,1
AA2,1
AA1,2
AA2,2
DA1
DA2
DA3
DA4
DA5
DA6
Dummy array: CHARACTER(3) DA(6)
a10764

Figure 15. Array element sequence association for default characters

The provision that the ranks of the actual and dummy argument arrays need not match in array element sequence association has an interesting asymmetrical end condition when one is a scalar (effectively a rank of zero). The case of the actual argument having nonzero rank and the dummy argument being scalar occurs for elemental references to intrinsic functions. The reverse, passing a scalar to a dummy array, is allowed in a limited way in array element sequence association. If the dummy argument meets the conditions for array element sequence association (that is, it is declared as an explicit-shape or assumed-size array), the actual argument can be a single array element but cannot be any other kind of scalar. This functionality was provided in FORTRAN 77 to accommodate the passing of certain forms of array sections.

In the array element sequence association paradigm, the appearance of an array element as the actual argument causes the sequence of actual array elements to begin with this element, rather than the first element of the array, and extend to the end of the array in array element order. This sequence is then associated with the full dummy argument sequence. Care must be taken to ensure that the size of the dummy array is not greater than the size of the array element sequence from the specified array element on. An element of an assumed-shape or pointer array cannot be passed to a dummy array.

An actual argument of type character can be a substring of an array element. The reason is that for character arguments, the array sequence is character based rather than array-element based. The substring provides a third way (together with an array and an array element) to specify the beginning of the actual argument character sequence. As with the other two, the sequence extends to the end of the actual array in array element order. Also, as with the other two, the number of characters in the associated dummy array must not exceed the number in the specified portion of the actual array. In addition, as in the array element case, the substring must not be from an element of an assumed-shape or pointer array.

4.7.2.2 Passing array sections

The passing of array sections in procedure references represents an important part of the array processing facility and therefore is a significant feature of Fortran 90. Assumed-shape dummy arguments provide the normal method of passing array sections.

There are three principal ways of forming an array section:

1. An array reference containing a subscript triplet

2. An array reference containing a vector subscript

3. A structure component reference in which a part other than the rightmost is array valued

An array section can also be passed to an explicit-shape or assumed-size dummy array. For reasons of compatibility with FORTRAN 77 compiled code, the array arguments of procedures with implicit interfaces are assumed to be sequence associated with the dummy arguments. In this case the section must be converted (copied) to a form acceptable for array element sequence association, and possibly reconverted upon return (for example, if it returns results from the procedure execution). Such conversion results in performance inferior to that obtained from using assumed-shape dummy arguments and is the price of passing array sections to explicit-shape or assumed-size arrays. (Note that assumed-shape dummy arguments require explicit interfaces.)

To summarize, in the case of assumed-shape dummy arguments, the TKR association rules apply. Otherwise the array element sequence association rules apply to the compacted section. One restriction that applies to the use of array sections as actual arguments, regardless of the nature of the dummy argument, is that array sections generated by vector subscripts are not definable; they must not be assigned new values by the procedure. The associated dummy argument must not have the `INTENT(OUT)` or the `INTENT(INOUT)` attribute, or be treated as if they did have either of these attributes. The reason is that with vector subscripts the same actual array element could be part of the array section more than once, and thereby this actual array element becomes associated with more than one dummy argument element. If such an object could be defined, conflicting values could be specified for the same actual array element.

4.7.2.3 Miscellaneous argument association rules

The following paragraphs contain some miscellaneous rules regarding argument association.

If the dummy argument is assumed shape, the actual argument must not be an assumed-size array. An assumed-shape dummy argument requires that complete shape information about the actual argument be supplied to the dummy argument. Because the size of an assumed-size array is open ended, complete shape information is not available for the assumed-size array. Note that a section of an assumed-size array can be used as an actual argument, provided such a section is not open ended (that is, the extent of the last dimension is explicitly specified).

The same data object coming into a procedure through two or more arguments must not be defined. For example, if `A(1:5)` is an actual argument and

A(3:9) is another actual argument, then the three elements A(3:5) have come in through two arguments. In this case none of the three elements A(3:5) can be defined in the procedure. This restriction need not involve arrays, and applies to any data object associated to two dummy arguments. If A in the above example were a character string, the same associations are possible and the same restrictions apply. Even for a simple scalar this can be the case. If K is a scalar integer variable, it may appear twice in the actual argument list, but if it does, it must not become defined as a result of a reference to the procedure.

A data object can be available to a procedure through argument association and by a different method of association. For example, it might come in as an actual argument and also be available through use or host association. In this case it can be defined and referenced only as a dummy argument. It would be illegal to assign A a value within subroutine S in the following example:

```
CALL S(A). . .
CONTAINS
   SUBROUTINE S(D)
      D = 5
   . . .
```

If the dummy argument is an array that has the POINTER attribute, it is effectively an assumed-shape dummy argument. Therefore the TKR rules apply to associated actual arguments. (The argument association rules that apply to pointers is the topic of the next section).

For generic references, defined operators, or defined assignments, the TKR method is required for all arguments and operands. Thus, an array element must not be passed to an array dummy argument under any circumstance. For example, if SQUIRT is a generic procedure name, then SQUIRT(A(7)) has a scalar actual argument and the associated dummy argument must be scalar. If the generic procedure allows both a scalar dummy argument and an array dummy argument, the specific procedure with the scalar dummy argument is selected.

An array is never passed to an intrinsic procedure if the actual argument is an array element; only the array element is passed.

4.7.2.4 Structure sequence association

If a structure is a dummy argument of an external procedure, the derived type for the structure must be specified so that it is exactly the same type as that for the corresponding actual argument. The preferred way to do this is to define a derived type in a module and access the module through a USE statement, both for the external procedure and for the program unit referencing it. An alternative way is to use structure sequence association. This technique avoids the need for a module but bypasses the error checking available in the compiler.

Two structures are *structure sequence associated* if one is an actual argument and the other is the corresponding dummy argument, and the types of the two structures are equivalent but not the same. Two derived types are equivalent if the two types have the same name, are sequence types, have no components that are private or are of a private type, and have components that agree in order, name, and attributes. Having the same attributes means having the same or equivalent type.

When a reference to a procedure is made using structure sequence association, the calling and called program units can assume a consistent storage layout that causes the structures to be associated correctly. This allows values to be passed into and out of the procedure from the calling program.

4.7.3 Pointer association

Generally, a data object may have the `POINTER` attribute, the `TARGET` attribute, or neither of these attributes, but not both. (A structure component that has the `POINTER` attribute can effectively also have the `TARGET` attribute if the structure has the `TARGET` attribute.) A dummy argument can be any of these three, as may an actual argument, but of the nine possible combinations for associated actual and dummy arguments, two are disallowed and of the remaining cases, only the five labeled `A` through `E` in Figure 16, page 211, are distinct (the other two are equivalent to the cases below them in the figure). AESA is an acronym for *Array Element Sequence Association*.

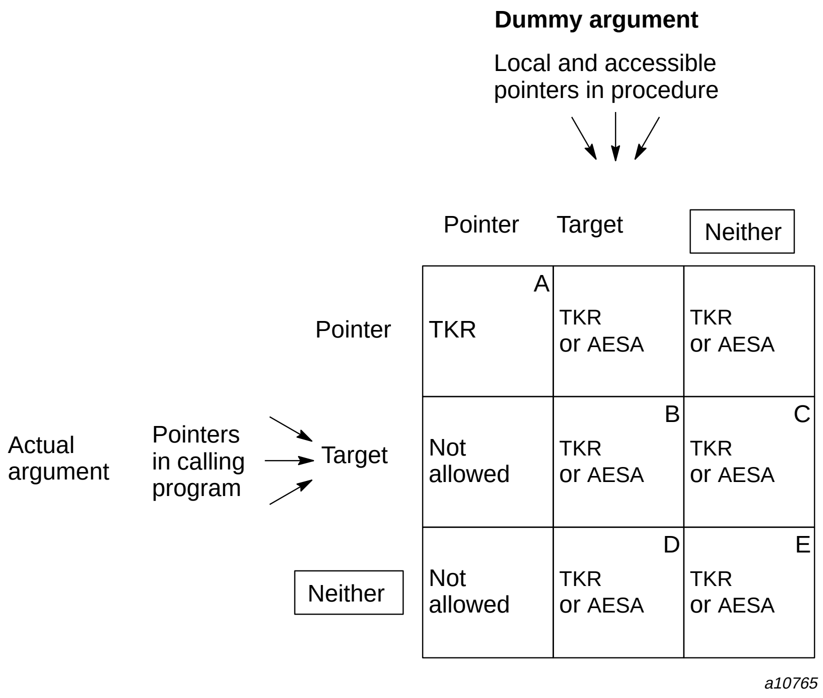


Figure 16. Association of objects with `POINTER` and `TARGET` attributes

In combination `E` in Figure 16, neither the actual or dummy argument has either the `POINTER` or `TARGET` attribute; this situation is explained in the previous two sections. Combinations `B`, `C`, and `D` are similar; they are the cases in which either the actual or dummy argument, or both, have the `TARGET` attribute. As far as argument association is concerned, these cases are very much like combination `E`; that is, either TKR or AESA applies.

Because cases `B`, `C`, and `D` in Figure 16 involve arguments with the `TARGET` attribute, there may be pointers associated with these targets. In the calling program, a target object can be used as an actual argument (cases `B` and `C`), and at the time of the call there may be pointers associated with this target. In the procedure, a dummy argument can have the `TARGET` attribute (combinations `B` and `D`), which means that during execution of the procedure a pointer, including another dummy argument with the `POINTER` attribute, may become associated with this `TARGET` argument.

The rules governing the associated pointers of target arguments are as follows:

1. During argument association, any pointers associated with a target actual argument remain pointer associated with that object, but do not become pointer associated with the dummy argument. This means that even if the actual argument has pointers pointing to it, the procedure does not know what they are, and therefore cannot use this information in any way and cannot affect any such pointer associations in any way.

2. Upon completion of procedure execution and disassociation of the actual and dummy arguments, any pointers associated with the actual argument before the call remain pointing to it.

3. For a dummy argument having the `TARGET` attribute, its pointer association status with local pointers before argument association is undefined. Upon completion of procedure execution and disassociation of the actual and dummy arguments, the pointer association status of all previously pointer-associated pointers with a dummy argument having the `TARGET` attribute becomes undefined.

In cases `B` and `D` in Figure 16, page 211, pointer association between the dummy target and pointers has no effect in the calling program. The preceding rule 3 says that even if the calling program passes a pointer and a target as two different actual arguments, pointer association between these two within the scope of the calling program cannot be established (or removed) by the called procedure.

Case `A` in Figure 16, page 211, illustrates that both the actual argument and the dummy argument can have the `POINTER` attribute. When the dummy argument is a pointer, the procedure interface must be explicit in the calling program (see Section 4.8.1, page 221), and the associated actual argument must also be a pointer. In this case the following rules apply:

1. The TKR association rules apply.

2. Upon argument association, the dummy argument acquires the same pointer association status as the actual argument and becomes pointer associated with the same target as is the actual argument, if the actual argument is associated with a target.

3. During procedure execution, the pointer association status of the dummy argument can change, and any such changes are reflected in the actual argument.

The two, unlabeled, allowed cases in Figure 16, page 211, are those in which the actual argument is a pointer but the dummy argument is not. In this case, the actual argument must be associated with a target, and it is this target that becomes argument associated with the dummy argument. Thus, these two cases are equivalent to cases B and C. In effect, the appearance of a pointer as an actual argument, without the dummy argument known to be a pointer, is treated as a pointer reference (a reference to the target). For example, calling programs can pass target-associated pointers to FORTRAN 77 procedures, because in reality the underlying (pointed to) object is passed.

The two cases in Figure 16, page 211, described as *not allowed* are illegal because the actual arguments would be incompatible with (pointer) operations allowable on the dummy argument.

In case C the procedure interface need not be explicit in the calling program. In cases A, B, and D, an explicit interface is required.

4.7.4 Argument keywords

The fundamental pairing of arguments in the actual argument list with those in the dummy argument list is positional, as shown in Figure 12, page 201, but Fortran 90 also provides an order-independent way of constructing actual argument lists. With this option the programmer can explicitly specify in the call which actual argument is to be associated with a dummy argument rather than using its position in the actual argument list to determine the pairing. To do this, the name of the dummy argument, which in this context is referred to as a *keyword*, is specified in the actual argument list along with the actual argument. It takes the following form:

dummy_argument_name = *actual_argument*

This form can be used for all of the actual arguments in an actual argument list, and the arguments can be in any order. A reference can use keywords for only some of the actual arguments. For those actual arguments not having keywords, the positional mechanism is used to determine the associated dummy arguments. Positionally associated actual arguments must appear in the actual argument list before the keyword actual arguments. After the appearance of the first keyword actual argument (if any) in the actual argument list, all subsequent actual arguments must use keywords. This is shown in the following examples:

```
CALL GO(X, HT=40)
CALL TELL(XYLOPHONE, NET=10, QP=PI/6)
```

Thus, when only some arguments in the actual argument list use keywords, the first part is positional, with no argument keywords, and the last part uses keywords. In the keyword portion of the list the order of the arguments is completely immaterial, and the keyword alone is used to determine which dummy argument is associated with a given actual argument. Care must be taken with keyword arguments in each call to make sure that one and only one actual argument is specified for each required dummy argument and that at most one actual argument is specified for each dummy argument.

Keyword actual argument lists can aid readability by decreasing the need to remember the precise sequence of dummy arguments in dummy argument lists. This functionality, and the form that it takes, is modeled after keyword specifiers in I/O statements. The one situation that requires keyword arguments is when an optional argument not at the end of the argument list is omitted; keyword arguments constitute the only way to skip such arguments in an actual argument list.

To use keyword actual arguments, the procedure interface must be explicit in the scope of the program containing the reference. The compiler can generate the proper reference in this case because the defined sequence of dummy argument names (keywords) is known to the calling program. Intrinsic, internal, and module procedure interfaces are always explicit, so keyword references can be used with these. An interface block must be provided in the calling program for an external procedure before keyword references can be made to it. The price of an interface block comes with an interesting benefit not available from the automatic explicitness of intrinsic, internal, and module procedure interfaces — the keywords do not have to be the same as the dummy argument names in the procedure definition. This ability to tailor the argument keywords to the application is available only with external procedures.

4.7.5 Optional arguments

Fortran 90 includes the ability to specify that an argument be optional. An actual argument need not be supplied for a corresponding optional dummy argument in a particular reference, even though it is in the list of dummy arguments. In this case, an optional argument is specified by giving the dummy argument the `OPTIONAL` attribute, either in an entity-oriented declaration that includes the `OPTIONAL` attribute or by its inclusion in an `OPTIONAL` statement in the procedure definition. The `OPTIONAL` attribute can be specified only for a dummy argument. Any dummy argument in any procedure can be specified to be optional.

In a positional argument list, an optional argument at the right-hand end of the list can be simply omitted from the reference. To omit an argument from the *keyword* part of an actual argument list, that dummy argument name is not used as one of the keywords. Note that the keyword technique must be used to omit an optional argument that is not at the end of the list, unless all of the remaining arguments are also being omitted in this reference. This is shown in the following example:

```
CALL TELL(1.3, T=F(K))
   . . .
SUBROUTINE TELL (X, N, T)
   OPTIONAL N, T
      . . .
END
```

Optional arguments require explicit procedure interfaces.

During execution of a procedure with an optional dummy argument, it is usually necessary to know in that particular reference if an actual argument has been supplied for that dummy argument. The PRESENT intrinsic function is available for that purpose. It is an inquiry function and has one argument, the name of an optional argument in the procedure. Upon execution it returns a value of default logical type, depending on whether or not the dummy argument is associated with an actual argument.

```
IF (PRESENT(NUM_CHAR)) THEN
  ! Processing if an actual argument has been
  ! supplied for optional dummy argument NUM_CHAR
   USABLE_NUM_CHAR = NUM_CHAR
ELSE
  ! Processing if nothing is supplied for NUM_CHAR
   USABLE_NUM_CHAR = DEFAULT_NUM_CHAR
END IF
```

The preceding example illustrates how you can use the PRESENT function to control the processing in the procedure as is appropriate depending on the presence or absence of an optional argument. For an optional dummy argument not present (corresponding actual argument not supplied), the following rules apply:

- A dummy argument not present must not be referenced or defined.
- A dummy procedure not present must not be invoked.
- A dummy argument not present must not be supplied as an actual argument corresponding to a nonoptional dummy argument, except in a reference to the PRESENT intrinsic function.

- A dummy argument not present may be supplied as an actual argument corresponding to an optional dummy argument. In this case, the latter dummy argument is also considered to be not present.

4.7.6 Argument intent

Any dummy argument, except a procedure or pointer, can be given an INTENT attribute (see the *Fortran Language Reference Manual, Volume 1*, publication SR–3902). There are three possible formats for this attribute:

```
 INTENT(IN)
INTENT(OUT)
INTENT(INOUT)
```

The INTENT attribute can be specified only for dummy arguments and indicates something about the intended use of the argument in the procedure. The use of this attribute enables the compiler to detect uses of the argument within the procedure that are inconsistent with the intent.

INTENT(IN) specifies that an argument is to be used to input data to the procedure, is therefore defined upon entry to the procedure, is not to be used to return results to the calling program, and must not be redefined by the procedure.

INTENT(OUT) specifies that an argument is to be used to return results to the calling program and cannot be used to supply input data. A dummy argument with INTENT(OUT) must not be referenced within the procedure before it is defined. The actual argument associated with an INTENT(OUT) dummy must be a definable data object, that is, a variable.

INTENT(INOUT) specifies that an argument has a defined value upon entry to the procedure and that this value can be redefined during execution of the procedure; it can be referenced before being changed. This would be the intent, for example, of a data object whose value is to be updated by the procedure. The actual argument associated with an INTENT(INOUT) dummy must be a definable data object.

Actual arguments that are array sections with vector-valued subscripts are not allowed to be associated with dummy arguments having INTENT(OUT) or INTENT(INOUT); that is, the associated dummy argument must not be defined.

The use of the INTENT attribute for a dummy argument does not require an explicit interface, because it governs use within the procedure. Making the interface of a procedure containing an `INTENT(OUT)` or `INTENT(INOUT)` dummy argument explicit in the calling program, although not required, can nevertheless be useful in detecting possible attempts to use nondefinable data objects for the associated actual argument.

If the `INTENT` attribute is not specified for a dummy argument, it can be referenced only if the actual argument can be referenced. It can be defined only if the actual argument can be defined.

4.7.7 Resolving references to generic procedures

Two or more procedures are generic if they can be referenced with the same name. With such a reference it must be possible to determine which of the procedures is being called. That is, a generic reference must be resolved to that specific procedure in the set of procedures sharing the generic name to which the call applies. The distinguishing property of the reference that is used for this resolution is the nature of the actual argument list. The sequence of TKR patterns in the actual argument in effect selects the appropriate specific procedure to be called.

The set of specific procedures making up the generic set are restricted such that any given sequence of actual argument TKR patterns will match only one of the dummy argument lists of this set. Thus, the requirement of TKR matches in the argument lists can be used to resolve generic references. The operational rules follow; for further details see Section 6.1.7, page 270. Considering the dummy argument lists of any two procedures in the generic set, one of them must have a required dummy argument that satisfies both of the following conditions:

- It must be in a position in the list at which the other list has no dummy argument or it has a TKR pattern different from that of the dummy argument in the same position in the other list.
- It must have a name different from all the dummy argument names in the other list or it must have a TKR pattern different from that of the dummy argument with the same name in the other list.

The reason for the second of these rules is the need for unique resolution with respect to references with keyword arguments, as well as strictly positional actual argument lists. Section 4.8.3, page 228, contains an example that illustrates why just the first rule is not enough.

Because resolution of a generic reference requires matching the actual argument list with the candidate dummy argument lists from the generic set, clearly the interfaces of the procedures in the generic set must be explicit in the scoping unit containing the reference. How this is done and how a procedure is added to a generic set is described in Section 4.8.3, page 228.

4.7.8 Elemental references to intrinsic procedures

As mentioned previously, in certain special cases involving references to intrinsic procedures, the rule that disallows passing an actual array argument to a scalar dummy argument is relaxed. Many intrinsic procedures have scalar dummy arguments, and many of these can be called with array actual arguments. These are called *elemental* intrinsic procedures.

Elemental functions are defined to have scalar results as well as scalar dummy arguments. For an elemental intrinsic function with one argument, calling that function with an array argument causes the function to be applied to each element of that array, with each application yielding a corresponding scalar result value. This collection of result values, one for each element of the actual argument, is returned to the calling program as the result of the function call in the form of an array of the same shape as the actual argument. Thus, the function is applied element-by-element (hence the term *elemental reference*) to the actual argument, resulting in an array of the same shape as the argument and whose element values are the same as if the function had been individually applied to the corresponding elements of the argument.

The square root function SQRT is an example of an elemental intrinsic function. Its dummy argument is a scalar, and it returns a scalar result. The following example shows a typical reference to SQRT:

```
Y = SQRT(X)
```

If both X and Y are scalar variables, this would be a normal call to SQRT. Now assume that X and Y are both one-dimensional arrays with bounds X(1:100) and Y(1:100). The previous assignment statement is still valid and has a result equivalent to the following:

```
DO J = 1, 100
   Y(J) = SQRT(X(J))
END DO
```

The elemental call to SQRT, however, does not imply the ordering of the individual computations that is specified by the DO construct.

According to the rules of intrinsic assignment, X and Y must have the same shape.

An elemental procedure that has more than one dummy argument can still be called elementally if all of the dummy arguments and the result are scalar and if the actual arguments are conformable. The exception to this, however, is if the name of the dummy argument is KIND (which means its value is a kind type value), its associated actual argument must be scalar. The other actual arguments can be scalars or arrays, as long as they are conformable, and the result has the shape of these arrays or a scalar. The KIND actual argument specifies the kind value for the resulting array.

All of the elemental intrinsic procedures are identified as such in the man pages for each intrinsic procedure. Many have multiple arguments (including the single elemental intrinsic subroutine MVBITS) and many have a KIND dummy argument. In the case of MVBITS there is no function result (one of the arguments returns the result), but the rule for conformable actual arguments is the same as for the elemental functions.

4.7.9 Alternate returns

An alternate return is one of the two kinds of procedure arguments that are not data objects. The other is a dummy procedure. Alternate returns can appear only in subroutine argument lists. They are used to specify a return different than the normal execution upon completion of the subroutine.

There can be any number of alternate returns in a subroutine argument list, and they can appear at any position in the list. In the dummy argument list each alternate return is simply an asterisk. For example, the following dummy argument list for subroutine CALC_2 has two alternate return indicators, in the second and fifth argument positions:

```
SUBROUTINE CALC_2(A, *, P, Q, *)
```

Alternate returns cannot be optional, and the associated actual arguments cannot have keywords.

An actual argument associated with an alternate return dummy argument must be an asterisk followed by a label of a branch target in the scope of the calling program. These actual arguments specify the return points for the corresponding alternate returns. For example, the following is a valid reference to CALC_2:

```
CALL CALC_2(X, *330, Y, Z, *200)
```

This assumes that the statements labeled 200 and 330 are branch targets. The statement having the label 330 is the return point for the first alternate return, and the statement having the label 200 is the return point for the second alternate return.

Using an alternate return is done with the extended form of the RETURN statement, which is defined as follows:

```
RETURN scalar_int_expr
```

The scalar integer expression must have an integer value between 1 and the number of asterisks in the dummy argument list, inclusive. The integer scalar expression value selects which of the alternate returns, counting from left to right, is to be utilized. Assuming the preceding example call, the following results can be expected:

```
RETURN  2  ! returns to statement 200 in
           !   the calling program
RETURN (1) ! returns to statement 330
RETURN     ! normal return from the call
```

Alternate returns are an obsolescent feature.

4.7.10 Dummy procedures

A dummy argument can be a name that is subsequently used in the procedure as a procedure name. That is, it can appear in an interface block, in an EXTERNAL or INTRINSIC statement, or as the name of the procedure referenced in a function or subroutine reference. The associated actual argument must be the name (without an argument list) of an external, module, intrinsic, or dummy procedure.

The actual argument must not be the name of an internal procedure or a statement function.

The actual argument must not be a generic procedure name, unless there is a specific procedure with the same name; only specific procedures may be passed in argument lists.

If the interface of the dummy procedure is explicit, the associated actual procedure must be consistent with this interface as described in Section 4.8.2.2, page 228.

If the dummy procedure is typed, referenced as a function, or has an explicit function interface, the actual argument must be a function.

If the dummy procedure is referenced as a subroutine or has an explicit subroutine interface, the actual argument must be a subroutine.

Dummy procedures can be optional, but must not have the `INTENT` attribute. They can occur in either function or subroutine subprograms. The associated actual argument can be specified by using a keyword.

4.8 Procedure interfaces

The term *procedure interface* refers to those properties of a procedure that interact with or are of direct concern to a calling program in referencing the procedure. These properties are the names of the procedure and its dummy arguments, the attributes of the procedure (if it is a function), and the attributes and order of the dummy arguments. If these properties are all known to the calling program (that is, known within the scope of the calling program), the procedure interface is said to be *explicit* in that scope; otherwise, the interface is *implicit* in that scope.

An *interface block* can be used in the specification part of a scoping unit to make explicit a procedure interface (other than a statement function) that otherwise would be implicit in that scoping unit. In addition, interface blocks:

- Allow you to give generic properties to procedures, including extending intrinsic procedures
- Define new user-defined operators and extend the generic properties of intrinsic operators
- Extend the assignment operation to new data combinations (user-defined coercions)
- Specify that a procedure is external

The following sections describe the roles of explicit interfaces, situations in which explicit interfaces are needed, when a procedure definition provides an explicit interface, and all of the uses of interface blocks.

4.8.1 Explicit interfaces

The following situations require explicit interfaces:

1. Optional arguments
2. Array-valued functions
3. Pointer-valued functions

4. Character-valued functions whose lengths are determined dynamically
5. Assumed-shape dummy arguments (needed for efficient passing of array sections)
6. Dummy arguments with the `POINTER` or `TARGET` attribute
7. Keyword actual arguments (which allow for better argument identification and order independence of the argument list)
8. Generic procedures (calling different procedures with the same name)
9. User-defined operators (which is just an alternate form for calling certain functions)
10. User-defined assignment (which is just an alternate form for calling certain subroutines)

Explicit interfaces are required for items 1 and 7 in this list so that the proper association between actual and dummy arguments can be established. Recall that any of the arguments in the dummy argument list can be declared to be optional and any such argument can be omitted in a reference to the procedure. Keyword arguments allow actual arguments to occur in any order and are needed when omitting an optional argument that is not the last in the dummy argument list. Consider, for example, the following procedure:

```
SUBROUTINE EX (P, Q, R, S, T)
OPTIONAL Q, R, T
```

The following calls are all valid:

```
CALL EX(V, W, X, Y, Z)

CALL EX(V, W, X, Y)
                  ! Last argument omitted

CALL EX(P=V, Q=W, R=X, S=Y, T=Z)
                  ! Same as first call

CALL EX(P=V, Q=W, R=X, S=Y)
                  ! Same as second call

CALL EX(P=V, S=Y)
                  ! All optional arguments omitted

CALL EX(S=Y, P=V) ! Same as fifth call
CALL EX(R=X, S=Y, P=V, Q=W)
                  ! Same as second call

CALL EX(S=Y, T=Z, Q=W, P=V)
                  ! An optional argument omitted
```

The last four of these example CALL statements illustrate why explicit interfaces are needed for optional and keyword arguments. Namely, so that the calling routine knows the names of the dummy arguments in order that the proper subroutine reference can be generated.

Items 2 and 4 in the preceding list involve function results whose size (number of items returned) is determined by the procedure and may be different from reference to reference. Explicit interfaces convey the necessary information to the calling routines to process such references correctly.

In item 3, the calling procedure needs to know that there is a layer of indirection (the pointer) buffering the actual data involved.

Item 5 represents a significant new functionality in Fortran 90 — one that requires additional information in the calling program that is provided by an explicit interface. With discontiguous arrays, additional information must be passed in the call. The FORTRAN 77 array passing mechanism requires contiguous arrays and does not require passing the array shape and size. In contrast, Fortran 90 also allows passing sections of arrays as arguments, which can comprise array elements discontiguous in storage, and can represent very sparse array sections. Assumed-shape dummy arrays are provided to accommodate this new form of array passing. Any array, contiguous or not, can be passed to either form of dummy argument, assumed-shape or otherwise. For implicit interfaces, an explicit-shape or assumed-size (FORTRAN 77 style) dummy argument is the default, and, therefore, assumed-shape arguments require explicit interfaces. Discontiguous array sections can be passed to explicit-shape or assumed-size dummy arguments, but in this case the CF90 and MIPSpro 7 Fortran 90 compilers pack such sections into contiguous temporaries on entry to the procedure and unpack them on return, incurring performance penalties.

Dummy arguments with the POINTER or TARGET attribute (item 6) require explicit interfaces because an actual argument can be a pointer, but what is passed can be either the pointer itself or the target. Which one is passed depends upon what the procedure expects, and whether or not the dummy argument has the POINTER attribute. The explicit interface provides the required information. The default for implicit interfaces is that the target is passed.

Generic procedures (item 8) must have explicit interfaces because the calling routine must be able to disambiguate a generic procedure name; that is, because *generic* means two or more procedures with the same name, the calling routine must have enough information to determine which specific procedure to invoke for a given reference. An explicit interface provides this information by making dummy argument attribute information available to the calling routine. The specific procedure called is the one for which the dummy argument attribute pattern matches that for the actual arguments. Generic procedures, including the use of interface blocks for configuring generic names, are discussed in detail in Section 4.8.3, page 228.

User-defined operators (item 9) represent an alternative way to reference certain functions. They allow the use of infix operator notation, rather than traditional function notation for two-argument functions. Thus, for example, `A + B` can be used in place of `RATIONAL_ADD(A, B)` if `A` and `B` are of the derived type representing rational numbers, and the operator `+` has been extended as the operator form of `RATIONAL_ADD`. An example of using a new operator, rather than extending an intrinsic operator, is `P .SMOOTH. 3`, where `P` represents a matrix of picture elements to be smoothed and `3` is the size of the smoothing neighborhood. This is alternative syntax for the function reference `PICTURE_SMOOTH(P,3)`. Such alternative syntax provisions are similar to generic procedures, and the same sort of interface information is needed to resolve such references. This topic is treated in detail in Section 4.8.4, page 231.

A third form of generic interface for which explicit interface information is used to resolve references is that of assignment coercion (user-defined assignment). This allows a value of one data type to be converted to a corresponding value of another data type and assigned to an object of the latter type. A simple example of coercion, intrinsic to Fortran, is `K = X + 2.2`, where `X` is of type real and `K` is integer. Examples of desirable new coercions might be `R = K` where `R` is of the derived type representing rational numbers and `K` is an integer, and `P = M2D` where `P` is of the derived type representing a two-dimensional picture and `M2D` is a two-dimensional integer array. These last two effects could also be achieved by subroutine calls, as follows:

```
CALL RATINT(R, K)
CALL PXINIT(P, M2D)
```

These coercion operations are performed by the same subroutines, but these subroutines can be invoked by the assignment syntax rather than the traditional subroutine call syntax. This topic is treated in more detail in Section 4.8.5, page 233.

4.8.2 Interface blocks

A procedure interface block is used to perform the following functions:

1. Make explicit interfaces for external and dummy procedures.
2. Define a generic procedure name, specify the set of procedures to which that name applies, and make explicit the interfaces of any external procedures included in the set.
3. Define a new operator symbol or specify the extension of an intrinsic or already defined operator, identify the function or functions to which it applies, and make explicit the interfaces of any of those functions that are external.
4. Define one or more new assignment coercions, identify the subroutine or subroutines involved, and make explicit the interfaces of any of those subroutines that are external.

In all of these cases, the purpose of the interface block is to make the necessary information available to the calling routine so that a procedure reference can be processed correctly. Therefore, the interface block must either appear in the specification part of the calling routine or be in a module used by the calling routine.

Of the previous four items, the first is further described in the following sections. The others were described in previous sections. In the following sections, the general form of interface blocks, covering all four of these cases, is described, followed by a discussion of the simplified form that applies to just the first case.

4.8.2.1 General form of procedure interface blocks

An interface block has the following general format:

```
INTERFACE [ generic_spec ]
   [ interface_body ] ...
   [ MODULE PROCEDURE procedure_name_list ] ...
   END INTERFACE
```

Interface blocks are defined as follows:

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

An *interface_body* specifies the interface for either a function or a subroutine.

The following is the general format of a function interface body:

```
function_statement
   [ specification_part ]
   END [ FUNCTION [ function_name ] ]
```

The following is the general format of a subroutine interface body:

```
subroutine_statement
   [ specification_part ]
   END [ SUBROUTINE [ subroutine_name ] ]
```

The keyword FUNCTION or SUBROUTINE is optional on an interface body's END statement, but use of these keywords is recommended as an aid to program clarity.

The form without the *generic_spec* applies to case 1 in the list of four in Section 4.8.2, page 225. The choice of a *generic_name* for a *generic_spec* is case 2. The OPERATOR choice for a *generic_spec* is case 3. The ASSIGNMENT choice for a *generic_spec* is case 4.

The following guidelines apply to interface blocks:

1. If the *generic_spec* is omitted, the interface block must not contain any MODULE PROCEDURE statements.

2. In all cases, an interface body must be for an external or dummy procedure.

3. The *specification_part* of an interface body contains specifications pertaining only to the dummy arguments and, in the case of functions, the function result. This means, for example, that an interface body cannot contain an ENTRY statement, DATA statement, FORMAT statement, or statement function statement.

4. The attributes of the dummy arguments and function result must be completely specified in the specification part, and these specifications must be consistent with those specified in the procedure definition. Note that dummy argument names may be different, but the attributes must be the same.

5. Because an interface body describes properties defined in an external scope rather than in its host's scope, an interface body comprises its own scoping unit, separate from any other scoping unit; an interface body does not inherit anything from its host via host association, such as named constants nor does it inherit its host's implicit type rules.

   An interface body may contain a USE statement and access entities, such as derived-type definitions, via use association.

6. A procedure name in a MODULE PROCEDURE statement must be the name of a module procedure either in that module (if the interface block is contained in a module) or accessible to the program unit containing the interface block through use association. It must not appear in another interface with the same generic specifier in the same scoping unit or in an accessible scoping unit.

7. An interface block must not contain an ENTRY statement, but an entry interface can be specified by using the entry name as the function or subroutine name in an interface body.

8. A procedure must not have more than one explicit interface in a given scoping unit.

9. An interface block must not appear in a block data program unit.

4.8.2.2 Explicit interfaces for external procedures

The simplest use of an interface block is to make the interface for an external or dummy procedure explicit. The format of the interface block for this purpose is as follows:

```
INTERFACE
   interface_body ...
   END INTERFACE
```

Guidelines 2 through 5 and 7 through 9, from Section 4.8.2.1, page 225, apply in this case. Rule 5 means that IMPLICIT statements, type declarations, and derived-type definitions in the host do not carry down into an interface body. Rule 8 means that an interface body for a given external procedure may be specified at most once in a host program unit. It also means that an interface body cannot be specified for intrinsic, internal, and module procedures, because these procedures already have explicit interfaces. This is the reason for the MODULE PROCEDURE statement's existence.

4.8.3 Generic procedures

Fortran programmers are familiar with generic procedures because many of the intrinsic procedures in Fortran are generic. The following are examples of generic procedure references:

```
INT(R)
INT(D)
```

Assume in the previous example, that R and D are real and double precision objects, respectively. It looks like there is only one procedure involved here (INT), but there are really two. There is a specific procedure that accepts a real argument and another one that accepts a double-precision argument. Because the purpose of these two procedures is virtually identical, it is desirable to refer to each of them with the same generic name. The type of the argument is sufficient to identify which of the specific procedures is involved in a given reference.

Thus, *generic* refers to a set of different procedures with different specific names that all have the same (generic) name. The mechanism for declaring generic procedures is the interface block, which in this case has the following format:

```
INTERFACE generic_name
   [ interface_body ] ...
   [ MODULE PROCEDURE procedure_name_list ] ...
   END INTERFACE
```

Rules and restrictions 2 through 9 in Section 4.8.2.1, page 225, apply in this case. The generic name in the INTERFACE statement, of course, specifies the generic name to be used in this procedure. All the procedures being assigned this generic name are specified in the interface block. This potentially includes both external procedures and module procedures. In the case of an external procedure, the procedure is identified by its specific name and its interface body, thereby making its interface explicit as well as defining a generic name for it. In the case of a module procedure, only the specific name of the procedure is given (in order to identify the procedure) because its interface is already explicit. Note that because of rule 8 an external procedure can be included in only one generic set in a given procedure. Because the MODULE PROCEDURE statement does not specify an explicit interface, however, a module procedure may be included in any number of generic sets.

Note also that internal procedures cannot be given generic names, nor can statement functions. Similarly, intrinsic procedures cannot be included in an interface block, but a generic name may be the same as an intrinsic procedure name, including a generic intrinsic procedure name.

Consider the following example:

```
INTERFACE INT
   MODULE PROCEDURE RATIONAL_TO_INTEGER
END INTERFACE
```

The generic properties of the INT intrinsic function are extended to include a user-defined procedure.

The generic name can also be the same as one of the specific names of the procedures included in the generic set, or the same as any other generic name, or completely different. The only real requirement is that any procedure reference involving a generic procedure name be resolvable to one specific procedure. Thus, for example, the generic name INT cannot be applied to a user-defined function that has a single real argument, because then a reference to INT with a real actual argument would be ambiguous as to whether the reference was to the corresponding intrinsic function or to the user-defined function.

The rules for resolving a generic reference involve the number of arguments and the type, kind type parameter, and rank of each argument. Based upon these rules, and only these rules, a given procedure reference must be consistent with precisely one of the specific procedures in the generic set. Consider, for example, a simple two-argument subroutine G(P, Q) with generic name G, dummy argument names P and Q, and neither argument optional. A reference to G, with actual arguments X and Y could take any of the following four forms:

```
CALL G(X, Y)
CALL G(X, Q=Y)
CALL G(P=X, Q=Y)
CALL G(Q=Y, P=X)
```

The last three are allowed because the interface to G is explicit and keyword references can be used when the interface is explicit. What subroutine H could be added to the generic set with G? The first of the preceding four calls rules out any two-argument H whose first argument has the same type, kind type parameter, and rank (TKR) as the P argument of G and whose second argument has the same TKR as the Q argument of G. The third and fourth of these four calls rules out any subroutine H of the form H(Q, P), whose first argument is named Q and has the same TKR as the Q (second) argument of G and whose second argument is named P and has the same TKR as the P (first) argument of G. The reason for this last case is that a reference to H in which all the actual arguments had keywords would look exactly like a call to G, in terms of TKR patterns; such a reference would not be uniquely resolvable to either a call to G only or to H only. Any other H could be included in the generic set with G.

Thus, the essence of the generic reference resolution rules is uniqueness with respect to TKR patterns under both positional or keyword references. The complete formal rules for this are described in Section 6.1.7, page 270.

A procedure can always be referenced by its specific name. It can also be referenced by any generic name it might also have been given.

4.8.4 Defined operators

Just as generic names allow you to give procedures alternative and presumably better forms of reference, so do defined operators. In this case functions of one or two nonoptional arguments are involved.

Often the purpose of a function is to perform some computation (operation) on the values represented by its arguments and to return the result for computational use in the calling program. In mathematical tradition, such operations of one or two arguments are usually expressed as operators in an expression, with the arguments as the operands. A good example is the `INVERSE` function described in Section 3.6.5.4, page 164. The defined operator provisions of the interface block give you the option of specifying a function reference with operator syntax.

The format of the interface block for defining a new operator or extending the generic properties of an existing operator is as follows:

```
INTERFACE OPERATOR (defined_operator)
   [ interface_body ] ...
   [ MODULE PROCEDURE procedure_name_list ] ...
   END INTERFACE
```

The same rules apply here as in the generic name case. In addition, each interface body must be for a one- or two-argument function, and each procedure name in the `MODULE PROCEDURE` statement must be that of a one- or two-argument function. The arguments are all required and all must be specified with `INTENT(IN)`.

The defined operator in the `INTERFACE` statement specifies the operator that can be used in the operation form of reference for each of the functions identified in the interface block. The operation takes the infix (operator between the arguments) form for two-argument functions and takes the prefix form for one-argument functions.

Consider the following example:

```
INTERFACE OPERATOR(+)
   FUNCTION INTEGER_PLUS_INTERVAL(X, Y)
      USE INTERVAL_ARITHMETIC
      TYPE(INTERVAL)     :: INTEGER_PLUS_INTERVAL
      INTEGER, INTENT(IN)        :: X
      TYPE(INTERVAL), INTENT(IN) :: Y
   END FUNCTION INTEGER_PLUS_INTERVAL
   MODULE PROCEDURE RATIONAL_ADD
END INTERFACE
```

This code extends the + operator to two user-defined functions, an external function INTEGER_PLUS_INTERVAL that presumably computes an appropriate value for the sum of an integer value and something called an interval, and a module function RATIONAL_ADD that probably computes the sum of two rational numbers. Both functions now can be called in the form A+B, where A and B are the two actual arguments.

The following example shows a new operator being defined, rather than extending an existing operator:

```
INTERFACE OPERATOR(.INVERSETIMES.)
    MODULE PROCEDURE MATRIX_INVERSE_TIMES
END INTERFACE
```

In this example, the inverse of matrix A can be multiplied by B using the expression `A .INVERSETIMES. B`, which produces $A^{-1}xB$, and in effect solves the system of linear equations, A*x*=B, for *x*.

Functions with operator interfaces can be referenced with the operator form, but they also can be referenced through the traditional functional form by using the specific function name.

The two forms for a defined operator (given by R311, R704, R724) are as follows:

intrinsic_operator

.letter [*letter*] ... .

Note that these are the same as some of the intrinsic operators and that neither .TRUE. nor .FALSE. can be chosen as a defined operator. Note also that if an operator has the same name as a standard intrinsic operator, it must have the same number of operands as the intrinsic operator; for example, .NOT. must not be defined as a binary operator. The masking or boolean operators that are extensions to the Fortran 90 standard can be redefined in an interface. In this case, your definition overrides the masking operator.

Operator interfaces define a set of generic procedures, with the operator being the generic name. This is particularly obvious with intrinsic operators, as each intrinsic operation may be thought of as being performed by a hidden intrinsic function and the operator interface merely extends the set of functions that share that operator form. As with the use of generic procedure names, a function reference through a generic operator must resolve to a unique specific function. The resolution rules in this case are exactly the same TKR rules used for the generic name case when the functional form is used (see the previous section), but are somewhat simpler when the operator syntax is used because this syntax does not allow the use of argument keywords. Thus, the argument TKR pattern must be unique solely on the basis of argument position.

This means, for example, that + cannot be specified in an operator interface for a function with a scalar integer argument and a scalar real argument because + already has a meaning for any such TKR pattern. Specifying such an operator extension would mean that `I+R`, where `I` is a scalar integer and `R` is a scalar real, would be ambiguous between the intrinsic meaning and the extended meaning. Therefore, the TKR rules disallow such extensions.

4.8.5 Defined assignment

The last form of generic procedures is assignment (or conversion) subroutines. These specify the conversion of data values with one set of TKR attributes into another set with a different pattern of TKR attributes. Although such conversions can be performed by ordinary subroutines, it is convenient and natural to express their use with assignment syntax, and hence defined assignment extensions. If you want to think of assignment as an operation, then defined assignment is precisely the same as defined operators, except that there is only one defined assignment operator symbol (=), and all defined assignment procedures are two-argument subroutines rather than functions. As with the other forms of generic procedures, the interface block is used to specify defined assignments.

The format of the interface block for defining new assignment operations is as follows:

```
INTERFACE ASSIGNMENT (=)
   [ interface_body ] ...
   [ MODULE PROCEDURE procedure_name_list ] ...
   END INTERFACE
```

Again, most of the same rules apply here as in the generic name case, except that each interface body must be for a two-argument external subroutine and each procedure name in the `MODULE PROCEDURE` statement must be that of an accessible two-argument module subroutine. Both arguments are required. The first argument must have the attribute `INTENT(OUT)` or `INTENT(INOUT)`; this is the location for the converted value. The second argument must have the attribute `INTENT(IN)`; this is the value to be converted.

The assignment interface specifies that an assignment statement can be used in place of a traditional subroutine call for the subroutines identified in the interface block.

Recall that an assignment statement is defined as follows:

variable = *expression*

The *variable* would be the first actual argument in a traditional call to the subroutine and the *expression* would be the second argument. The *variable* must be a variable designator legitimate for the left-hand side of an assignment.

You can use the traditional subroutine call or the assignment syntax.

An example of an assignment interface block is as follows:

```
INTERFACE ASSIGNMENT(=)
   SUBROUTINE ASSIGN_STRING_TO_CHARACTER(C, S)
      USE STRING_DATA
      CHARACTER(*), INTENT(OUT) :: C
      TYPE(STRING), INTENT(IN)  :: S
   END SUBROUTINE ASSIGN_STRING_TO_CHARACTER
   MODULE PROCEDURE  RATIONAL_TO_INTEGER
END INTERFACE
```

This interface block allows `ASSIGN_STRING_TO_CHARACTER` (which extracts the character value) to be called in the following form:

```
C = S
```

In addition, `RATIONAL_TO_INTEGER` can be called in the following format:

```
K = R
```

Here, `R` is of derived type `RATIONAL` and `K` is an integer. The purpose of `RATIONAL_TO_INTEGER` is to convert a value in `RATIONAL` form into an integer.

Each intrinsically defined assignment operation may be thought of as being performed by a hidden intrinsic subroutine. Thus, the assignment symbol is the generic name of a set of generic subroutines. The assignment interface block allows you to add user-defined external and module subroutines to that generic set. Any given assignment must be resolvable to the specific subroutine. Not surprisingly, when assignment syntax is used, it is the TKR pattern rules without the keyword argument complication that are applied to perform this resolution, exactly as with defined operators.

This means, for example, that an assignment interface cannot be specified for a subroutine whose first argument is a scalar of type integer and whose second argument is a scalar of type real. All such coercions have intrinsic meanings, and thus an assignment interface block of this form would introduce an unresolvable ambiguity. The TKR rules prevent this from happening.

Intrinsic Procedures [5]

The Fortran 90 standard defines intrinsic procedures, which consist of intrinsic functions and intrinsic subroutines. The CF90 and MIPSpro 7 Fortran 90 compilers include several more intrinsic procedures as extensions to the standard.

Intrinsic procedures can be called from any program unit or subprogram. However, a user-written function or subroutine with the same name as an intrinsic function or subroutine takes precedence over the intrinsic procedure in a given scoping unit if one of the following is true:

- Its interface is explicit.
- It is listed in an `EXTERNAL` statement in that scoping unit.
- It is a statement function.

A module or internal procedure, for example, overrides an intrinsic procedure with the same name, in the absence of an applicable `INTRINSIC` statement, because its interface is explicit. If a module or internal procedure is accessible by the same name as a name specified on an `INTRINSIC` statement, it is an error in a program unit.

This chapter introduces all of the intrinsic procedures supported by the CF90 and MIPSpro 7 Fortran 90 compilers. Each procedure is described in detail on an online man page. For more information on any of the intrinsic procedures introduced in this chapter, enter `man` *intrinsic_name* at your system prompt.

5.1 Intrinsic procedure terms and concepts

Intrinsic procedures are predefined by the language. They conform to the principles and rules for procedures as described in the *Fortran Language Reference Manual, Volume 1*, publication SR–3902. Intrinsic procedures are invoked in the same way as other procedures and employ the same argument association mechanism. The intrinsic procedure interfaces, including the argument keywords (dummy argument names) and argument options, are all described partially in this section and are described further in the man pages.

The four classes of intrinsic procedures are as follows:

- Inquiry functions.

- Elemental functions.
- Transformational functions.
- Subroutines. One intrinsic subroutine, `MVBITS`, is elemental.

The CF90 and MIPSpro 7 Fortran 90 compilers also include the following types of intrinsic procedures:

- Intrinsic procedures to support IEEE floating-point arithmetic.
- Miscellaneous procedures. All procedures in this group are extensions to the Fortran 90 standard.

5.1.1 Inquiry functions

An inquiry function returns information about something. The results of these functions depend on the properties of the principal argument rather than the value of this argument. The argument can be undefined.

The standard inquiry functions are as follows. In the following list, the numeric model referred to is described in Section 5.3, page 251.

Name	Information returned
`ALLOCATED`	Array allocation status
`ASSOCIATED`	Pointer association status or comparison
`BIT_SIZE`	Number of bits in the bit manipulation model
`DIGITS`	Number of significant digits in the numeric model
`EPSILON`	Number that is almost negligible compared to one
`HUGE`	Largest number in the numeric model
`KIND`	Kind type parameter value
`LBOUND`	Lower dimension bounds of an array
`LEN`	Length of a character entity
`MAXEXPONENT`	Maximum exponent in the numeric model
`MINEXPONENT`	Minimum exponent in the numeric model
`PRECISION`	Decimal precision of the numeric model
`PRESENT`	Dummy argument presence
`RADIX`	Base of the numeric model
`RANGE`	Decimal exponent range of the numeric model

SHAPE	Shape of an array or scalar
SIZE	Total number of elements in an array
TINY	Smallest positive number in the numeric model
UBOUND	Upper dimension bounds of an array

5.1.2 Elemental functions

Many intrinsic functions and the `MVBITS` subroutine can be referenced elementally; that is, the intrinsic is called with one or more array arguments and yields an array result.

Some of the intrinsic functions included in the CF90 and MIPSpro 7 Fortran 90 compilers as extensions to the Fortran 90 standard are elemental functions. This section contains two lists. The first list summarizes the standard elemental functions; in this list, the numeric model referred to is described in Section 5.3, page 251. The second list summarizes the extensions that are elemental.

Name	Information returned or operation
ABS	Absolute value
ACHAR	Character in given position in ASCII collating sequence
ACOS	Arccosine
ADJUSTL	Adjust a string left
ADJUSTR	Adjust a string right
AIMAG	Imaginary part of a complex number
AINT	Truncation to whole number
ANINT	Nearest whole number
ASIN	Arcsine
ATAN	Arctangent
ATAN2	Arctangent of a nonzero complex value
BTEST	Bit testing
CEILING	Least integer greater than or equal to a given real number
CHAR	Character in given position in the collating sequence used by this compiler
CMPLX	Conversion to complex type

`CONJG`	Conjugate of a complex number
`COS`	Cosine
`COSH`	Hyperbolic cosine
`DBLE`	Conversion to double-precision real type
`DIM`	Positive difference
`DPROD`	Double-precision real product
`EXP`	Exponential
`EXPONENT`	Exponent part of a numeric model number
`FLOOR`	Greatest integer less than or equal to a given real number
`FRACTION`	Fraction part of a numeric model number
`IACHAR`	Position of a character in the ASCII collating sequence
`IAND`	Logical `AND`
`IBCLR`	Clear a bit
`IBITS`	Bit extraction
`IBSET`	Set a bit
`ICHAR`	Position of a character in the collating sequence used by the compiler
`IEOR`	Exclusive `OR`
`INDEX`	Starting position of a substring
`INT`	Conversion to integer type
`IOR`	Inclusive `OR`
`ISHFT`	Logical shift
`ISHFTC`	Circular shift
`LEN_TRIM`	Length without trailing blank characters
`LGE`	Lexically greater than or equal
`LGT`	Lexically greater than
`LLE`	Lexically less than or equal
`LLT`	Lexically less than
`LOG`	Natural logarithm
`LOG10`	Common logarithm (base 10)

`LOGICAL`	Convert between objects of type logical with different kind type parameters
`MAX`	Maximum value
`MERGE`	Merge under mask
`MIN`	Minimum value
`MOD`	Remainder function
`MODULO`	Modulo function
`NEAREST`	Nearest different machine-representable number in given direction
`NINT`	Nearest integer
`NOT`	Logical complement
`REAL`	Conversion to real type
`RRSPACING`	Reciprocal of the relative spacing of numeric model numbers near given number
`SCALE`	Multiply a real by its base to an integer power
`SCAN`	Scan a string for a character in a set
`SET_EXPONENT`	Set exponent part of a numeric model number
`SIGN`	Transfer of sign
`SIN`	Sine
`SINH`	Hyperbolic sine
`SPACING`	Absolute spacing of numeric model numbers near a given number
`SQRT`	Square root
`TAN`	Tangent
`TANH`	Hyperbolic tangent
`VERIFY`	Verify the set of characters in a string

The CF90 and MIPSpro 7 Fortran 90 compilers implement several miscellaneous intrinsic procedures that are extensions to the Fortran 90 standard and are elemental functions.

Note: Not all of the miscellaneous procedures are implemented on all platforms. The `GETPOS` intrinsic function is not implemented on IRIX systems, and the implementation of `FCD` is deferred on IRIX systems. See the man pages for each routine for more implementation information.

The following miscellaneous intrinsic procedures are supported:

Name	Information returned or operation
`COT`	Cotangent
`DSHIFTL`	Double-word left shift
`DSHIFTR`	Double-word right shift
`EQV`	Logical equivalence
`FCD`	Constructs a character pointer in Fortran Character Descriptor format
`GETPOS`	Obtains file position
`LEADZ`	Counts number of leading 0 bits
`LENGTH`	Returns the number of words successfully transferred
`POPCNT`	Counts number of bits set to 1
`POPPAR`	Computes bit population parity
`UNIT`	Performs I/O functions

5.1.3 Transformational functions

Transformational procedures have either a dummy argument that is array valued (for example, the `SUM` function) or an actual argument that is array valued without causing an elemental interpretation (for example, the `SHAPE` function). A transformational function transforms an array actual argument into a scalar result or another array, rather than applying the operation element-by-element.

The Fortran 90 standard transformational functions are as follows:

Name	Information returned or operation
`ALL`	True if all values are true
`ANY`	True if any value is true
`COUNT`	Number of true elements in an array
`CSHIFT`	Circular shift
`DOT_PRODUCT`	Dot product of two rank one arrays
`EOSHIFT`	End-off shift
`MATMUL`	Matrix multiplication

`MAXLOC`	Location of a maximum value in an array
`MAXVAL`	Maximum value in an array
`MINLOC`	Location of a minimum value in an array
`MINVAL`	Minimum value in an array
`PACK`	Pack an array into an array of rank one
`PRODUCT`	Product of array elements
`REPEAT`	Repeated concatenation
`RESHAPE`	Reshape an array
`SELECTED_INT_KIND`	Integer kind type parameter, given range
`SELECTED_REAL_KIND`	Real kind type parameter, given range
`SPREAD`	Replicates array by adding a dimension
`SUM`	Sum of array elements
`TRANSFER`	Treat first argument as if of type of second argument
`TRANSPOSE`	Transpose an array of rank two
`TRIM`	Remove trailing blank characters
`UNPACK`	Unpack an array of rank one into an array under a mask

5.1.4 Subroutines

An intrinsic subroutine is referenced by the `CALL` statement that uses its name explicitly. The name of an intrinsic subroutine cannot be used as an actual argument. `MVBITS` is an elemental subroutine.

Note: Not all of the intrinsic subroutines are implemented on all platforms. `FREE` is not implemented as an intrinsic subroutine on UNICOS and UNICOS/mk systems. See the man pages for each routine for more implementation information.

The intrinsic subroutines are as follows:

Name	Purpose
`DATE_AND_TIME`	Obtains date and time
`FREE`	Deallocates memory

`MVBITS`	Copies bits from one bit manipulation model integer to another
`RANDOM_NUMBER`	Returns pseudorandom number
`RANDOM_SEED`	Initializes or restarts the pseudorandom number generator
`SYSTEM_CLOCK`	Obtains data from the system clock

5.1.5 Intrinsic procedures to support IEEE floating-point arithmetic (EXTENSION)

The intrinsic procedures described in this section support the IEEE Standard for Binary Floating-point Arithmetic.

Note: The intrinsic procedures to support IEEE floating-point arithmetic are supported on all UNICOS/mk and IRIX platforms. They are supported on UNICOS platforms when running on CRAY T90 systems that support IEEE floating-point arithmetic. See the man pages for each routine for more implementation information.

The IEEE intrinsic procedures allow you to obtain, alter, and restore the floating-point status flags. For example, they allow a procedure to save the current floating-point state as a single word on entry to a procedure and to restore the floating-point state before returning to the caller.

The IEEE intrinsic procedures are as follows:

Name	Information returned or operation
`CLEAR_IEEE_EXCEPTION`	Clears floating-point exception indicator
`DISABLE_IEEE_INTERRUPT`	Disables floating-point interrupt
`ENABLE_IEEE_INTERRUPT`	Enables floating-point interrupt
`GET_IEEE_EXCEPTIONS`	Retrieves flags that represent the current floating-point exception status
`GET_IEEE_INTERRUPTS`	Retrieves flags that represent the current floating-point interrupt status

`GET_IEEE_ROUNDING_MODE`	Returns current floating-point rounding mode
`GET_IEEE_STATUS`	Retrieves flags that represent the current floating-point status
`IEEE_BINARY_SCALE`	Returns *y* multiplied by 2^n
`IEEE_CLASS`	Returns the class to which *x* belongs
`IEEE_COPY_SIGN`	Returns *x* with the sign of *y*
`IEEE_EXPONENT`	Returns the unbiased exponent of *x*
`IEEE_FINITE`	Tests for *x* being greater than negative infinity and less than positive infinity
`IEEE_IS_NAN`	Tests for *x* being a NaN
`IEEE_INT`	Converts *x* to an integral value according to the rounding mode in effect
`IEEE_NEXT_AFTER`	Returns the next representable neighbor of *x* in the direction toward *y*
`IEEE_REAL`	Converts *x* to a real value according to the rounding mode in effect
`IEEE_REMAINDER`	Returns the remainder of the operation x/y
`IEEE_UNORDERED`	Tests for *x* or *y* being a NaN
`INT_MULT_UPPER`	Multiplies unsigned integers and returns the uppermost bits
`SET_IEEE_EXCEPTION`	Sets floating-point exception indicator
`SET_IEEE_EXCEPTIONS`	Restores the floating-point status

`SET_IEEE_INTERRUPTS`	Restores floating-point interrupt status
`SET_IEEE_ROUNDING_MODE`	Restores floating-point rounding mode
`SET_IEEE_STATUS`	Restores floating-point status before
`TEST_IEEE_EXCEPTION`	Returns the state of a floating-point exception
`TEST_IEEE_INTERRUPT`	Returns the state of a floating-point interrupt

For more information on how Cray Research has implemented the IEEE Standard for Binary Floating-point Arithmetic, see *Migrating to the CRAY T90 Series IEEE Floating Point*, publication SN–2194.

5.1.5.1 Using named constants

A number of the procedures named in Section 5.1.5, page 244, use named constants. The named constants are provided in module `FTN_IEEE_DEFINITIONS` on UNICOS, UNICOS/mk, and IRIX systems and in module `CRI_IEEE_DEFINITIONS` on UNICOS and UNICOS/mk systems. To use this module, include the following statements in your program:

- On UNICOS, UNICOS/mk, and IRIX systems: `USE FTN_IEEE_DEFINITIONS`
- On UNICOS and UNICOS/mk systems: `USE CRI_IEEE_DEFINITIONS`

Note: The `CRI_IEEE_DEFINITIONS` and `FTN_IEEE_DEFINITIONS` modules must be used if you want to use named constants to pass values to IEEE intrinsic procedures or to interpret return values. The named constants should always be used as arguments to the appropriate procedures, as opposed to using hard-coded values, because the values represented by the named constants might not be the same on all architectures. Cray Research and Silicon Graphics reserve the right to change the values represented by the named constants.

Warning: The `USE CRI_IEEE_DEFINITIONS` module is obsolescent. It will be removed for the CF90 4.0 release.

The named constants are available to you if the scoping unit from which they are referenced (or a parent scoping unit) has used the module `FTN_IEEE_DEFINITIONS` or `CRI_IEEE_DEFINITIONS`. The following paragraphs describe the named constants that are available to you and the systems upon which they are supported.

The following named constants allow you to access floating-point exception flags:

- `IEEE_XPTN_CRI_INVALID_OPND` (supported only on CRAY T90 systems that support IEEE floating-point arithmetic)
- `IEEE_XPTN_INEXACT_RESULT`
- `IEEE_XPTN_UNDERFLOW`
- `IEEE_XPTN_OVERFLOW`
- `IEEE_XPTN_DIV_BY_ZERO`
- `IEEE_XPTN_INVALID_OPR`
- `IEEE_XPTN_ALL`

The following named constants allow you to access floating-point interrupt flags:

- `IEEE_NTPT_CRI_INVALID_OPND` (supported only on CRAY T90 systems that support IEEE floating-point arithmetic)
- `IEEE_NTPT_INEXACT_RESULT`
- `IEEE_NTPT_UNDERFLOW`
- `IEEE_NTPT_OVERFLOW`
- `IEEE_NTPT_DIV_BY_ZERO`
- `IEEE_NTPT_INVALID_OPR`
- `IEEE_NTPT_ALL`

The following named constants allow you to access the floating-point rounding mode:

- `IEEE_RM_NEAREST`
- `IEEE_RM_POS_INFINITY`
- `IEEE_RM_ZERO`
- `IEEE_RM_NEG_INFINITY`

The following named constants allow you to determine the class of a floating-point value:

- `IEEE_CLASS_SIGNALING_NAN`
- `IEEE_CLASS_QUIET_NAN`
- `IEEE_CLASS_NEG_INFINITY`
- `IEEE_CLASS_NEG_NORM_NONZERO`
- `IEEE_CLASS_NEG_DENORM`
- `IEEE_CLASS_NEG_ZERO`
- `IEEE_CLASS_POS_ZERO`
- `IEEE_CLASS_POS_DENORM`
- `IEEE_CLASS_POS_NORM_NONZERO`
- `IEEE_CLASS_POS_INFINITY`

5.1.5.2 Renaming intrinsic procedures

Some of the IEEE intrinsic procedure names are long. This was done to avoid creating names that might be identical to names that already exist in user programs. If, however, an IEEE intrinsic procedure name conflicts with an existing user identifier, you can use the Fortran 90 renaming capabilities to provide your own local name by following these steps:

1. Create a module that contains an `INTRINSIC` statement for each intrinsic procedure you wish to rename.
2. Reference this module name in a `USE` statement in a context in which you want to reference the intrinsic procedure. Use a rename clause on the `USE` statement for the intrinsic procedures you want to rename.

For example, to rename `IEEE_IS_NAN` to `IS_NAN`:

```
MODULE MY_NAMES
INTRINSIC IEEE_IS_NAN
END MODULE
PROGRAM MAIN
USE MY_NAMES, IS_NAN => IEEE_IS_NAN
...
! REFERENCES TO IS_NAN HERE
! ARE ACTUALLY TO IEEE_IS_NAN
....
END PROGRAM
```

5.1.6 Miscellaneous procedures (EXTENSION)

Most of the CF90 and MIPSpro 7 Fortran 90 compiler intrinsic functions that are extensions to the Fortran 90 standard are elemental intrinsic functions.

> **Note:** Not all of the miscellaneous procedures are implemented on all platforms. Only `LOC` and `MALLOC` are available on IRIX systems. `MALLOC` is not implemented as an intrinsic function on UNICOS and UNICOS/mk systems. See the man pages for each routine for more implementation information.

The following functions are extensions to the standard, but they are **not** elemental functions:

Name	Information returned or operation
`CLOC`	Obtains the address of a character entity.
`LOC`	Returns address of a variable
`M@CLR`	Clears BML bit
`M@LD`	Loads transposed bit matrix
`M@LDMX`	Loads bit matrix and multiplies
`M@MX`	Multiplies bit matrix
`M@UL`	Unloads bit matrix
`MALLOC`	Allocates memory on the heap
`MY_PE`	Returns the number of the processing element (PE) that is executing
`MEMORY_BARRIER`	Blocks memory loads and stores until processes finish
`REMOTE_WRITE_BARRIER`	Blocks processor until remote writes have finished

`WRITE_MEMORY_BARRIER`	Blocks processor until remote writes have finished

5.2 Argument keywords

Intrinsic procedure references can use keyword arguments, as described in Section 4.7.4, page 213. A number of Fortran 90 intrinsic procedure arguments are optional. Using keywords allows you to omit corresponding actual arguments. In the following example, the keyword form shown is used because the optional first argument `SIZE` is omitted:

```
CALL RANDOM_SEED(PUT=SEED_VALUE)
```

If you are using argument keywords, the position at which the first argument keyword appears determines whether others in the call must also appear in keyword form. If an argument keyword is used, any other required arguments to the right of that argument must also appear in keyword form.

The following examples show **correct** use of actual arguments:

```
CALL DOT_PRODUCT(VECTOR_A=ARRAY1, VECTOR_B=ARRAY2)
CALL DOT_PRODUCT(ARRAY1, VECTOR_B=ARRAY2)
CALL DOT_PRODUCT(ARRAY1, ARRAY2)
```

The following examples show **incorrect** use of actual arguments:

```
CALL DOT_PRODUCT(VECTOR_A=ARRAY1, ARRAY2)
CALL DOT_PRODUCT(ARRAY2)
```

Intrinsic procedure keywords (dummy argument names) have been made as consistent as possible, including using the same name in different intrinsic procedures for dummy arguments that play a corresponding or identical role. These include `DIM`, `MASK`, `KIND`, and `BACK`.

- `DIM` is used, mostly in the array reduction functions and in some of the other array functions, to specify the array dimension involved. `DIM` must be a scalar integer value and usually is optional.
- `MASK` is used, mostly, in the array functions, to mask out elements of an array that are not to be involved in the operation. For example, in the function `SUM`, any element of the array that is not to be included in the sum of the elements can be excluded by using an appropriate mask. The `MASK` must be a logical array with the same shape as the array it is masking; it usually is an optional argument.

- `KIND` is an argument that is used mainly in the transfer and conversion functions to specify the kind type parameter of the function result. The `KIND` actual argument must be a scalar integer initialization expression, even in elemental references; it usually is optional.
- `BACK` is an optional logical argument used in several of the intrinsic functions to specify reverse order (backward) processing. For example, if `BACK=.TRUE.` in the `INDEX` function, then the search is performed beginning from the right end of the string rather than the left end.

5.3 Representation models

Some intrinsic functions compute values in relationship to how data is represented. These values are based upon and determined by the underlying representation model. There are three such models in Fortran 90:

- The bit model
- The integer number system model
- The real number system model

These models, and the corresponding functions that return values related to the models, allow development of robust and portable code. For example, by obtaining information about the spacing of real numbers, you can control the convergence of a numerical algorithm so that the code achieves maximum accuracy while attaining convergence.

The models themselves apply to the CF90 and MIPSpro 7 Fortran 90 compilers running on any platform. The values that go into these models, however, are platform-specific. Please see the `models`(3I) man page information on how these models apply to the various platforms that the CF90 and MIPSpro 7 Fortran 90 compilers support.

5.4 Intrinsic procedure summary

This section summarizes the syntax of each intrinsic procedure. These tables are meant to supplement, not replace, the intrinsic procedure man pages. For more information on any of the intrinsics listed in these tables, enter `man` *intrinsic_name* at your system prompt.

Some intrinsic functions have both generic and specific names. The intrinsic functions that have specific names can be called with those names as well as with the generic names. To pass an intrinsic procedure as an actual argument, however, you must use the specific name. The following table shows the specific intrinsic procedure names available in the CF90 and MIPSpro 7 Fortran 90 compilers. Some of the specific intrinsic names are the same as the generic names.

The argument names shown are the keywords for keyword argument calls. All of the optional arguments are enclosed in brackets [ ].

The following table shows the intrinsic procedures that the CF90 and MIPSpro 7 Fortran 90 compilers support both as part of the Fortran 90 standard and as extensions to the standard. If the procedure has both a generic and a specific name, the generic name is shown first, and the specific names are listed underneath.

> **Note:** Table 15, page 253, contains entries for the `I24MULT`(3I) and `LINT`(3I) intrinsic procedures. These intrinsics are outmoded. They will be removed in the CF90 4.0 release.

Table 15. CF90 and MIPSpro 7 Fortran 90 intrinsic procedures

Procedure name	Arguments
ABS	([A=]*a*)
IABS	([A=]*a*)
DABS	([A=]*a*)
CABS	([A=]*a*)
CDABS	([A=]*a*)
ACHAR	([I=]*i*)
ACOS	([X=]*x*)
DACOS	([X=]*x*)
ADJUSTL	([STRING=]*string*)
ADJUSTR	([STRING=]*string*)
AIMAG	([Z=]*z*)
AINT	([A=]*a* [,[KIND=]*kind*])
DINT	([A=]*a* [,[KIND=]*kind*])
ALL	([MASK=]*mask* [,[DIM=]*dim*])
ALLOCATED	([ARRAY=]*array*)
AND	([I=]*i*, [J=]*j*)
ANINT	([A=]*a* [,[KIND=]*kind*])
DNINT	([A=]*a* [,[KIND=]*kind*])
ANY	([MASK=]*mask* [,[DIM=]*dim*])
ASIN	([X=]*x*)
DASIN	([X=]*x*)
ASSOCIATED	([POINTER=]*pointer* [,[TARGET=]*target*])
ATAN	([X=]*x*)
DATAN	([X=]*x*)
ATAN2	([Y=]*y*, [X=]*x*)
DATAN2	([Y=]*y*, [X=]*x*)
BIT_SIZE	([I=]*i*)
BTEST	([I=]*i*, [POS=]*pos*)
CEILING	([A=]*a*)

Procedure name	Arguments
CHAR	([I=]*i* [,[KIND=]*kind*])
CLEAR_IEEE_EXCEPTION	([EXCEPTION=]*exception*)
CLOC	([C=]*c*)
CLOCK	()
CMPLX	([X=]*x* [,[Y=]*y*] [,[KIND=]*kind*])
COMPL	([I=]*i*)
CONJG	([Z=]*z*)
COS	([X=]*x*)
DCOS	([X=]*x*)
CCOS	([X=]*x*)
CDCOS	([X=]*x*)
COSH	([X=]*x*)
DCOSH	([X=]*x*)
COT	([X=]*x*)
DCOT	([X=]*x*)
COUNT	([MASK=]*mask* [,[DIM=]*dim*])
CSHIFT	([ARRAY=]*array*, [SHIFT=]*shift* [,[DIM=]*dim*])
CSMG	([I=]*i*, [J=]*j*, [K=]*k*)
CVMGM	([I=]*i*, [J=]*j*, [K=]*k*)
CVMGN	([I=]*i*, [J=]*j*, [K=]*k*)
CVMGP	([I=]*i*, [J=]*j*, [K=]*k*)
CVMGT	([I=]*i*, [J=]*j*, [K=]*k*)
CVMGZ	([I=]*i*, [J=]*j*, [K=]*k*)
DATE	()
JDATE	()
DATE_AND_TIME	([[DATE=]*date*] [,[TIME=]*time*] [,[ZONE=]*zone*] [,[VALUES=]*values*])
DBLE	([A=]*a*)
DFLOAT	([A=]*a*)
DIGITS	([X=]*x*)

Procedure name	Arguments
DIM	([X=]*x*, [Y=]*y*)
IDIM	([X=]*x*, [Y=]*y*)
DDIM	([X=]*x*, [Y=]*y*)
DISABLE_IEEE_INTERRUPT	([INTERRUPT=]*interrupt*)
DOT_PRODUCT	([VECTOR_A=]*vector_a*, [VECTOR_B=]*vector_b*)
DPROD	([X=]*x*, [Y=]*y*)
DSHIFTL	([I=]*i*, [J=]*j*, [K=]*k*)
DSHIFTR	([I=]*i*, [J=]*j*, [K=]*k*)
ENABLE_IEEE_INTERRUPT	([INTERRUPT=]*interrupt*)
EOSHIFT	([ARRAY=]*array*, [SHIFT=]*shift* [,[BOUNDARY=]*bound*] [,[DIM=]*dim*])
EPSILON	([X=]*x*)
EQV	([I=]*i*, [J=]*j*)
EXP	([X=]*x*)
DEXP	([X=]*x*)
CEXP	([X=]*x*)
CDEXP	([X=]*x*)
EXPONENT	([X=]*x*)
FCD	([I=]*i*, [J=]*j*)
FLOOR	([A=]*a*)
FRACTION	([X=]*x*)
FREE	([P=]*iptr*)
GET_IEEE_EXCEPTIONS	([STATUS=]*status*)
GET_IEEE_INTERRUPTS	([STATUS=]*status*)
GET_IEEE_ROUNDING_MODE	([ROUNDING_MODE=]*rounding_mode*)
GET_IEEE_STATUS	([STATUS=]*status*)
GETPOS	([I=]*i*)
HUGE	([X=]*x*)
I24MULT	([I=]*i*, [J=]*j*)
IACHAR	([C=]*c*)
IAND	([I=]*i*, [J=]*j*)

Procedure name	Arguments
IBCLR	([I=]*i*, [POS=]*pos*)
IBITS	([I=]*i*, [POS=]*pos*, [LEN=]*len*)
IBSET	([I=]*i*, [POS=]*pos*)
ICHAR	([C=]*c*)
IEEE_BINARY_SCALE	([Y=]*y*, [N=]*n*)
IEEE_CLASS	([X=]*x*)
IEEE_COPY_SIGN	([X=]*x*, [Y=]*y*)
IEEE_EXPONENT	([X=]*x* [,[Y=]*y*])
IEEE_FINITE	([X=]*x*)
IEEE_INT	([X=]*x* [,[Y=]*y*])
IEEE_IS_NAN	([X=]*x*)
IEEE_NEXT_AFTER	([X=]*x*, [Y=]*y*)
IEEE_REAL	([X=]*x* [,[Y=]*y*])
IEEE_REMAINDER	([X=]*x*, [Y=]*y*)
IEEE_UNORDERED	([X=]*x*, [Y=]*y*)
IEOR	([I=]*i*, [J=]*j*)
INDEX	([STRING=]*string*, [SUBSTRING=]*substring* [,[BACK=]*back*])
INT_MULT_UPPER	([I=]*i*, [J=]*j*)
INT	([A=]*a* [,[KIND=]*kind*])
IFIX	([A=]*a* [,[KIND=]*kind*])
IDINT	([A=]*a* [,[KIND=]*kind*])
INT24	([I=]*i*)
LINT	([I=]*i*)
IOR	([I=]*i*, [J=]*j*)
IRTC	()
ISHFT	([I=]*i*, [SHIFT=]*shift*)
ISHFTC	([I=]*i*, [SHIFT=]*shift* [,[SIZE=]*size*])
KIND	([X=]*x*)
LBOUND	([ARRAY=]*array* [,[DIM=]*dim*])

Procedure name	Arguments
LEADZ	([I=]*i*)
LEN	([STRING=]*string*)
LEN_TRIM	([STRING=]*string*)
LENGTH	([I=]*i*)
LGE	([STRING_A=]*string_a*, [STRING_B=]*string_b*)
LGT	([STRING_A=]*string_a*, [STRING_B=]*string_b*)
LLE	([STRING_A=]*string_a*, [STRING_B=]*string_b*)
LLT	([STRING_A=]*string_a*, [STRING_B=]*string_b*)
LOC	([I=]*i*)
LOG	([X=]*x*)
ALOG	([X=]*x*)
DLOG	([X=]*x*)
CLOG	([X=]*x*)
CDLOG	([X=]*x*)
LOG10	([X=]*x*)
ALOG10	([X=]*x*)
DLOG10	([X=]*x*)
LOGICAL	([L=]*l* [,[KIND=]*kind*])
M@CLR	()
M@LD	([X=]*x*)
M@LDMX	([X=]*x*, [Y=]*y*)
M@MX	([X=]*x* [,[Y=]*y*])
M@UL	()
MALLOC	([N=]*nbytes*)
MASK	([I=]*i*)
MATMUL	([MATRIX_A=]*matrix_a*, [MATRIX_B=]*matrix_b*)
MAX	([A1=]*a1*, [A2=]*a2* [,[A3=]*a3*]...)
MAX0	([A1=]*a1*, [A2=]*a2* [,[A3=]*a3*]...)
AMAX1	([A1=]*a1*, [A2=]*a2* [,[A3=]*a3*]...)
DMAX1	([A1=]*a1*, [A2=]*a2* [,[A3=]*a3*]...)
MAX1	([A1=]*a1*, [A2=]*a2* [,[A3=]*a3*]...)
AMAX0	([A1=]*a1*, [A2=]*a2* [,[A3=]*a3*]...)

Procedure name	Arguments
MAXEXPONENT	([X=]*x*)
MAXLOC	([ARRAY=]*array* [,[DIM=]*dim*] [,[MASK=]*mask*]) ([ARRAY=]*array* [,[MASK=]*mask*])
MAXVAL	([ARRAY=]*array* [,[DIM=]*dim*] [,[MASK=]*mask*]) ([ARRAY=]*array* [,[MASK=]*mask*])
MEMORY_BARRIER	()
MERGE	([TSOURCE=]*tsource*, [FSOURCE=]*fsource*, [MASK=]*mask*)
MIN	([A1=]*a1*, [A2=]*a2* [,[A3=]*a3*]...)
MIN0	([A1=]*a1*, [A2=]*a2* [,[A3=]*a3*]...)
AMIN1	([A1=]*a1*, [A2=]*a2* [,[A3=]*a3*]...)
DMIN1	([A1=]*a1*, [A2=]*a2* [,[A3=]*a3*]...)
MIN1	([A1=]*a1*, [A2=]*a2* [,[A3=]*a3*]...)
AMIN0	([A1=]*a1*, [A2=]*a2* [,[A3=]*a3*]...)
MINEXPONENT	([X=]*x*)
MINLOC	([ARRAY=]*array* [,[DIM=]*dim*] [,[MASK=]*mask*]) ([ARRAY=]*array* [,[MASK=]*mask*])
MINVAL	([ARRAY=]*array* [,[DIM=]*dim*] [,[MASK=]*mask*]) ([ARRAY=]*array* [,[MASK=]*mask*])
MOD	([A=]*a*, [P=]*p*)
AMOD	([A=]*a*, [P=]*p*)
DMOD	([A=]*a*, [P=]*p*)
MODULO	([A=]*a*, [P=]*p*)
MVBITS	([FROM=]*from*, [FROMPOS=]*frompos*, [LEN=]*len*, [TO=]*to*, [TOPOS=]*topos*)
MY_PE	()
NEAREST	([X=]*x*, [S=]*s*)
NEQV	([I=]*i*, [J=]*j*)
XOR	([I=]*i*, [J=]*j*)
NINT	([A=]*a* [,[KIND=]*kind*])
IDNINT	([A=]*a* [,[KIND=]*kind*])
NOT	([I=]*i*)
NUMARG	()
OR	([I=]*i*, [J=]*j*)

Procedure name	Arguments
PACK	([ARRAY=]*array*, [MASK=]*mask* [,[VECTOR=]*vector*])
POPCNT	([I=]*i*)
POPPAR	([I=]*i*)
PRECISION	([X=]*x*)
PRESENT	([A=]*a*)
PRODUCT	([ARRAY=]*array* [,[DIM=]*dim*] [,[MASK=]*mask*]) ([ARRAY=]*array* [,[MASK=]*mask*])
RADIX	([X=]*x*)
RANDOM_NUMBER	([HARVEST=]*harvest*)
RANDOM_SEED	([[SIZE=]*size*] [,[PUT=]*put*] [,[GET=]*get*])
RANF	()
RANGET	([I=]*i*)
RANSET	([I=]*i*)
RANGE	([X=]*x*)
REAL	([A=]*a* [,[KIND=]*kind*])
FLOAT	([A=]*a* [,[KIND=]*kind*])
SNGL	([A=]*a* [,[KIND=]*kind*])
DREAL	([A=]*a* [,[KIND=]*kind*])
REMOTE_WRITE_BARRIER	()
REPEAT	([STRING=]*string*, [NCOPIES=]*ncopies*)
RESHAPE	([SOURCE=]*source*, [SHAPE=]*shape* [,[PAD=]*pad*] [,[ORDER=]*order*])
RRSPACING	([X=]*x*)
RTC	()
SCALE	([X=]*x*, [I=]*i*)
SCAN	([STRING=]*string*, [SET=]*set* [,[BACK=]*back*])
SELECTED_INT_KIND	([R=]*r*)
SELECTED_REAL_KIND	([[P=]*p*] [,[R=]*r*])
SET_EXPONENT	([X=]*x*, [I=]*i*)
SET_IEEE_EXCEPTION	([EXCEPTION=]*exception*)
SET_IEEE_EXCEPTIONS	([STATUS=]*status*)

Procedure name	Arguments
SET_IEEE_INTERRUPTS	([STATUS=]*status*)
SET_IEEE_ROUNDING_MODE	([ROUNDING_MODE=]*rounding_mode*)
SET_IEEE_STATUS	([STATUS=]*status*)
SHAPE	([STATUS=]*status*)
SHIFT	([I=]*i*, [J=]*j*)
SHIFTA	([I=]*i*, [J=]*j*)
SHIFTL	([I=]*i*, [J=]*j*)
SHIFTR	([I=]*i*, [J=]*j*)
SIGN	([A=]*a*, [B=]*b*)
ISIGN	([A=]*a*, [B=]*b*)
DSIGN	([A=]*a*, [B=]*b*)
SIN	([X=]*x*)
DSIN	([X=]*x*)
CSIN	([X=]*x*)
CDSIN	([X=]*x*)
SINH	([X=]*x*)
DSINH	([X=]*x*)
SIZE	([ARRAY=]*array* [,[DIM=]*dim*])
SPACING	([X=]*x*)
SPREAD	([SOURCE=]*source*, [DIM=]*dim*, [NCOPIES=]*ncopies*)
SQRT	([X=]*x*)
DSQRT	([X=]*x*)
CSQRT	([X=]*x*)
CDSQRT	([X=]*x*)
SUM	([ARRAY=]*array* [,[DIM=]*dim*] [,[MASK=]*mask*])
	([ARRAY=]*array* [,[MASK=]*mask*])
SYSTEM_CLOCK	([[COUNT=]*count*] [,[COUNT_RATE=]*count_rate*] [,[COUNT_MAX=]*count_max*])
TAN	([X=]*x*)
DTAN	([X=]*x*)
TANH	([X=]*x*)
DTANH	([X=]*x*)
TEST_IEEE_EXCEPTION	([EXCEPTION=]*exception*)

Procedure name	Arguments
`TEST_IEEE_INTERRUPT`	`([INTERRUPT=]`*interrupt*`)`
`TINY`	`([X=]`*x*`)`
`TRANSFER`	`([SOURCE=]`*source*`, [MOLD=]`*mold* `[,[SIZE=]`*size*`])`
`TRANSPOSE`	`([MATRIX=]`*matrix*`)`
`TRIM`	`([STRING=]`*string*`)`
`UBOUND`	`([ARRAY=]`*array* `[,[DIM=]`*dim*`])`
`UNIT`	`([I=]`*i*`)`
`UNPACK`	`([VECTOR=]`*vector*`, [MASK=]`*mask*`, [FIELD=]`*field*`)`
`VERIFY`	`([STRING=]`*string*`, [SET=]`*set* `[,[BACK=]`*back*`])`
`WRITE_MEMORY_BARRIER`	`()`

5.5 Man pages

The *Intrinsic Procedures Reference Manual*, publication SR–2138, contains copies of the man pages that describe the CF90 and MIPSpro 7 Fortran 90 intrinsic procedures. That manual also contains copies of the UNICOS and UNICOS/mk math library routine man pages. On IRIX systems, the underlying math library differs from that on UNICOS and UNICOS/mk systems, so please consult IRIX system math library documentation when running on an IRIX system.

These man pages can be accessed online through the `man`(1) command.

Note: Many intrinsic procedures have both a vector and a scalar version. If a vector version of an intrinsic procedure exists, and the intrinsic is called within a vectorizable loop, the compiler uses the vector version of the intrinsic. For information on which intrinsic procedures vectorize, see `intro_intrin`(3I).

Scope, Association, and Definition [6]

Scope, association, and definition (value assignment) provide the communication pathways between the different parts of the program.

With the new data types, program units, and procedure provisions, Fortran 90 builds significantly on the concepts of scope, association, and definition, making these concepts even more comprehensive and central to the working of Fortran. Fully understanding them is the key to understanding Fortran 90.

Various aspects of scope, association, and definition have already been mentioned. The material was not presented completely in those sections because a comprehensive and thorough understanding of the entire language is needed to fully assimilate these topics. Fortunately, all of the detailed rules and conditions presented in this chapter usually are not necessary in order to construct pieces of Fortran programs. If simple programming disciplines are followed, many of the subtle issues and concerns related to scope, association, and definition can be avoided in writing correct programs. However, there are some situations in which it is necessary to know all the details, particularly when modifying or maintaining programs, or looking for subtle bugs.

The concept of scope is introduced in the *Fortran Language Reference Manual, Volume 1*, publication SR–3902. Scope specifies that part of a program in which a particular entity is known and accessible. The spectrum of scope varies from an entire program (global) to individual program units (local) to statements or parts of statements.

To both communicate data between program units and limit and control accessibility of data, the language defines the concept of association, which relates local objects within and between program units. The association methods were introduced in previous sections and include association through arguments (argument association), association through storage (storage association), association through modules (use association), association through hosts (host association), and association through name aliases (pointer association).

After objects are associated, there can be multiple ways to define values for them. For example, assignment statements and input statements define objects directly by name, which can also cause associated items to become defined. In some cases, such associated object values can be unpredictable or unreliable. For example, if a real object is equivalenced (storage associated) with an integer object, defining the real object with a valid real value causes the integer object to acquire a meaningless value. This meaningless value is called an *undefined value.* In certain cases, such as for the integer object in the preceding example, certain values are considered to be undefined; references to undefined values do not conform to the standard.

Thus, the three topics — scope, association, and definition — are related. Scope specifies the part of a program where an entity is known and accessible. Association is the pathway along which entities in the same or different scopes communicate. Definition, and its opposite, undefinition, characterize the ways in which variables are defined and become undefined indirectly as a consequence of being associated with other objects.

6.1 Scope

Named entities such as variables, constants, procedures, block data subprograms, modules, and namelist groups, have scope. Other (unnamed) entities that have scope are operator symbols, the assignment symbol, labels, and input/output (I/O) unit numbers.

The *scope* of an entity is that part of a Fortran program in which that entity has a given meaning and can be used, defined, or referenced by its designator. The scope of an entity might be as large as the whole executable program or as small as part of a Fortran statement. Entities that can be used with the same meaning throughout the executable program are said to be *global entities* and have a global scope. An example of a global entity is an external procedure name.

Entities that can be used only within the smaller context of a subprogram are said to be *local entities* and have a local scope. An example of a local entity is a statement label. An even smaller context for scope might be a Fortran statement (or part of one); entities valid for only this context are said to be *statement entities*, such as the dummy arguments in a statement function.

A scoping unit can contain other scoping units. A scoping unit surrounded by another scoping unit may or may not inherit properties from the surrounding scope. For example, internal procedures, module procedures, and derived-type definitions inherit implicit typing rules from the surrounding scope. An interface body, on the other hand, is an example of a scope that is nonoverlapping, as opposed to nested. An interface body in a subprogram causes a hole in the subprogram scoping unit that is filled with the scoping unit of the interface body. This means that an interface body, for example, does not inherit implicit typing rules from the surrounding scope.

A scoping unit in Fortran 90 is one of the following:

- A derived-type definition

- A procedure interface body, excluding any derived-type definitions and procedure interface bodies contained within it
- A program unit or subprogram, excluding derived-type definitions, procedure interface bodies, and subprograms contained within it

To visualize the concept of scope and scoping units, consider Figure 17. The outer rectangle defines the boundary of the pieces of an executable Fortran program; it is not a scoping unit, but it could be said to represent global scope. Within the executable program four other rectangles depict program units. One is the main program, two others are external subprogram units, and the fourth one is a module program unit.

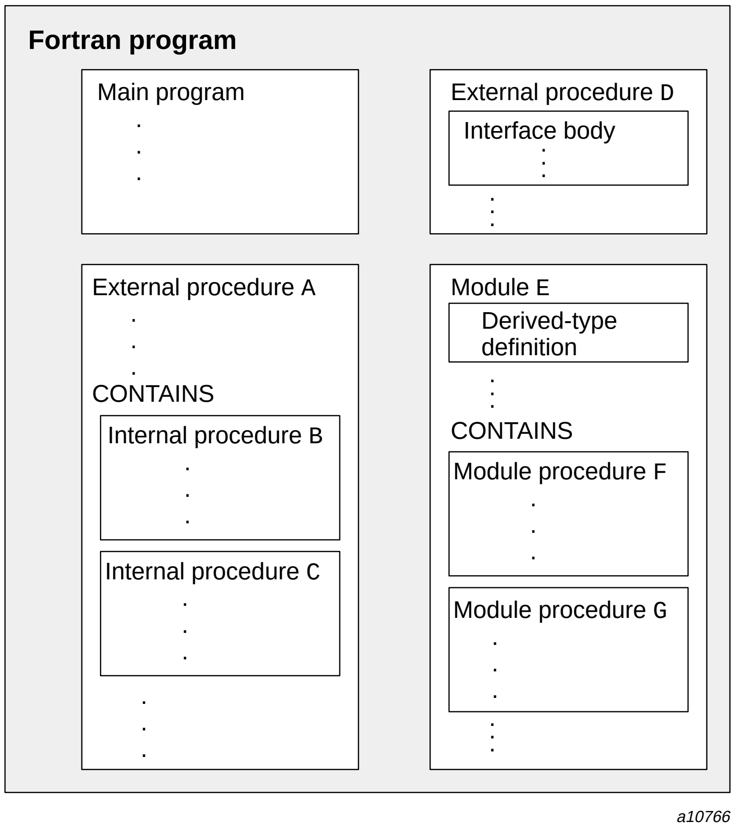


Figure 17. Scoping units

All four of these program unit rectangles represent scoping units, excluding any rectangles within them. The main program in this example encloses no rectangle and so is an integral scoping unit without holes. External subprogram A has two internal procedures within it, and therefore procedure A’s scoping unit is this rectangle, excluding internal procedures B and C. External subprogram D has an interface body in it and no internal procedures. Its scoping unit is procedure D, excluding the interface bodies in the interface block. Module E has a derived-type definition and two module procedures within it. Its scoping unit is similarly the module program unit, excluding the derived-type definition and the module procedures.

In addition, the interface bodies within the interface block, the derived-type definition, and each of the internal and module procedures are scoping units. In this example, these latter scoping units have no holes, as they do not themselves contain internal procedures, module procedures, interface bodies, or derived-type definitions, although they could in general.

6.1.1 Scope of names

A name has one of the following scopes:

- A *global* scope, which is the scope of an executable program (for example, an external function name)
- A *local* scope, which is the scope of a scoping unit (for example, an array name in a subroutine subprogram)
- A *statement* scope, which is the scope of a Fortran statement (for example, a statement function argument) or a part of a Fortran statement (for example, an implied-DO variable in a DATA statement)

6.1.1.1 Names as global entities

The name of a main program, an external procedure, a module, a block data program unit, or a common block has global scope. No two program unit names can have the same name; that is, a module cannot have the same name as an external subroutine. The CF90 compiler, however, allows a common block to have the same name as a program unit.

ANSI/ISO: The Fortran 90 standard does not allow two global entities to have the same name.

6.1.1.2 Names as local entities

There are three classes of names that have local scope:

- Names of variables, constants, control constructs, statement functions, internal procedures, module procedures, dummy procedures, intrinsic procedures, user-defined generic procedures, derived types, and namelist groups.
- Names of the components of a derived type. There is a separate class of names for each derived type, which means that two different derived types can have the same component names.
- Names of argument keywords. There is a separate class of names for each procedure with an explicit interface, which means that two different procedures can have the same argument keyword names.

A global entity name cannot be used to identify a local entity, except that a local entity can have the same name as a common block.

A nongeneric local name is unique within a scoping unit and within a name class; that is, it identifies exactly one entity. That name can also be used to identify a different object in a different scoping unit or a different object in a different class in the same scoping unit. When that name or a different name is used in other scoping units, it usually represents a different entity but may represent the same entity because of association.

A generic local name can be used for two or more different procedures in that scoping unit.

A local name can be used for another local entity in a different class in that scoping unit. For example, a structure component of type logical can have the same name as a local integer variable.

Components of a derived type have the scope of the derived-type definition, and when used in a qualified structure reference have the scope of the structure reference itself.

Argument keywords are local entities, and are in a separate class for each procedure with an explicit interface. This means that an argument keyword used for one procedure can be used as an argument keyword for another procedure, as a local variable or procedure name, and as a component name of a derived type.

If a common block name is the same as the name of a local entity, the name is the local entity except where it appears to identify the common block.

Uniqueness of the reference is determined by the context of the name. For example, a name enclosed in slashes in a SAVE statement is a common block name rather than the name of a local variable of the same name.

The name of an intrinsic procedure can be used as a local entity provided the intrinsic procedure itself is not used in that scoping unit. For example, if the scoping unit uses the name SIN as a local variable name, the intrinsic function SIN may not be used in the same program unit.

For each function and function entry that does not have a result variable, there is a local variable within the function subprogram scoping unit whose name is the same as the function or entry name. This local variable is used to define the value of the function or entry name within the function subprogram.

For each internal or module procedure, the name of the procedure is also a name local to the host scoping unit. Similarly, for any entry name used in a module procedure, the name of the entry is also a name local to the host scoping unit.

6.1.1.3 Names as statement entities

The name of a dummy argument in a statement function statement, or a DO variable in an implied-DO list of a DATA statement or array constructor, has a scope that is the statement or part of the statement. Such a name can be used elsewhere as a variable name or common block name without a name conflict and refers to a different entity when so used.

ANSI/ISO: The Fortran 90 standard states that the variable whose name is duplicated by the statement entity must be a scalar variable.

The name of a dummy argument used in a statement function statement has the scope of the statement. The type and type parameters of the name are determined by the declarations in the containing scoping unit.

The DO variable in an array constructor or an implied-DO loop in a DATA statement has the scope of that part of the statement in which it appears. The type, which must be integer, and type parameters of the name are determined by the declarations in the containing scoping unit.

The scope of the DO variable in these implied-DO lists is part of a statement, whereas the scope of the DO variable in a DO construct is local to the scoping unit containing the DO construct. Similarly, the DO variable of an implied-DO in an I/O item list has the scope of the program unit containing the I/O statement.

6.1.2 Scope of labels

A label is a local entity. No two statements in the same scoping unit can have the same label, but the same label can be used in different scoping units.

6.1.3 Scope of input/output (I/O) units

External I/O unit numbers are global entities and have global scope. Within an executable program, a unit number refers to the same I/O unit wherever it appears in a unit number context in an I/O statement.

6.1.4 Scope of operators

Intrinsic operators have global scope; the scope of a defined operator is local to a scoping unit. An operator symbol can refer to both an intrinsic operator and a defined operator. For example, the operator symbol (+) can be both global (its intrinsic meaning) and local (its defined meaning). Operators can be generic; that is, two or more operators can be designated by the same operator symbol. The types, kind parameter values, and the ranks of the operands distinguish which operator is used.

6.1.5 Scope of assignment

Intrinsic assignment has global scope; defined assignment has the scope of a scoping unit. The assignment symbol (=) always has global meanings and may, like operator symbols, also have local meanings. Assignment is generic; many assignment operations are designated by the same operator symbol. The types, kind parameter values, and the ranks of the entities on the left and right sides of the equal sign distinguish which assignment is used.

6.1.6 Unambiguous procedure references

A procedure reference is unambiguous if the procedure name in the reference is a specific procedure name that is not the same as any generic procedure name in that scoping unit. This is the case in references to the following items:

- Internal procedures
- Module and external procedures not appearing in an interface block with a generic specification in that scoping unit or available through use or host association

- Nongeneric specific names of intrinsic functions
- Statement functions
- Dummy procedures

Specific names of external and intrinsic procedures are global; all other specific procedure names are local.

Procedure references involving generic procedure names are also unambiguous under certain circumstances. Considering the dummy argument lists of any two procedures sharing the same generic name, one of these lists must have a nonoptional dummy argument that satisfies both of the following conditions:

- Either it is in a position in the list at which the other list has no dummy argument, or it has type, kind type parameter, and rank (TKR) properties different from that of the dummy argument in the same position in the other list.
- It has either a name different from all the dummy argument names in the other list, or it has TKR properties different from that of the dummy argument with the same name in the other list.

These rules apply regardless of whether the generic names are intrinsic, defined by interface blocks with generic specifications, or both. They also apply to generic operator and assignment symbols. Generic names of intrinsic functions are global, and defined generic names are local.

6.1.7 Resolving procedure references

A procedure reference is involved in the following situations:

1. Executing a `CALL` statement
2. Executing a defined assignment statement
3. Evaluating a defined operation
4. Evaluating an expression containing a function reference

In case 2, a generic name (the assignment symbol, `=`) is involved, and there must be an interface block with an `ASSIGNMENT` generic specification in the scoping unit or available through use or host association that identifies a specific external or module subroutine that defines this assignment. Section 4.7.7, page 217, Section 4.8.5, page 233, and Section 6.1.6, page 269, contain information that can help determine the specific subroutines involved in the reference.

In case 3, a generic name (the operator symbol) is involved, and there must be an interface block with an `OPERATOR` generic specification in the scoping unit or available through use or host association that identifies a specific external or module function that defines this operation. Section 4.7.7, page 217, Section 4.8.4, page 231, and Section 6.1.6, page 269, contain information that can help determine the specific function involved in the reference.

In cases 1 and 4, the following sequence of rules can be used to resolve the reference (that is, determine which specific procedure is involved in the reference). The first of these rules that applies, taken in order, resolves the reference.

1. If the procedure name in the reference is a dummy argument in that scoping unit, then the dummy argument is a dummy procedure and the reference is to that dummy procedure. Thus, the procedure invoked by the reference is the procedure supplied as the associated actual argument.

2. If the procedure name appears in an `EXTERNAL` statement in that scoping unit, the reference is to an external procedure with that name.

3. If the procedure name is that of an accessible internal procedure or statement function, the reference is to that internal procedure or statement function.

4. If the procedure name is specified as a generic name in an interface block in that scoping unit or in an interface block made accessible by use or host association, and the reference is consistent with one of the specific interfaces for that generic name, the reference is to that specific procedure. Section 4.7, page 200, contains information that can help determine the specific procedure invoked (there will be at most one such procedure).

5. If the procedure name appears in an `INTRINSIC` statement in that scoping unit, the reference is to the corresponding specific intrinsic procedure.

6. If the procedure name is accessible through use association, the reference is to that specific procedure. Note that it is possible, because of the renaming facility, for the procedure name in the reference to be different from that in the module.

7. If the scoping unit of the reference has a host scoping unit, and if application in the host of the preceding six rules resolves the reference, then the reference is so resolved.

8. The reference is to the corresponding specific intrinsic procedure if the following are true:

- The procedure name is either the specific or generic name of an intrinsic procedure.
- The actual arguments match the characteristics of a particular intrinsic procedure.

9. If the procedure name is not a generic name, the reference is to an external procedure with that name.
10. Otherwise the reference cannot be resolved and is not standard conforming.

6.2 Association

Fortran uses the concept of scoping so that the same name can be used for different things in different parts of a program. This is desirable so that programmers do not have to worry about conflicting uses of a name.

However, there are times when just the opposite is desired: the programmer wants different names in the same or different parts of a program to refer to the same entity. For example, there may be a need to have one data value that can be examined and modified by all of the procedures of a program. In general, particularly with external program units, the names used will be different in the different parts of the program, but they can be the same.

The *association* mechanism is used to indicate that local names in different scoping units or different local names in the same scoping unit refer to the same entity. The forms of association are as follows:

- Name association, which involves the use of names, always in different scoping units, to establish an association.
- Pointer association, which allows dynamic association of names within a scoping unit and is essentially an aliasing mechanism.
- Storage association, which involves the use of storage sequences to establish an association between data objects. The association can be between two objects in the same scoping unit (`EQUIVALENCE`) or in different scoping units (`COMMON`).
- Sequence association, which is a combination of name association and storage association. It applies to the association of actual arguments and array, character, and sequence structure dummy arguments using storage sequence association. It associates names in different scoping units.

Figure 18 and Figure 19, page 274, illustrate the various kinds of association in an executable program.

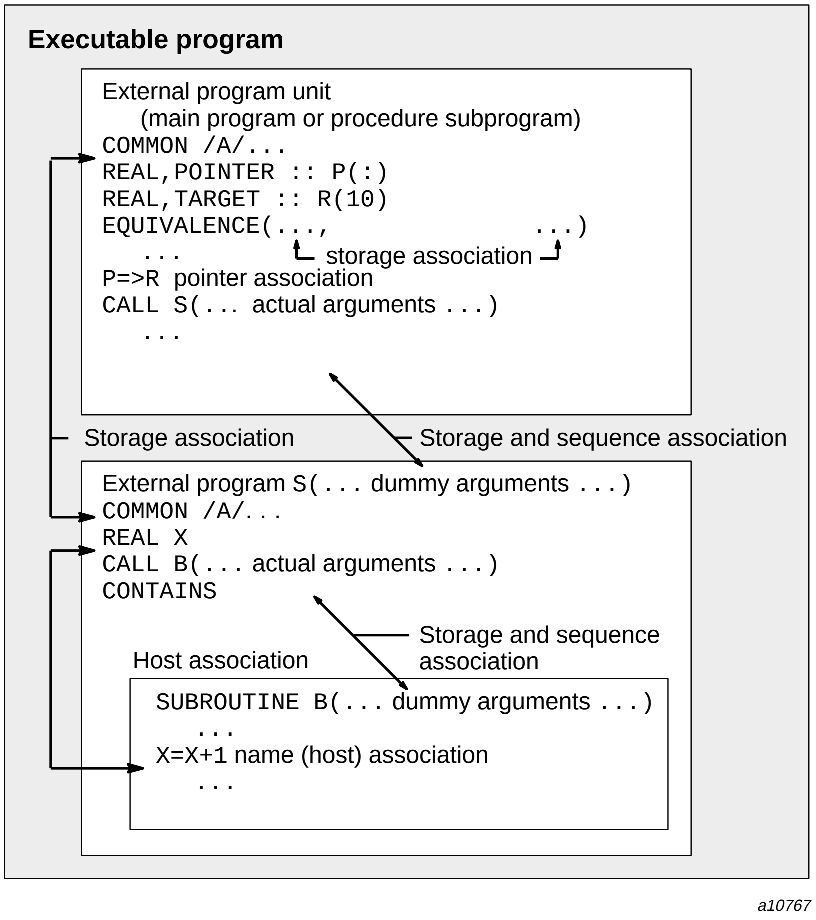

Executable program
External program unit
(main program or procedure subprogram)
COMMON /A/...
REAL,POINTER :: P(:)
REAL,TARGET :: R(10)
EQUIVALENCE(...,   ...)
...
storage association
P=>R pointer association
CALL S(... actual arguments ...)
...
Storage association
Storage and sequence association
External program S(... dummy arguments ...)
COMMON /A/...
REAL X
CALL B(... actual arguments ...)
CONTAINS
Storage and sequence association
Host association
SUBROUTINE B(... dummy arguments ...)
...
X=X+1 name (host) association
...
a10767

Figure 18. Associations between two nonmodule scoping units

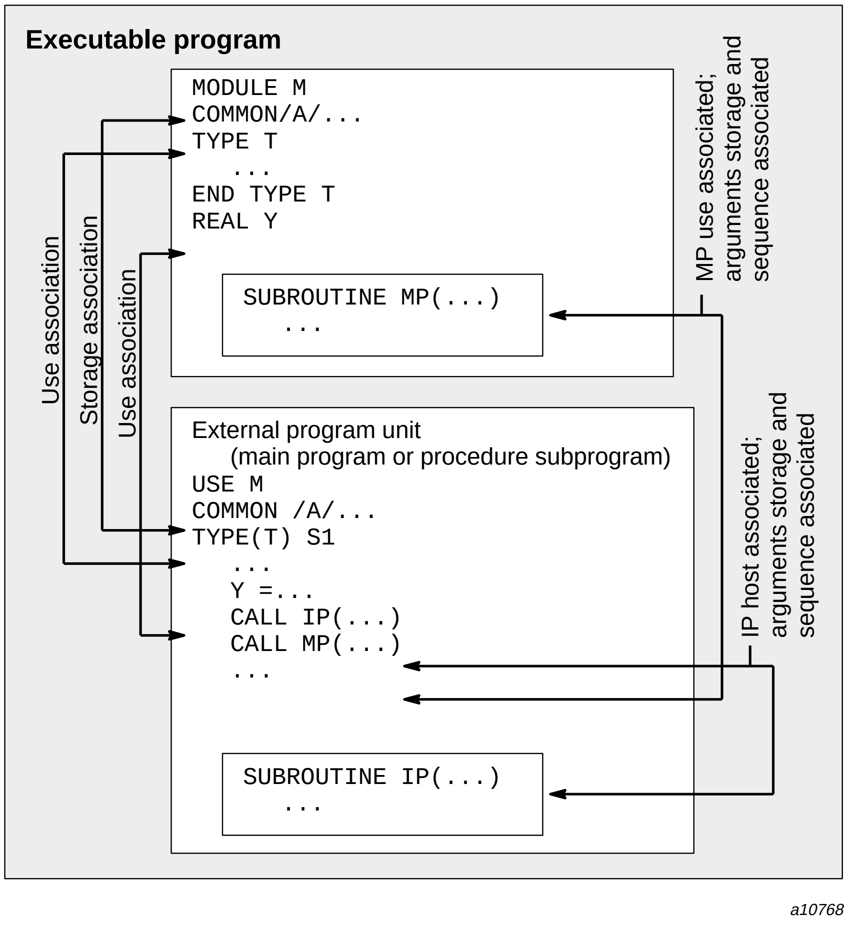


Figure 19. Associations between a module scoping unit and a nonmodule scoping unit

6.2.1 Name association

Name association permits access to the same entity (either data or a procedure) from different scoping units by the same or a different name. There are three forms of name association: argument, use, and host.

6.2.1.1 Argument association

Argument association is explained in detail in Section 4.7, page 200. It establishes a correspondence between the actual argument in the scoping unit containing the procedure reference and the dummy argument in the scoping unit defining the procedure. An actual argument can be the name of a variable or procedure, or it can be an expression. The dummy argument name is used in the procedure definition to refer to the actual argument, whether it is a name or an expression. When the program returns from the procedure, the actual and dummy arguments become disassociated.

6.2.1.2 Use association

Use association causes an association between entities in the scoping unit of a module and the scoping unit containing a `USE` statement referring to the module. It provides access to entities specified in the module. The default situation is that all public entities in the module are accessed by the name used in the module, but entities can be renamed selectively in the `USE` statement and excluded with the `ONLY` option. Use association is explained in Section 3.6.4.5, page 161.

If renaming occurs by a `USE` statement, the type, type parameters, and other attributes of the local name are those of the module entity. No respecification can occur in the scoping unit containing the `USE` statement, other than another module adding the `PRIVATE` attribute.

When an entity is renamed by a `USE` statement, the original name in the module can be used as a local name for a different entity in the scoping unit containing the `USE` statement. There would be no name conflict.

The `PUBLIC` and `PRIVATE` access specifications are determined in the module referenced in the `USE` statement. The `PUBLIC` attribute can be overridden by the referencing scoping unit if the referencing scoping unit is a module (see the example in Section 3.6.4.5, page 161, for an exception to this).

6.2.1.3 Host association

Host association causes an association between entities in a host scoping unit and the scoping unit of an internal procedure, module procedure, or derived-type definition. The basic idea of host association is that entities in the host (for example, host variables) are also available in any procedures or derived-type definitions within the host. As with default use association, such entities are known by the same name in the internal or module procedure or derived-type definition as they are known in the host. There is no mechanism for renaming entities, but the association of names between the host and the contained scoping unit can be replaced by the local declaration of an entity with that name; such a declaration blocks access to the entity of the same name in the host scoping unit. Host association is described in Section 3.4, page 145.

6.2.2 Pointer association

A *pointer* is a variable with the `POINTER` attribute. During program execution, the pointer variable is undefined, disassociated, or associated with a scalar or an array data object or function result. The association of pointers is dynamic throughout a program; that is, the association can be changed as needed during execution. Pointers are initially undefined.

Pointers can be associated in one of the following ways:

- By pointer assignment, pointers are associated with other pointers or with scalar or array data objects that have the `TARGET` attribute
- By execution of an `ALLOCATE` statement, pointers are associated with previously unnamed space

Associated pointers can become disassociated or undefined. Disassociated pointers can become associated or undefined. Undefined pointers can become associated.

Associated pointers become disassociated when the association is nullified (`NULLIFY` statement) or when the pointer is deallocated (`DEALLOCATE` statement).

6.2.3 Storage association

Storage association is the provision that two or more variables can share the same memory space. This allows the same value to be referenced by more than one name, achieving an effect similar to that of name association and pointer association. Consider the following simple example:

```
EQUIVALENCE (X, Y)
```

Variables `X` and `Y` share the same memory location; changing the value of either one affects the value of the other.

The effects of `EQUIVALENCE` and `COMMON` statements can be complicated because of partially overlapping variables and storage association between different data types. The concept of storage association is used to describe these effects.

6.2.3.1 Storage units and storage sequence

A *storage unit* corresponds to a particular part of memory that holds a single value. Thus, memory may be thought of as a sequence of storage units, each possibly holding a value. A storage unit can be a *numeric storage unit*, a *character storage unit*, or an *unspecified storage unit*.

A sequence of any number of consecutive storage units is a *storage sequence*. A storage sequence has a size, which is the number of storage units in the sequence.

6.2.3.2 Fortran data and storage units

This section describes some of the relationships between Fortran data and storage units.

All of the following data objects are nonpointer values unless explicitly specified otherwise:

- A scalar data object of type default integer, default real, or default logical occupies one numeric storage unit. Default complex and double-precision real values occupy two, consecutive, numeric storage units.
- A single character occupies one character storage unit.
- All other values, that is, pointers and all values with nondefault kinds, occupy unspecified storage units. An unspecified storage unit is treated as a different object for each of these types of values. For example, a storage unit for a pointer to a default integer occupies an unspecified storage unit that is different than the unspecified storage unit for an integer of nondefault type.

Composite objects occupy storage sequences, depending on their form, as follows:

- A scalar character object of length *len* has *len* consecutive character storage units.
- An array occupies a storage sequence consisting of one storage unit of the appropriate sort for each element of the array in array element order.
- A scalar of a sequence derived type occupies a storage sequence consisting of storage units corresponding to the components of the structure, in the

order they occur in the derived-type definition. Recall that to be a sequence type, the derived type must contain the SEQUENCE statement.

- Each common block has a storage sequence as described in the *Fortran Language Reference Manual, Volume 1*, publication SR–3902.
- Each ENTRY statement in a function subprogram has a storage sequence as described in Section 4.4.3, page 192.
- EQUIVALENCE statements create storage sequences from the storage units of the objects making up the equivalence lists.

Two objects that occupy the same storage sequence are *storage associated*. Two objects that occupy parts of the same storage sequence are *partially storage associated*.

6.2.3.3 Partial association

Partial association applies to character data and other composite objects in COMMON or EQUIVALENCE statements. When such objects only partially overlap in storage, they are then said to be *partially associated*. For example, two character strings are partially associated if substrings of each share the same storage but the entire strings do not.

6.2.3.4 Examples

This section contains examples of storage sequences of various sorts and illustrations of how equivalencing causes association of storage sequences and the data objects in them.

Example 1: This is a simple example involving numeric storage units. Consider the following code fragment:

```
COMPLEX :: C
REAL    :: X(0:5)
EQUIVALENCE (C, X(3))
```

The storage sequence occupied by C consists of two numeric storage units, one for the real part and one for the imaginary part.

The storage sequence occupied by X consists of six numeric storage units, one for each element of the array.

The EQUIVALENCE statement indicates that X(3) and the real part of C occupy the same storage unit, creating the following association (items above and below each other are storage associated):

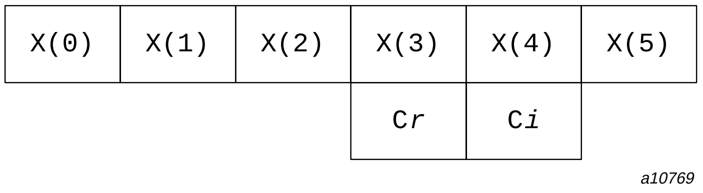


Figure 20. Example 1

Example 2: The following example deals with character data. Consider this code fragment:

```
CHARACTER A(2,2)*2, B(2)*3, C*5
EQUIVALENCE (A(2,1)(1:1), B(1)(2:3), C(3:5))
```

A, B, and C occupy character storage sequences of size 8, 6, and 5 respectively, and the EQUIVALENCE statement sets up the following associations:

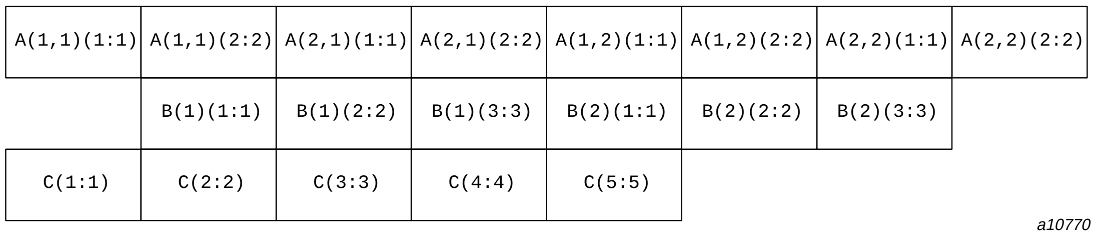


Figure 21. Example 2

6.2.4 Sequence association

Sequence association is a special form of argument association that applies to character, array, and sequence structure arguments. For more information on sequence association, see Section 4.7.2, page 204.

6.3 Definition status

The value of a variable can be used safely when that value is well-defined and predictable. There are a number of things that can cause the value of a variable to become ill-defined or unpredictable during the course of program execution. Informally, *defined* is the term given to a variable whose value is well-defined and predictable. *Undefined* is the term given to a variable whose value is ill-defined or unpredictable.

Using undefined values does not conform to the standard. Unfortunately, either the compiler cannot always check for undefined conditions or it might be too costly to do so. It is your responsibility to avoid using undefined values.

During execution of a program, variables are said to be *defined* or *undefined*. If a variable is defined, it has a value established during some statement execution (or event) in the program. If a variable is undefined, it is considered to not have a value. Variables are initially undefined except for initial values specified in `DATA` statements and type statements and objects of zero size. (Zero-sized arrays and zero-length character strings are always defined). As execution proceeds, other events may cause a variable to become defined or undefined. Undefined variables must not be referenced in a context in which the value of the variable is used.

6.3.1 Definition status of subobjects

An array element or array section is part of an array. A substring is part of a character variable. A component is part of a structure. An object of type complex consists of two parts, its real and imaginary parts. All parts of an object must be defined for the object to be defined. If any part of an object is undefined, that object is undefined. Zero-sized arrays and zero-length character strings are always defined.

6.3.2 Events that affect definition status of variables

Assignment defines the value of the variable on the left of the equal sign. Similarly, reading input establishes values for variables in the input list. Certain specifier variables are defined when a statement such as the `INQUIRE` statement is executed.

Returning from a procedure causes all unsaved local variables to become undefined. Deallocation and disassociation during program execution causes variables to become undefined. In addition, the process of defining an entity can cause certain associated or partially associated entities to become undefined. Lists of events that cause variables to be defined and undefined are provided in the following sections.

6.3.3 Events that cause variables to become defined

Variables become defined when the following events occur:

- Execution of an intrinsic assignment statement other than a masked array assignment statement causes the variable that precedes the equal sign to become defined. Execution of a defined assignment statement can cause all or part of the variable that precedes the equal sign to become defined.
- Execution of a masked array assignment statement can cause some or all of the array elements in the assignment statement to become defined
- As execution of an input statement proceeds, each variable that is assigned a value from the input file becomes defined at the time that data is transferred to it. Execution of a `WRITE` statement whose unit specifier identifies an internal file causes each record that is written to become defined.
- Execution of a `DO` statement causes the `DO` variable, if any, to become defined.
- Beginning execution of the action specified by an implied-`DO` list in an input/output (I/O) statement causes the implied-`DO` variable to become defined.
- Execution of an `ASSIGN` statement causes the variable in the statement to become defined with a statement label value.
- A reference to a procedure causes a dummy argument data object to become defined if the associated actual argument is defined with a value that is not a statement label. If only a subobject of an actual argument is defined, only the corresponding subobject of the associated dummy argument is defined.
- Execution of an I/O statement containing an `IOSTAT=` specifier causes the specified integer variable to become defined.
- Execution of a `READ` statement containing a `SIZE=` specifier causes the specified integer variable to become defined.
- Execution of an `INQUIRE` statement causes any variable that is assigned a value during the execution of the statement to become defined if no error condition exists.
- When a character storage unit becomes defined, all associated character storage units become defined.

- When a numeric storage unit becomes defined, all associated numeric storage units of the same type become defined. Variables associated with the variable in an `ASSIGN` statement, however, become undefined as integers when the `ASSIGN` statement is executed. When an entity of double-precision real type becomes defined, all totally associated entities of double-precision real type become defined.
- When an unspecified storage unit becomes defined, all associated unspecified storage units become defined.
- When a default complex entity becomes defined, all partially associated default real entities become defined.
- When both parts of a default complex entity become defined as a result of partially associated default real or default complex entities becoming defined, the default complex entity becomes defined.
- When all components of a numeric sequence structure or character sequence structure become defined as a result of partially associated objects becoming defined, the structure becomes defined.
- Execution of an `ALLOCATE` or `DEALLOCATE` statement with a `STAT=` specifier causes the variable specified by the `STAT=` specifier to become defined.
- Execution of a pointer assignment statement that associates a pointer with a target that is defined causes the pointer to become defined.

6.3.4 Events that cause variables to become undefined

Variables become undefined when the following events occur:

- When a variable of a given type becomes defined, all associated variables of different type become undefined. However, when a variable of type default real is partially associated with a variable of type default complex, the complex variable does not become undefined when the real variable becomes defined, and the real variable does not become undefined when the complex variable becomes defined. When a variable of type default complex is partially associated with another variable of type default complex, definition of one does not cause the other to become undefined.
- Execution of an `ASSIGN` statement causes the variable in the statement to become undefined as an integer. Variables that are associated with the variable also become undefined.

- If the evaluation of a function can cause an argument of the function or a variable in a module or in a common block to become defined, and if a reference to the function appears in an expression in which the value of the function is not needed to determine the value of the expression, the argument or variable becomes undefined when the expression is evaluated.
- The execution of a `RETURN` statement or an `END` statement within a subprogram causes all variables local to its scoping unit or local to the current instance of its scoping unit for a recursive invocation to become undefined, except for the following:
  - Variables with the `SAVE` attribute
  - Variables in blank common
  - Variables in a named common block

    Note: The Fortran 90 standard specifies that a variable in a named common block retains its value (remains defined) only if the common block also appears in at least one other scoping unit that is making either a direct or indirect reference to the current subprogram. The CF90 and MIPSpro 7 Fortran 90 compilers retain the value regardless of whether the common block also appears in any other currently active scoping unit.
  - Variables accessed from the host scoping unit
  - Variables accessed from a module that also is accessed in at least one other scoping unit that is making either a direct or indirect reference to the module
  - Variables in a named common block that are initially defined and that have not been subsequently defined or redefined
- When an error condition or end-of-file condition occurs during execution of an input statement, all of the variables specified by the input list or namelist group of the statement become undefined.
- When an error or end-of-file condition occurs during execution of an I/O statement that includes implied-`DO` loops, all of the implied-`DO` variables become undefined.
- Execution of a defined assignment statement can leave all or part of the variable that precedes the equal sign undefined.

- Execution of a direct access input statement that specifies a record that has not been written previously causes all of the variables specified by the input list of the statement to become undefined.
- Execution of an `INQUIRE` statement can cause the `NAME=`, `RECL=`, and `NEXTREC=` variables to become undefined.
- When a character storage unit becomes undefined, all associated character storage units become undefined.
- When a numeric storage unit becomes undefined, all associated numeric storage units become undefined unless the undefinition is a result of defining an associated numeric storage unit of different type.
- When an entity of double-precision real type becomes undefined, all totally associated entities of double-precision real type become undefined.
- When an unspecified storage unit becomes undefined, all associated unspecified storage units become undefined.
- A reference to a procedure causes part of a dummy argument to become undefined if the corresponding part of the actual argument is defined with a value that is a statement label.
- When an allocatable array is deallocated, it becomes undefined. Successful execution of an `ALLOCATE` statement creates an array that is undefined.
- Execution of an `INQUIRE` statement causes all inquire specifier variables to become undefined if an error condition exists, except for the variable in the `IOSTAT=` specifier, if any.
- When a procedure is invoked, variables become undefined under the following circumstances:
  - An optional dummy argument that is not associated with an actual argument is undefined
  - A dummy argument with `INTENT(OUT)` is undefined
  - An actual argument associated with a dummy argument with `INTENT(OUT)` becomes undefined
  - A subobject of a dummy argument is undefined if the corresponding subobject of the actual argument is undefined
  - The result variable of a function is undefined

- When the association status of a pointer becomes undefined or disassociated, the pointer becomes undefined.

Glossary

argument keyword

The name of a dummy (or formal) argument. This name is used in the subprogram definition; it also may be used when the subprogram is invoked to associate an actual argument with a dummy argument. Using argument keywords allows the actual arguments to appear in any order. The Fortran 90 standard specifies argument keywords for all intrinsic procedures. Argument keywords for user-supplied external procedures may be specified in a procedure interface block.

array

(1) A data structure that contains a series of related data items arranged in rows and columns for convenient access. The C shell and the `awk`(1) command can store and process arrays. (2) In Fortran 90, an object with the `DIMENSION` attribute. It is a set of scalar data, all of the same type and type parameters. The rank of an array is at least 1, and at most 7. Arrays may be used as expression operands, procedure arguments, and function results, and they may appear in input/output (I/O) lists.

association

An association permits an entity to be referenced by different names in a scoping unit or by the same or different names in different scoping units. Several kinds of association exist. The principal kinds of association are pointer association, argument association, host association, use association, and storage association.

automatic variable

A variable that is not a dummy argument but whose declaration depends on a nonconstant expression (array bounds and/or character length).

Autotasking

A trademarked process of Cray Research that automatically divides a program into individual tasks and organizes them to make the most efficient use of the computer hardware.

bottom loading

An optimization technique used on some scalar loops in which operands are prefetched during each loop iteration for use in the next iteration. The operand is available as soon as the first loop instruction executes. A prefetch is performed even during the final loop iteration, before the loop's final jump test has been performed.

cache

In a processing unit, a high-speed buffer storage that is continually updated to contain recently accessed contents of main storage. Its purpose is to reduce access time. In disk subsystems, a method the channel buffers use to buffer disk data during transfer between the devices and memory.

cache line

On Cray MPP systems, a cache line consists of four quad words, which is the maximum size of a hardware message.

CIV

A constant increment variable is a variable that is incremented only by a loop invariant value (for example, in a loop with index J, the statement J = J + K, in which K can be equal to 0, J is a CIV).

constant

A data object whose value cannot be changed. A named entity with the `PARAMETER` attribute is called a named constant. A constant without a name is called a literal constant.

construct

A sequence of statements that starts with a `SELECT CASE`, `DO`, `IF`, or `WHERE` statement and ends with the corresponding terminal statement.

control construct

An action statement that can change the normal execution sequence (such as a `GO TO`, `STOP`, or `RETURN` statement) or a `CASE`, `DO`, or `IF` construct.

critical region

On Cray MPP systems, a synchronization mechanism that enforces serial access to a piece of code. Only one PE may execute in a critical region at a time.

data entity

A data object, the result of the evaluation of an expression, or the result of the execution of a function reference (also called the function result). A data entity always has a type.

data object

A constant, a variable, or a part of a constant or variable.

declaration

A nonexecutable statement that specifies the attributes of a data object (for example, it may be used to specify the type of a variable or function result or the shape of an array).

definition

This term is used in two ways. (1) A data object is said to be defined when it has a valid or predictable value; otherwise, it is undefined. It may be given a valid value by execution of statements such as assignment or input. Under certain circumstances, it may subsequently become undefined. (2) Procedures and derived types are said to be defined when their descriptions have been supplied by the programmer and are available in a program unit.

derived type

A type that is not intrinsic (a user-defined type); it requires a type definition to name the type and specify its components. The components may be of intrinsic or user-defined types. An object of derived type is called a structure. For each derived type, a structure constructor is available to specify values. Operations on objects of derived type must be defined by a function with an interface and the generic specifier `OPERATOR`. Assignment for derived type objects is defined intrinsically, but it may be redefined by a subroutine with the `ASSIGNMENT` generic specifier. Data objects of derived type may be used as procedure arguments and function results, and they may appear in input/output (I/O) lists.

designator

Sometimes it is convenient to reference only part of an object, such as an element or section of an array, a substring of a character string, or a component of a structure. This requires the use of the name of the object followed by a selector that selects a part of the object. A name followed by a selector is called a **designator**.

entity

(1) In Open Systems Interconnection (OSI) terminology, a layered protocol machine. An entity in a layer performs the functions of the layer in one computer system, accessing the layer entity below and providing services to the layer entity above at local service access points. (2) In Fortran 90, a general term used to refer to any Fortran 90 concept (for example, a program unit, a common block, a variable, an expression value, a constant, a statement label, a construct, an operator, an interface block, a derived type, an input/output (I/O) unit, a name list group, and so on).

executable construct

A statement (such as a `GO TO` statement) or a construct (such as a `DO` or `CASE` construct).

expression

A set of operands, which may be function invocations, and operators that produce a value.

extent

A structure that defines a starting block and number of blocks for an element of file data.

function

Usually a type of operating-system-related function written outside a program and called in to do a specific function. Smaller and more limited in capability than a utility. In a programming language, a function is usually defined as a closed subroutine that performs some defined task and returns with an answer, or identifiable return value.

The word "function" has a more specific meaning in Fortran than it has in C. In C, it is refers to any called code; in Fortran, it refers to a subprogram that returns a value.

generic specifier

An optional component of the INTERFACE statement. It can take the form of an identifier, an OPERATOR (defined_operator) clause, or an ASSIGNMENT (=) clause.

heap

A section of memory within the user job area that provides a capability for dynamic allocation. See the HEAP directive in SR-0066.

inlining

The process of replacing a user subroutine or function call with the definition itself. This saves subprogram call overhead and may allow better optimization of the inlined code. If all calls within a loop are inlined, the loop becomes a candidate for vectorization and/or tasking.

intrinsic

Anything that the language defines is intrinsic. There are intrinsic data types, procedures, and operators. You may use these freely in any scoping unit. Fortran programmers may define types, procedures, and operators; these entities are not intrinsic.

local

(1) A type of scope in which variables are accessible only to a particular part of a program (usually one module). (2) The system initiating the request for service. This term is relative to the perspective of the user.

multitasking

(1) The parallel execution of two or more parts of a program on different CPUs; these parts share an area of memory. (2) A method in multiuser systems that incorporates multiple interconnected CPUs; these CPUs run their programs simultaneously (in parallel) and shares resources such as memory, storage devices, and printers. This term can often be used interchangeably with **parallel processing**.

name

A term that identifies many different entities of a program such as a program unit, a variable, a common block, a construct, a formal argument of a subprogram (dummy argument), or a user-defined type (derived type). A name may be associated with a specific constant (named constant).

operator

(1) A symbolic expression that indicates the action to be performed in an expression; operator types include arithmetic, relational, and logical. (2) In Fortran 90, an operator indicates a computation that involves one or two operands. Fortran 90 defines several intrinsic operators (for example, +, -, *, /, ** are numeric operators, and .NOT., .AND., and .OR. are logical operators). Users also may define operators for use with operands of intrinsic or derived types.

overindexing

The nonstandard practice of referencing an array with a subscript not contained between the declared lower and upper bounds of the corresponding dimension for that array. This practice sometimes, but not necessarily, leads to referencing a storage location outside of the entire array.

parallel processing

Processing in which multiple processors work on a single application simultaneously.

pointer

(1) A data item that consists of the address of a desired item. (2) A symbol that moves around a computer screen under the control of the user.

procedure

(1) A named sequence of control statements and/or data that is saved in a library for processing at a later time, when a calling statement activates it; it provides the capability to replace values within the procedure. (2) In Fortran 90, procedure is defined by a sequence of statements that expresses a computation that may be invoked as a subroutine or function during program execution. It may be an intrinsic procedure, an external procedure, an internal procedure, a module procedure, a dummy procedure, or a statement function. If a subprogram contains an ENTRY statement, it defines more than one procedure.

procedure interface

In Fortran 90, a sequence of statements that specifies the name and characteristics of one or more procedures, the name and attributes of each dummy argument, and the generic specifier by which it may be referenced if any. See **`generic specifier`**.

In FORTRAN 77 and Fortran 90, a **`generic function`** is one whose output value data type is determined by the data type of its input arguments. In FORTRAN 77, the only generic functions allowed are those that the standard defines. In Fortran 90, programmers may construct their own generic function by creating "generic interface," which is like a regular procedure interface, except that it has a "generic specifier" (the name of the generic function) after the keyword `INTERFACE`.

reduction loop

A loop that contains at least one statement that reduces an array to a scalar value by doing a cumulative operation on many of the array elements. This involves including the result of the previous iteration in the expression of the current iteration.

reference

A data object reference is the appearance of a name, designator, or associated pointer in an executable statement that requires the value of the object. A procedure reference is the appearance of the procedure name, operator symbol, or assignment symbol in an executable program that requires execution of the procedure. A module reference is the appearance of the module name in a `USE` statement.

scalar

(1) In Fortran 90, a single object of any intrinsic or derived type. A structure is scalar even if it has a component that is an array. The rank of a scalar is 0. (2) A nonvectorized, single numerical value that represents one aspect of a physical quantity and may be represented on a scale as a point. This term often refers to a floating-point or integer computation that is not vectorized; more generally, it also refers to logical and conditional (jump) computation.

scope

The region of a program in which a variable is defined and can be referenced.

scoping unit

Part of a program in which a name has a fixed meaning. A program unit or subprogram generally defines a scoping unit. Type definitions and procedure interface bodies also constitute scoping units. Scoping units do not overlap, although one scoping unit may contain another in the sense that it surrounds it. If a scoping unit contains another scoping unit, the outer scoping unit is referred to as the host scoping unit of the inner scoping unit.

search loop

A loop that can be exited by means of an IF statement.

sequence

A set ordered by a one-to-one correspondence with the numbers 1, 2, through **n**. The number of elements in the sequence is **n**. A sequence may be empty, in which case, it contains no elements.

shared

Accessible by multiple parts of a program. Shared is a type of scope.

shell variable

A name representing a string value. Variables that are usually set only on a command line are called **parameters** (positional parameters and keyword parameters). Other variables are simply names to which a user (user-defined variables) or the shell itself may assign string values. The shell has predefined shell variables (for example, HOME). Variables are referenced by prefixing the variable name by a $ (for example, $HOME).

software pipelining

Software pipelining is a compiler code generation technique in which operations from various loop iterations are overlapped in order to exploit instruction-level parallelism, increase instruction issue rate, and better hide memory and instruction latency. As an optimization technique, software pipelining is similar to bottom loading, but it includes additional, and more efficient, scheduling optimizations.

Cray compilers perform safe bottom loading by default. Under these conditions, code generated for a loop contains operations and stores associated with the present loop iteration and contains loads associated with the next loop iteration. Loads for the first iteration are generated in the loop preamble.

When software pipelining is performed, code generated for the loop contains loads, operations, and stores associated with various iterations of the loop. Loads and operations for first iterations are generated in the preamble to the loop. Operations and stores for last iterations of loop are generated in the postamble to the loop.

statement keyword

A keyword that is part of the syntax of a statement. Each statement, other than an assignment statement and a statement function definition, begins with a statement keyword. Examples of these keywords are `IF`, `READ`, and `INTEGER`. Statement keywords are not reserved words; you may use them as names to identify program elements.

stripmining

A single-processor optimization technique in which arrays, and the program loops that reference them, are split into optimally-sized blocks, termed strips. The original loop is transformed into two nested loops. The inner loop references all data elements within a single strip, and the outer loop selects the strip to be addressed in the inner loop. This technique is often performed by the compiler to maximize the usage of cache memory or as part of vector code generation.

structure

A language construct that declares a collection of one or more variables grouped together under one name for convenient handling. In C and C++, a structure is defined with the `struct` keyword. In Fortran 90, a derived type is defined first and various structures of that type are subsequently declared.

subobject

Parts of a data object may be referenced and defined separately from other parts of the object. Portions of arrays are array elements and array sections. Portions of character strings are substrings. Portions of structures are structure components. Subobjects are referenced by designators and are considered to be data objects themselves.

subroutine

A series of instructions that accomplishes a specific task for many other routines. (A subsection of a user-written program of varying size and, therefore, function. It is written within the program. It is not a subsection of a routine.) It differs from a main routine in that one of its parameters must specify the location to which to return in the main program after the function has been accomplished.

TKR

An acronym that represents attributes for argument association. It represents the data type, kind type parameter, and rank of the argument.

type parameter

Two type parameters exist for intrinsic types: kind and length. The kind type parameter `KIND` indicates the decimal range for the integer type, the decimal precision and exponent range for the real and complex types, and the machine representation method for the character and logical types. The length type parameter `LEN` indicates the length of a character string.

variable

(1) A name that represents a string value. Variables that usually are set only on a command line are called parameters. Other variables are simply names to which the user or the shell may assign string values. (2) In Fortran 90, data object whose value can be defined and redefined. A variable may be a scalar or an array. (3) In the shell command language, a named parameter. See also **`shell variable`**.

Index