CF90 Fortran Language Reference Manual

Volume 2

SR–3903 2.0

Cray Research, Inc.

Cray Research Software Documentation Map

The information on the following pages highlights the major body of documentation available for Cray Research customers. The manuals are organized into categories by product designation:

Product	Description
Programming environments	Cray Research programming environments provide an integrated development environment by combining compilers with development tools and libraries.
UNICOS operating system	The UNICOS operating system provides basic system control and management functions. It provides services, including basic batch and interactive system access for users, operating system interfaces for programmers, networking facilities, and sophisticated resource management for administrators. UNICOS MAX, in conjunction with UNICOS on the host system, provides operating system functionality on CRAY T3D systems.
Networking products	Cray Research networking products provide media connections, network programming access, and batch and interactive access for users across a heterogeneous network of computers.
Operating systems supplementary products	The operating systems supplementary products include a variety of products that are used in conjunction with Cray Research operating systems but are not included on the same media.

To use the map, find the product designation closest to your specific needs or role as a Cray Research system user. Documents listed under other categories also may be of interest to you; documents are listed only once, underneath the product to which they most directly apply. The date on the bottom of the page tells you when the information was last revised.

For more information

In addition to the map, you can use the following publications to find documentation specific to your needs:

- *Software Documentation Ready Reference,* publication SQ–2122, serves as a general index to the Cray Research documentation set. The booklet lists documents and man pages according to topic.
- *Software Overview for Users*, publication SG–2052, provides an overview of Cray Research software, including operating systems, programming environments, libraries, tools, applications, and so on. It directs you to documentation that contains user-level information.
- *User Publications Catalog*, publication CP–0099, briefly describes all Cray Research manuals available to you, including some not shown on the map. Cray Research customers who subscribe to the Cray Inform (CRInform) program can access this information on CRInform.

Ordering

To obtain Cray Research publications, order them by publication number from the Distribution Center:

Cray Research, Inc.		
Distribution Center	Order desk	+1–612–683–5907
2360 Pilot Knob Road	Fax number	+1–612–452–0141
Mendota Heights, MN 55120	Email	`orderdsk@cray.com`
USA		

Programming Environments

Ada

Cray Ada Environment, Volume 1: Reference Manual (SR–3014)

Cray Ada Environment, Volume 2: Programming Guide (SR–3082)

Assemblers

Cray Assembler for MPP (CAM) Reference Manual (SR–2510)†

Cray Assembly Language (CAL) for Cray PVP Systems Ready Reference (SQ–3110)

Cray Assembly Language (CAL) for Cray PVP Systems Reference Manual (SR–3108)†

Fortran Compilers

CF90 Ready Reference (SQ–3900)

CF90 Commands and Directives Reference Manual (SR–3901)†

CF90 Fortran Language Reference Manual, Vol. 1 (SR–3902)†

CF90 Fortran Language Reference Manual, Vol. 2 (SR–3903)†

CF90 Fortran Language Reference Manual, Vol. 3 (SR–3905)†

CF90 SPARC Ready Reference (SQ–3904)

Introducing the CF90 SPARC Programming Environment (IN–2155)†

Cray MPP Fortran Reference Manual (SR–2504)†

CF77 Fortran Language Reference Manual (SR–3772)

Cray Standard C Compiler

Cray Standard C Reference Manual (SR–2074)†

Cray Standard C Ready Reference (SQ–2076)

Cray Standard C Reference Manual for MPP (SR–2506)†

C++ Compiler

Cray C/C++ Reference Manual (SR–2179)†

Cray C/C++ Ready Reference (SQ–2180)

CrayLibs

Application Programmer's I/O Guide (SG–2168)†

Application Programmer's Library Reference Manual (SR–2165)★

Introducing CrayLibs (IN–2167)†

Math Library Reference Manual (SR–2138)★

Parallel Virtual Machine (PVM) Reference Card (SQ–2512)

PVM and HeNCE Programmer's Manual (SR–2501)†

Scientific Libraries Reference Manual (SR–2081)★

CrayTools

Compiler Information File (CIF) Reference Manual (SR–2401)†

Cray TotalView Debugger Reference Card (SQ–2519)

Guide to Parallel Vector Applications (SG–2182)†

Introducing the Cray TotalView Debugger (IN–2502)†

Introducing the MPP Apprentice Tool (IN–2511)†

Introducing the Program Browser (IN–2140)†

UNICOS CDBX Symbolic Debugger Reference Card (SQ–2110)

UNICOS CDBX Symbolic Debugger Reference Manual (SR–2091)†

UNICOS CDBX Debugger User's Guide (SG–2094)†

General

Cray Research MPP Software Guide (SG–2508)†

CRAY T3D Emulator User's Guide (SG–2500)†

Installing Programming Environment Products (SG–5191)

Migrating to the CRAY T90 Series IEEE Floating Point (SN–2194)†

Optimizing Code on Cray PVP Systems (SG–2192)†

Programming Environment Installation Guide (SG–5217)

Message Passing Toolkit

MPT: PVM Programmer's Manual (SR–2196)†

MPT: MPI Programmer's Manual (SR–2197)†

† Available with CrayDoc
★ Man pages available with the `man` command

UNICOS Operating System

General UNICOS

Software Documentation Ready Reference (SQ–2122)

Software Overview for Users (SG–2052)†

User's Guide to Online Information (SG–2143)†

UNICOS `ed` Reference Card (SQ–2055)

UNICOS Environment Variables Ready Reference (SQ–2117)

UNICOS File Formats and Special Files Reference Manual (SR–2014)*

UNICOS Standard Shell Reference Card (SQ–2115)

UNICOS User Commands Ready Reference (SQ–2056)*

UNICOS User Commands Reference Manual (SR–2011)*

UNICOS `vi` Reference Card (SQ–2054)

UNICOS Visual Interfaces User's Guide (SG–3094)†

Networking Feature Use and Administration

Front-end Protocol Internal Reference Manual (SM–0042)

FTA User and Administrator Manual (SG–2144)†

Kerberos Administrator's Guide (SG–2306)†

Kerberos User's Guide (SG–2409)†

Network Queuing System (NQS) User's Guide (SG–2105)†

ONC+ Technology for the UNICOS Operating System (SG–2169)†

Remote Procedure Call (RPC) Reference Manual (SR–2089)†

TCP/IP Network User's Guide (SG–2009)†

UNICOS Networking Facilities Administrator's Guide (SG–2304)†

UNICOS NQS and NQE Administrator's Guide (SG–2305)†

UNICOS Station Call Processor (USCP) Administrator's Guide (SG–2308)†

UNICOS MAX

Cray MPP Loader User's Guide (SG–2514)†

Cray MPP Simulator User's Guide (SG–2503)†

CRAY T3D Administrator's Guide (SG–2507)†*

UNICOS Basic Administration and Operation

CRAY EL Series IOS Commands Ready Reference (SQ–2162)*

CRAY EL Series IOS Commands Reference Manual (SR–2408)*

CRAY EL Series IOS Messages (SQ–2402)

CRAY IOS-V Commands Reference Manual (SR–2170)*

CRAY IOS-V Messages (SQ–2172)†

General UNICOS System Administration (SG–2301)†

`nmake` Reference Card (SQ–2146)

OLNET Online Diagnostic Network Communications Program Maintenance Manual for UNICOS (SD–1021)†

Online Maintenance Tools Guide for Cray PVP Systems (SD–1012)†

UNICOS Basic Administration Guide for CRAY J90 Series and CRAY EL Series (SG–2416)†

UNICOS Configuration Administrator's Guide (SG–2303)†

UNICOS Installation Guide (SG–2112)

UNICOS Installation Guide for the CRAY EL Series (SG–5201)

UNICOS Installation Guide for the CRAY J90 Series (SG–5271)

UNICOS Installation Menu System Reference Card (SQ–2411)

UNICOS Installation/Configuration Menu System User's Guide (SG–2412)†

UNICOS Resource Administration (SG–2302)†

UNICOS Shared File System (SFS) Administrator's Guide (SG–2114)†

UNICOS Feature Use and Administration

UNICOS Administrator Commands Reference Manual (SR–2022)*

UNICOS Administrator Commands Ready Reference (SQ–2413)*

UNICOS Message System Programmer's Guide (SG–2121)†

UNICOS Multilevel Security (MLS) Feature User's Guide (SG–2111)†

UNICOS System Security Overview for Administrators (SG–2141)†

UNICOS Tape Subsystem Administrator's Guide (SG–2307)†

UNICOS Tape Subsystem User's Guide (SG–2051)†

UNICOS under UNICOS Administrator's Guide (SG–2156)†

† Available with CrayDoc
* Man pages available with the `man` command

UNICOS Operating System (continued)

Programming Support

Segment Loader (SEGLDR) and `ld` Reference Manual (SR–0066)†

SEGLDR Reference Card (SQ–0303)

UNICOS System Libraries Ready Reference (SQ–2147)*

UNICOS System Libraries Reference Manual (SR–2080)*

Programming Support (continued)

UNICOS Macros and Opdefs Reference Manual (SR–2403)*

UNICOS System Calls Reference Manual (SR–2012)*

UNICOS X Window System Reference Manual (SR–2101)†

X Window System Resources Ready Reference (SQ–2123)

Networking Products

ATM

Asynchronous Transfer Mode (ATM) Administrator's Guide (SG–2193)†

fy Driver

fy Driver Administrator's Guide (SG–2132)

HIPPI

HIPPI Channel Administrator's Guide (SG–2159)

Network Queuing Environment (NQE)

NQE User's Guide (SG–2148)†

NQE Ready Reference (SG–2149)

NQE Administration (SG–2150)†

Introducing NQE (IN–2153)†

RQS MVS

RQS User's Guide for IBM MVS Systems (SG–2405)

RQS Administrator's Guide for IBM MVS Systems (SG–2406)

Fiber Distributed Data Interface (FDDI)

Fiber Distributed Data Interface (FDDI) Administrator's Guide (SG–3091)

RQS VMS

RQS/NQC User's Guide for DEC OpenVMS/VAX and AXP Systems (SV–3151)

RQS/NQC Administrator's Guide for DEC OpenVMS/VAX and AXP Systems (SV–3152)

SUPERLINK VM

SUPERLINK VM User's Guide: RQS Component (SI–0170)

SUPERLINK VM Administrator's Guide: RQS Component (SI–0171)

SUPERLINK VMS

SUPERLINK Administrator's Guide for DEC VAX/VMS Systems (SV–3154)

SUPERLINK User's Guide for DEC VAX/VMS Systems (SV–3153)

Operating Systems Supplementary Products

CCMT

Concurrent Maintenance Tools (CCMT) Guide (SD–1027)†

Cray/REELlibrarian (CRL)

Cray/REELlibrarian (CRL) Administrator's Guide (SG–2127)

Cray/REELlibrarian (CRL) User's Guide (SG–2126)

SMARTE

System Maintenance and Remote Testing Environment (SMARTE) Guide (SD–1017)†

Data Migration Facility (DMF)

Cray Data Migration Facility (DMF) Administrator's Guide (SG–2135)†*

Cray Data Migration (DMF) MSP Writer's Guide (SN–2098)†

OWS

Support System Reference Manual (SR–3077)*

Support System Operator's Guide (SG–3078)†

Support System Administrator's Guide (SG–3079)†

Support System Ready Reference (SQ–3080)

† Available with CrayDoc
* Man pages available with the `man` command

New Features

CF90 Fortran Language Reference Manual, Volume 2 SR–3903 2.0

The CF90 compiler now supports CRAY T3E, CRAY J90, and CRAY T90 systems. These new platforms are in addition to the CRAY Y-MP M90, CRAY C90, CRAY EL, CRAY Y-MP E, and SPARC systems that were supported in previous releases. (Note that implementation on CRAY T3E systems is deferred.)

The CF90 2.0 release offers increased performance and many user-interface enhancements.

The `SYSTEM_CLOCK`, `RESHAPE`, `TRANSFER`, `MAXLOC`, `MINLOC`, and `SHAPE` intrinsic functions now produce inline code. More instances of `LBOUND` and `UBOUND` are inlined. `ALL`, `ANY`, and `COUNT` are inlined when the `DIM` argument is a constant at compile time. `MINVAL`, `MAXVAL`, `PRODUCT`, `SUM`, `SPREAD`, and `EOSHIFT` are inlined if `DIM` is a constant or if the intrinsic function returns a scalar result.

Cray PVP systems now support the following Bit Matrix Multiply (BMM) intrinsic functions: `M@CLR`, `M@LD`, `M@MX`, `M@UL`, and `M@LDMX`.

SPARC systems, CRAY T3E systems, and CRAY T90 systems that support IEEE floating-point arithmetic now support the following intrinsic procedures: `CLEAR_IEEE_EXCEPTION`, `DISABLE_IEEE_INTERRUPT`, `ENABLE_IEEE_INTERRUPT`, `GET_IEEE_EXCEPTIONS`, `GET_IEEE_INTERRUPTS`, `GET_IEEE_ROUNDING_MODE`, `GET_IEEE_STATUS`, `IEEE_BINARY_SCALE`, `IEEE_CLASS`, `IEEE_COPY_SIGN`, `IEEE_EXPONENT`, `IEEE_FINITE`, `IEEE_INT`, `IEEE_IS_NAN`, `IEEE_NEXT_AFTER`, `IEEE_REAL`, `IEEE_REMAINDER`, `IEEE_UNORDERED`, `INT_MULT_UPPER`, `SET_IEEE_EXCEPTION`, `SET_IEEE_EXCEPTIONS`, `SET_IEEE_INTERRUPTS`, `SET_IEEE_ROUNDING_MODE`, `SET_IEEE_STATUS`, `TEST_IEEE_EXCEPTION`, and `TEST_IEEE_INTERRUPT`.

CRAY T3E systems now support the following intrinsic procedures: `MEMORY_BARRIER`, `MY_PE`, `REMOTE_WRITE_BARRIER`, and `WRITE_MEMORY_BARRIER`.

Record of Revision

The date of printing or software version number is indicated in the footer.

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.

Preface

The CF90 compiler implements the Fortran 90 standard. This reference manual describes the Fortran programming language as implemented by the Cray Research CF90 compiler.

The CF90 release 2.0 compiler runs on the following systems:

- CRAY T90, CRAY J90, CRAY C90, CRAY EL, CRAY Y-MP E, and CRAY Y-MP M90 systems running the UNICOS 8.0, or later, operating system. This manual refers to these systems as *Cray Research parallel vector systems* or *Cray PVP systems*.
- CRAY T3E systems. This manual sometimes refers to the CRAY T3E system and other Cray Research massively parallel processor systems as *Cray MPP systems*.
- SPARC systems running the Solaris 2.3, or later, operating system. This manual refers to these systems as *SPARC systems*.

> **Note:** This manual describes how the CF90 compiler works on Cray PVP systems, CRAY T3E systems, and SPARC systems. Implementation of the CF90 2.0 compiler on CRAY T3E systems is deferred.

The CF90 compiler was 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. Because the ANSI Fortran 90 standard is a superset of the FORTRAN 77 standard, the CF90 compiler 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 and for other vendors. To maintain conformance to the Fortran 90 standard, Cray Research may need to change the behavior of certain CF90 features in future releases based upon the outcomes of the outstanding interpretations to the standard.

Related CF90 publications

This manual is one of a set of Cray Research, Inc. (CRI) manuals that describes the CF90 compiler. The other manuals in this set are as follows:

- *Introducing the CF90 SPARC Programming Environment,* publication IN–2155
- *CF90 Ready Reference*, publication SQ–3900
- *CF90 Commands and Directives Reference Manual*, publication SR–3901
- *CF90 Fortran Language Reference Manual, Volume 1*, publication SR–3902
- *CF90 Fortran Language Reference Manual, Volume 3*, publication SR–3905
- *CF90 Ready Reference for SPARC Systems*, publication SQ–3904

Sections 1 through 10 of the *CF90 Fortran Language Reference Manual, Volume 1*, publication SR–3902, correspond to sections 1 through 10 of the Fortran 90 standard. Sections 1 through 4 of this manual correspond to sections 11 through 14 of the Fortran 90 standard. The standard is the complete official description of the language. This manual and the standard can be examined in parallel for insights into the Fortran language.

The *CF90 Fortran Language Reference Manual, Volume 3*, publication SR–3905, contains information that supplements the *CF90 Fortran Language Reference Manuals*, *Volumes 1 and 2*.

CF90 messages

You can obtain CF90 compiler message explanations by using the online `explain`(1) command.

CF90 man pages

In addition to printed documentation, several online man pages describe aspects of the CF90 compiler. 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, col, and lpr commands. In the following example, these commands are used to print a copy of the f90(1) man page:

```
man f90 | 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 grep online, you can type man grep. To access information about egrep online, you can type either man grep or man egrep; both commands display the grep man page on your terminal.

The -p and -s options to the dmput(1) command require that the caller be super user.

Related Fortran publications

The following commercially available reference books are among those that you could 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.

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

Related programming environment publications

The CF90 compiler is part of the Fortran programming environment. The documentation available from Cray Research on the programming environment tools depends on your platform.

- If you are using the CF90 compiler on a SPARC system, you may want to read *Introducing the CF90 SPARC Programming Environment*, publication IN–2155. It describes the products in the environment and the Cray Research documentation that is available for them.
- If you are using the CF90 compiler on a Cray PVP system or a CRAY T3E system, the following Cray Research documents describe the products in the environment that may be of interest to you. These publications describe the operating system, input/output (I/O), and other topics:

Publication	Title
SR–0066	*Segment Loader (SEGLDR) and* `ld` *Reference Manual*
SR–2011	*UNICOS User Commands Reference Manual*
SR–2040	*UNICOS Performance Utilities Reference Manual*
SG–2052	*Software Overview for Users*
SQ–2056	*UNICOS User Commands Ready Reference*
SR–2165	*Application Programmer's Library Reference Manual*
SR–2081	*Scientific Libraries Reference Manual*
SR–2091	*UNICOS CDBX Symbolic Debugger Reference Manual*
SG–2094	*UNICOS CDBX Debugger User's Guide*
SR–2138	*Math Library Reference Manual*
IN–2140	*Introducing the Program Browser*
SG–2182	*Guide to Parallel Vector Applications*
IN–2502	*Introducing the TotalView Debugger*
IN–2511	*Introducing the MPP Apprentice Tool*
SN–3009	*Interlanguage Programming Conventions*
SG–2168	*Application Programmer's I/O Guide*

Publication	Title
SG–2192	*Optimizing Code on Cray PVP Systems*
SR–2401	*Compiler Information File (CIF) Reference Manual*
SG–3094	*UNICOS Visual Interfaces User's Guide*

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.

To order a 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 electronic mail to `orderdsk` (UNIX system users).

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

If you are a CraySoft customer, you can order additional copies of this product, increase the number of licensed users, or order CraySoft publications, by contacting the CraySoft organization in one of the following ways:

- Call one of these telephone numbers:

  +1–800–289–2729

  +1–612–683–3030

- Send mail to the following address:

  CraySoft
Cray Research, Inc.
1440 Northland Drive
Mendota Heights, Minnesota 55120
USA

- Send electronic mail from any system connected to Internet to the following address:

  `craysoft@cray.com`

- Send a facsimile (fax) to the following telephone number:

  +1–612–683–5508

Conventions

This manual describes Fortran statement syntax using a variant of Backus-Naur Form (BNF). The following conventions are used throughout this manual when describing the Fortran statements and other parts of the Fortran language:

Convention	Meaning
`command`	This font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures. `Typewriter` font is also used to represent terminal symbols in parts of BNF.
variable	This typeface denotes variable entries and words or concepts being defined.
`user input`	This font denotes literal items that the user enters in screen drawings of interactive sessions. Output is shown in nonbold `typewriter` font.
[ ]	Brackets enclose optional portions of a command line or syntax statement. The items inside the brackets may be repeated zero or more times.
_list ...	Both *_list* and an ellipses indicate that the preceding item can be repeated. Use of *_list* implies that the members of the list are separated from each other by commas.

Convention	Meaning
*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 *CF90 Fortran 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.

In this publication, *Cray Research*, *Cray*, and *CRI* refer to Cray Research, Inc. and/or its products.

BNF conventions

This subsection describes some of the commonly used 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** `GO TO` *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 not complete 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 does not need further definition. For example, `GO TO` is a terminal and is a required part of the statement form. The complete BNF list appears in the *CF90 Fortran Language Reference Manual, Volume 3*, publication SR–3905.

- Several abbreviations are commonly used in naming nonterminals. These are as follows:

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

add_op	**is**	+
	or	−

- The ≡ symbol indicates a continuation line. If a rule does not fit on one line, the convention is to use the symbol ≡ at the end of the line being continued as well as at the beginning of the line that continues or completes the statement. This is shown in the following example:

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

Online information

The following types of online information products are available to Cray Research customers:

- CrayDoc online documentation reader, which lets you see the text and graphics of a manual online. The CrayDoc reader is available on workstations. To start the CrayDoc reader at your workstation, use the `cdoc`(1) command.
- Man pages, which describe a particular element of the operating system or a compatible product. To see a detailed description of a particular command or routine, use the `man`(1) command.
- The message system, which provides explanations of error messages. To see an explanation of a message, use the `explain`(1) command.
- Cray Research online glossary, which explains the terms used in a manual. To get a definition, use the `define`(1) command. This command is supported on Cray PVP systems and CRAY T3E systems.
- `xhelp` help facility. This online help system is available within tools such as the Program Browser (`xbrowse`) and the Cray MPP Apprentice tool.

For detailed information on these topics, see the *User's Guide to Online Information*, publication SG–2143.

Reader comments

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

- Send us electronic mail from a UNICOS or UNIX system, using the following UUCP address:

  `uunet!cray!publications`

- Send us electronic mail from any system connected to the Internet, using the following Internet address:

  `publications@timbuk.cray.com` (general comments)

- Contact your Cray Research representative and ask that a Software Problem Report (SPR) be filed. Use `PUBLICATIONS` for the group name, `PUBS` for the command, and `NO-LICENSE` for the release name.

- CraySoft customers in the United States and Canada can call the CraySoft Customer Support Call Center at either of the following telephone numbers. These telephone numbers are also listed on the `cs_support`(7) man page and on the Service and Support card included in the product package.

  +1–800–272–9349 (toll free from the United States and Canada)

  +1–770–631–2244

- CraySoft customers outside the United States and Canada can call their local Cray Research support office. These telephone numbers are listed on the `cs_support`(7) man page and on the Service and Support card contained in the product package.

- Call our Software Publications Group in Eagan, Minnesota, through the Technical Support 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.

- Use the postage-paid Reader's Comment Form at the back of the printed manual.

We value your comments and will respond to them promptly.

Contents

Figures

Table

Program Units [1]

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 section describes each of these units as well as the closely related concepts of host association and use association.

Overview

1.1

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 *CF90 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 1 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
```

Figure 1. 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`.

Main program

1.2

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

Main program organization

1.2.1

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*)]
CRI	*arg*	**is**	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.

The specification part
1.2.2

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 *CF90 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.
- `COMMON`.
- `DATA`.
- `DIMENSION`.
- `EQUIVALENCE`.
- `EXTERNAL`.
- `FORMAT`.
- `IMPLICIT`.
- `INTRINSIC`.
- `NAMELIST`. The `NAMELIST` statement was first included in the CF77 compiling system as an extension, but it is part of the Fortran 90 standard.
- `PARAMETER`.
- `POINTER`. The `POINTER` attribute and statement, which are new in Fortran 90, specify dynamic objects. The Cray pointer is a Cray Research 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.

Execution part
1.2.3

The complete list of execution part statements (R208) is given in the *CF90 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 Cray Research 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. Some characteristics of the `DO` construct, such as the `END DO` statement, were included in the CF77 compiling system as extensions to the FORTRAN 77 standard.
- `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. These are new in Fortran 90, but they were included in the CF77 compiling system as extensions to the FORTRAN 77 standard.
- `WRITE`.

The internal subprogram part

1.2.4

A set of internal procedures comprises the internal subprogram part. Internal procedures are described in the following subsection.

Internal procedures

1.3

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 subsection 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).

Host association

1.4

The program unit containing an internal procedure is called the *host* of the internal procedure. The program unit (which must be a module) containing a module procedure is called the *host* of the module procedure. As shown in Figure 2, 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.

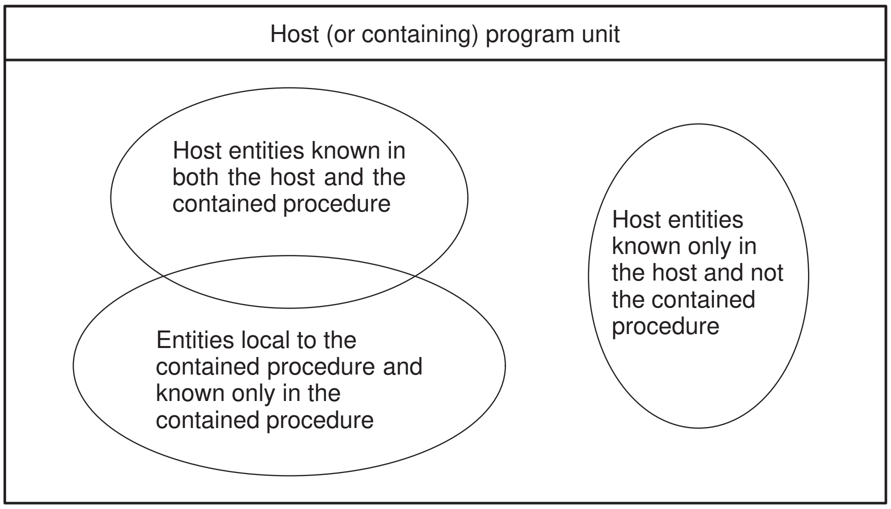
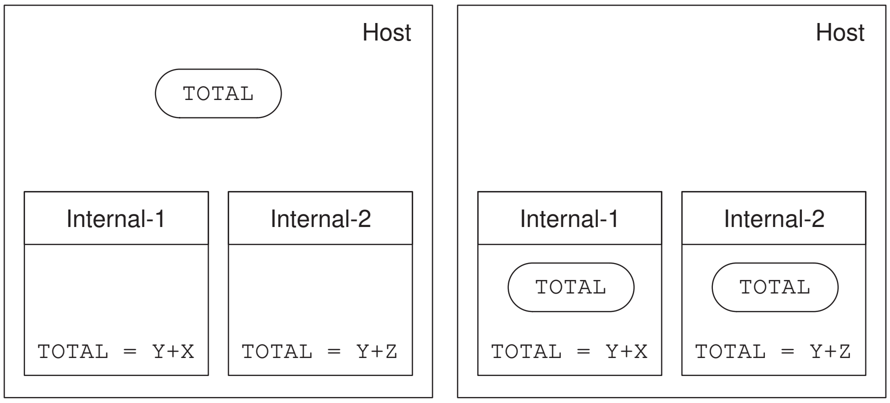


Figure 2. Host association

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 3.



(a) a single host TOTAL (b) two local TOTALs

Figure 3. Is there one TOTAL in the host or two local TOTALs?

The answer to both of these questions is case (b) in Figure 3, 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 *CF90 Fortran Language Reference Manual*, *Volume 1*, publication SR–3902, and the rules governing host association are given in subsection 4.2.1.3, page 438. 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 4, page 15).

```
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
```

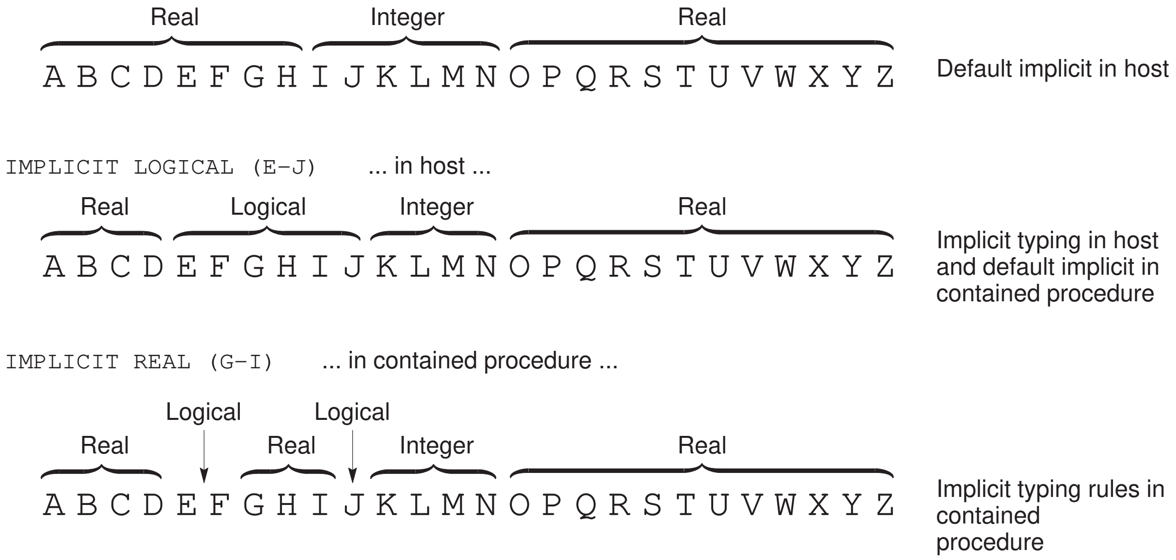


Figure 4. How the mapping of implicit typing progresses from host to contained procedure

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.

External subprograms

1.5

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; subsection 2.6, page 90, 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 subsection 2.3.1, page 52. For more information on the `SUBROUTINE` statement, see subsection 2.2.1, page 47.

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 2, page 39.

Modules

1.6

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.

Module organization

1.6.1

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.

The specification part
1.6.2

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 subsection.

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
```

Module subprogram part

1.6.3

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 subsection 1.4, page 11. 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.

Using modules
1.6.4

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 5 depicts these phenomena.

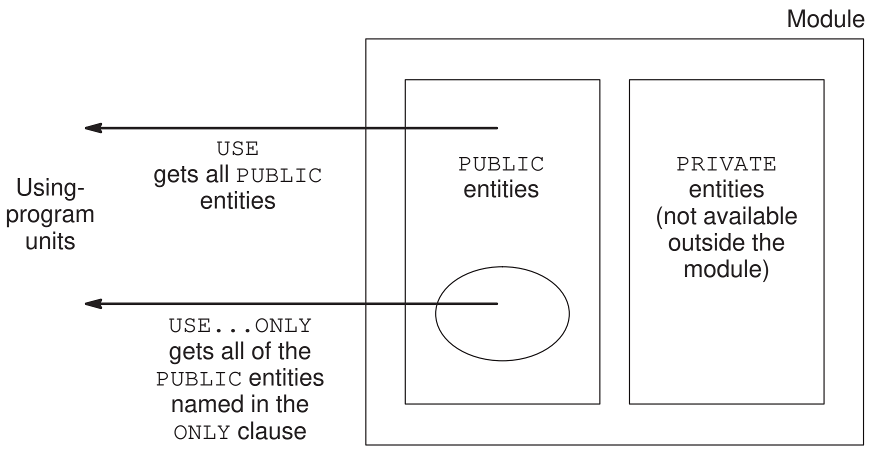


Figure 5. Public and private entities in a module

Accessing all public entities in a module 1.6.4.1

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.

Accessing only part of the public entities 1.6.4.2

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
```

1.6.4.3 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 6 illustrates these rules.

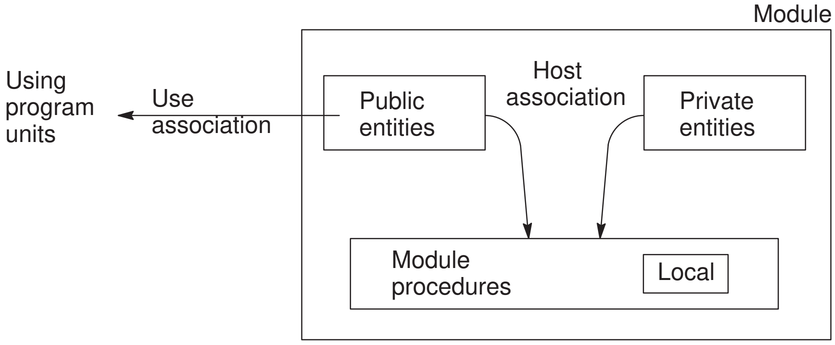


Figure 6. Use of public and private module entities

1.6.4.4 Name conflicts when using modules

When using modules, name conflicts can occur in the following two ways:

1. A public entity in a module might have the same name as a local entity in the using program.
2. 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
```

Use association 1.6.4.5

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.

Typical applications of modules
1.6.5

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 subsections.

Global data
1.6.5.1

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.

```
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
```

COMMON blocks in a module

1.6.5.2

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.

```
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.

Global user-defined types

1.6.5.3

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.

```
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.

Operator extensions
1.6.5.4

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 subsection 2.6, page 90. 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 (assuming + also has been extended to add a real value and a MATRIX):

```
1.0 + (.INVERSE. A)
```

Data abstraction
1.6.5.5

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.

Procedure libraries

1.6.5.6

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.

```
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
```

Independent compilation

1.6.6

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.

Block data program units

1.7

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 compiler allows 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 compiler permits 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 [2]

Procedures help to structure a problem solution into understandable segments. This section describes the details for constructing and using procedures in Fortran 90.

Procedure terms and concepts

2.1

This subsection describes some of the basic terms and concepts associated with Fortran procedures.

Procedure terms

2.1.1

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.

Subroutines

2.1.1.1

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.

Functions

2.1.1.2

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.

Function results
2.1.1.3

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.

External procedures
2.1.1.4

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.

Intrinsic procedures
2.1.1.5

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 compiler includes several additional procedures as extensions. These procedures are described in section 3, page 107.

Section 2

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 of the 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 .

Internal procedures
2.1.1.6

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.

Module procedures
2.1.1.7

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.

Statement functions
2.1.1.8

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.

2.1.1.9 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.

2.1.1.10 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 section, 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 stand-alone 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.

Actual arguments
2.1.1.11

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.

Dummy arguments
2.1.1.12

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.

Alternate return
2.1.1.13

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.

Dummy procedures
2.1.1.14

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.

Non-Fortran procedures
2.1.1.15

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.

Argument association
2.1.2

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 subsection 2.5.1, page 69.

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 subsection 2.5, page 67.

Recursion
2.1.3

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.

Host and use association
2.1.4

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 later in this section. Common blocks, extended with data structures, are the same as in FORTRAN 77 and are described in the *CF90 Fortran Language Reference Manual, Volume 1*, publication SR–3902. Host and use association are new in Fortran 90 and are described in subsection 1.4, page 11, and subsection 1.6.4, page 23.

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 subsection 1.4, page 11, 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 subsection 1.6.4.5, page 28, for a complete description of use association.

Implicit and explicit interfaces

2.1.5

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 section.

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.

Subroutines

2.2

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.

Subroutine definition
2.2.1

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**	[RECURSIVE] SUBROUTINE *subroutine_name* ≡ ≡ [([*dummy_arg_list*])]
R1221	*dummy_arg*	**is** **or**	*dummy_arg_name* *
R1222	*end_subroutine_stmt*	**is**	END [SUBROUTINE [*subroutine_name*]]

A dummy argument 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 subsection 2.5, page 67, 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.

If the subroutine is recursive, that is, it calls itself either directly or indirectly , the keyword RECURSIVE is required in the SUBROUTINE statement.

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.

2.2.2 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 subsection 2.5, page 67, 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*; variables 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 man pages in section 3, page 107, 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 subsection 2.5.4, page 81. 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.

Functions
2.3

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.

Function definition
2.3.1

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**	*type_spec* [`RECURSIVE`]
		or	`RECURSIVE` [*type_spec*]
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 subsection 2.5, page 67, 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 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 subsection 4.1, page 424.

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:

```
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.

The RESULT option
2.3.2

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
```

Function reference
2.3.3

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 subsection 2.5, page 67; and the statements in the body of the function are executed. The reference results in a value which 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 subsection 2.2.2, page 49.

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 subsection 2.5.3, page 78. 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 *CF90 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.

```
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
```

Statement functions
2.3.4

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 subsection 2.5.1, page 69, 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.

Section 2

Procedure-related statements

2.4

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 subsections describe these statements.

`RETURN` statement

2.4.1

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.

CONTAINS statement
2.4.2

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.

ENTRY statement
2.4.3

Subsection 2.1.1, page 39, 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 a *result_name* is specified, it must not be the same as any entry name, the function name, or any other *result_name*. If a *result_name* is specified, the *entry_name* must not appear in any specification statements in the subprogram; it inherits all of its attributes from the *result_name*.

8. The keyword RECURSIVE is not used in an ENTRY statement. Instead the presence or absence of RECURSIVE 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.

2.4.4 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.

`INTRINSIC` statement
2.4.5

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)
```

Argument association
2.5

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 7, 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 7, 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.

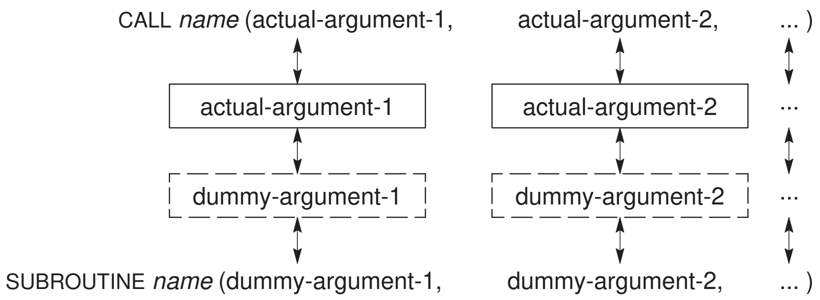


Figure 7. Actual and dummy argument lists

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 section describes the rules governing this association mechanism and the forms it can take.

Type, kind, and rank matching

2.5.1

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 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
```

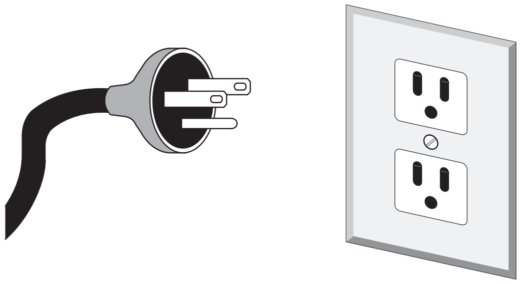
The TKR pattern of `IRIR` is: integer, nondefault kind, rank 2.

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 8), 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.



Actual arguments Dummy arguments

Figure 8. 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 subsection 2.6.1, page 91) 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 subsection 2.5.2, page 71, 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 subsection 2.5.2, page 71, and subsection 2.5.3, page 78, 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 subsection 2.5.3, page 78, 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 subsection 2.5.2.1, page 72.

Sequence association
2.5.2

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 subsection 2.5.2.1.

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 section 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 subsection 2.5.2.4, page 77.

2.5.2.1 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 9, 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)`

$AA_{1,1,1}$	$AA_{2,1,1}$	$AA_{1,2,1}$	$AA_{2,2,1}$	$AA_{1,3,1}$	$AA_{2,3,1}$	$AA_{1,1,2}$	$AA_{2,1,2}$	$AA_{1,2,2}$	$AA_{2,2,2}$	$AA_{1,3,2}$	$AA_{2,3,2}$

$DA_{1,1}$	$DA_{2,1}$	$DA_{3,1}$	$DA_{4,1}$	$DA_{5,1}$	$DA_{1,2}$	$DA_{2,2}$	$DA_{3,2}$	$DA_{4,2}$	$DA_{5,2}$

Dummy array: `REAL DA(5,2)`

Figure 9. 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 10. 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 10, 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)`.

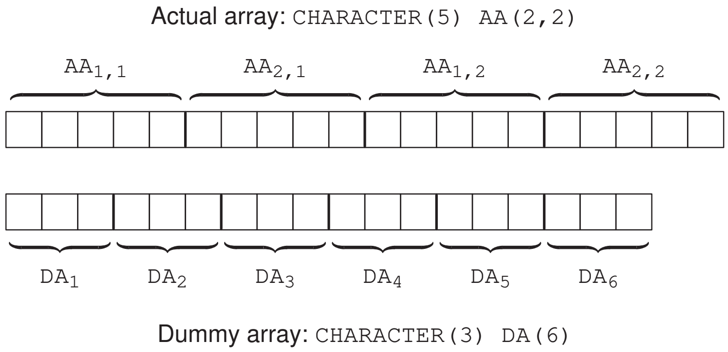


Figure 10. Array element sequence association for default characters

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 .

Passing array sections
2.5.2.2

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.

Miscellaneous array association rules 2.5.2.3

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 subsection).

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.

Structure sequence association 2.5.2.4

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.

Pointer association
2.5.3

In general, 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 11 are distinct (the other two are equivalent to the cases below them in the figure). AESA is an abbreviation for Array Element Sequence Association.

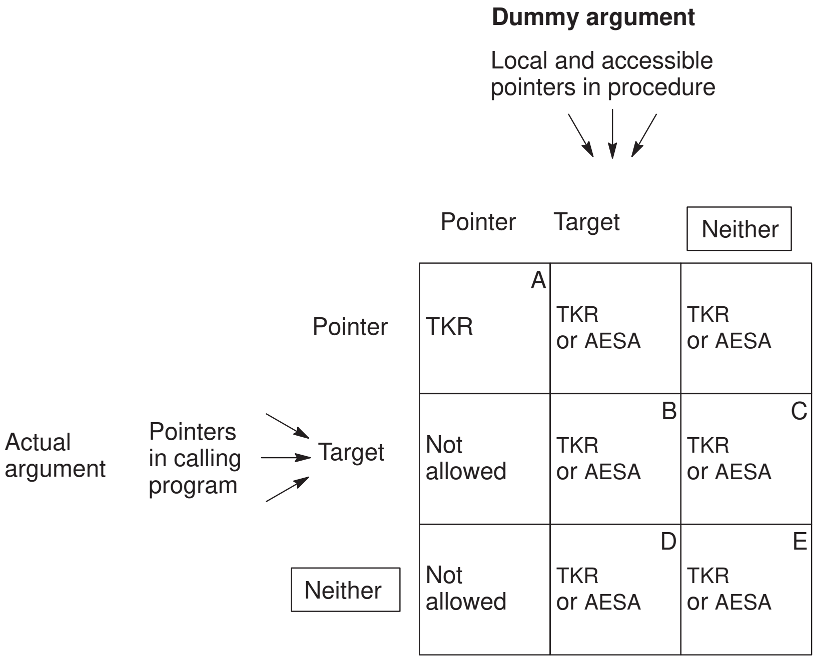


Figure 11. Association of objects with `POINTER` and `TARGET` attributes

In combination `E` in Figure 11, neither the actual or dummy argument has either the `POINTER` or `TARGET` attribute; this situation is explained in the previous two subsections. 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 11 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 11, page 79, 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 11, page 79, 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 subsection 2.6.1, page 91), 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 11, page 79, 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 11, page 79, 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.

Argument keywords
2.5.4

The fundamental pairing of arguments in the actual argument list with those in the dummy argument list is positional, as shown in Figure 7, page 68. 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.

Optional arguments
2.5.5

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.

Argument intent
2.5.6

Any dummy argument, except a procedure or pointer, can be given an `INTENT` attribute (see the *CF90 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.

2.5.7 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 subsection 4.1.7, page 431. 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. Subsection 2.6.3, page 99, 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 subsection 2.6.3, page 99.

Elemental references to intrinsic procedures
2.5.8

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.

Alternate returns
2.5.9

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.

Dummy procedures
2.5.10

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 subsection 2.6.2.2, page 98.

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.

Procedure interfaces

2.6

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 subsections 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.

Explicit interfaces 2.6.1

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 indeed 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 compiler packs such sections into contiguous temporaries on entry to the procedure and unpacks them on return, incurring performance penalties.

Item 6 is on the list 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.

For generic procedures (item 8), 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 subsection 2.6.3, page 99.

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 subsection 2.6.4, page 101.

A third form of generic interface for which explicit interface information is used to resolve references is that of assignment coercion (item 10). 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 subsection 2.6.5, page 104.

2.6.2 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 four items listed above, the first is further described in this subsection. The others were described previously in this section. In the remainder of this section, 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.

2.6.2.1 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 subsection 2.6.2, page 94. 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.

2.6.2.2 Explicit interfaces for external procedures

The simplest use of an interface block is to make the interface for 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 subsection 2.6.2.1, page 95, 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.

Generic procedures

2.6.3

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 subsection 2.6.2.1, page 95, 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 subsection 4.1.7, page 431.

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.

Defined operators

2.6.4

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 subsection 1.6.5.4, page 32. 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} *x* B, 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 Cray Research 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 subsection), 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.

Defined assignment
2.6.5

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 [3]

The Fortran 90 standard defines intrinsic procedures, which consist of intrinsic functions and intrinsic subroutines. The CF90 compiler includes 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 section introduces all of the intrinsic procedures that the CF90 compiler supports. Each procedure is described in detail on an online man page. For more information on any of the intrinsic procedures introduced in this section, enter `man` *intrinsic_name* at your system prompt.

Section 3

Intrinsic procedure terms and concepts

3.1

Intrinsic procedures are predefined by the language. They conform to the principles and rules for procedures as described in the *CF90 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 compiler also includes the following types of intrinsic procedures:

- Intrinsic procedures to support IEEE floating-point arithmetic
- Additional CF90 compiler extensions

Inquiry functions
3.1.1

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 subsection 3.3, page 123.

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

Name	Information returned
`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

Elemental functions
3.1.2

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 that Cray Research has added as CF90 compiler extensions to the Fortran 90 standard are elemental functions. This subsection contains two lists. The first list summarizes the standard elemental functions; in this list, the numeric model referred to is described in subsection 3.3, page 123. 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

Name	Information returned or operation
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

Name	Information returned or operation
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

Name	Information returned or operation
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

Name	Information returned or operation
`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 compiler intrinsic procedures that are extensions to the Fortran 90 standard and are elemental functions are as follows:

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 Cray words successfully transferred
`POPCNT`	Counts number of bits set to 1

Section 3

Name	Information returned or operation
POPPAR	Computes bit population parity
UNIT	Performs I/O functions

Transformational functions
3.1.3

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

Name	Information returned or operation
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

Subroutines
3.1.4

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.

The Fortran 90 standard subroutines are as follows:

Name	Purpose
DATE_AND_TIME	Obtains date and time
MVBITS	Copies bits from one bit manipulation model integer to another
RANDOM_NUMBER	Returns pseudorandom number

Name	Purpose
RANDOM_SEED	Initializes or restarts the pseudorandom number generator
SYSTEM_CLOCK	Obtains data from the system clock

3.1.5 Intrinsic procedures to support IEEE floating-point arithmetic (CF90 compiler extensions)

The intrinsic procedures described in this subsection support the IEEE Standard for Binary Floating-point Arithmetic. They are supported on CRAY T3E systems, SPARC systems, and CRAY T90 systems that support IEEE floating-point arithmetic.

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

Name	Information returned or operation
`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

Name	Information returned or operation
`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.

Using named constants
3.1.5.1

A number of the procedures named in subsection 3.1.5, page 116, use named constants. The named constants are provided in module `CRI_IEEE_DEFINITIONS`. To use this module, include the following statement in your program:

```
USE CRI_IEEE_DEFINITIONS
```

Note: The `CRI_IEEE_DEFINITIONS` module must be included 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, Inc. reserves the right to change the values represented by the named constants.

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 `CRI_IEEE_DEFINITIONS`. The following paragraphs describe the named constants that are available to you and the systems upon which they are supported.

These named constants allow you to access floating-point exception flags:

Constant	Systems
`IEEE_XPTN_CRI_INVALID_OPND`	CRAY T90 (IEEE)
`IEEE_XPTN_INEXACT_RESULT`	CRAY T90 (IEEE), CRAY T3E, SPARC
`IEEE_XPTN_UNDERFLOW`	CRAY T90 (IEEE), CRAY T3E, SPARC
`IEEE_XPTN_OVERFLOW`	CRAY T90 (IEEE), CRAY T3E, SPARC
`IEEE_XPTN_DIV_BY_ZERO`	CRAY T90 (IEEE), CRAY T3E, SPARC
`IEEE_XPTN_INVALID_OPR`	CRAY T90 (IEEE), CRAY T3E, SPARC
`IEEE_XPTN_ALL`	CRAY T90 (IEEE), CRAY T3E, SPARC

These named constants allow you to access floating-point interrupt flags:

Constant	Systems
`IEEE_NTPT_CRI_INVALID_OPND`	CRAY T90 (IEEE)
`IEEE_NTPT_INEXACT_RESULT`	CRAY T90 (IEEE), CRAY T3E, SPARC
`IEEE_NTPT_UNDERFLOW`	CRAY T90 (IEEE), CRAY T3E, SPARC
`IEEE_NTPT_OVERFLOW`	CRAY T90 (IEEE), CRAY T3E, SPARC
`IEEE_NTPT_DIV_BY_ZERO`	CRAY T90 (IEEE), CRAY T3E, SPARC
`IEEE_NTPT_INVALID_OPR`	CRAY T90 (IEEE), CRAY T3E, SPARC
`IEEE_NTPT_ALL`	CRAY T90 (IEEE), CRAY T3E, SPARC

These named constants allow you to access the floating-point rounding mode:

Constant	Systems
`IEEE_RM_NEAREST`	CRAY T90 (IEEE), CRAY T3E, SPARC
`IEEE_RM_POS_INFINITY`	CRAY T90 (IEEE), CRAY T3E, SPARC
`IEEE_RM_ZERO`	CRAY T90 (IEEE), CRAY T3E, SPARC
`IEEE_RM_NEG_INFINITY`	CRAY T90 (IEEE), CRAY T3E, SPARC

These named constants allow you to determine the class of a floating-point value:

Constant	Systems
IEEE_CLASS_SIGNALING_NAN	CRAY T90 (IEEE), CRAY T3E, SPARC
IEEE_CLASS_QUIET_NAN	CRAY T90 (IEEE), CRAY T3E, SPARC
IEEE_CLASS_NEG_INFINITY	CRAY T90 (IEEE), CRAY T3E, SPARC
IEEE_CLASS_NEG_NORM_NONZERO	CRAY T90 (IEEE), CRAY T3E, SPARC
IEEE_CLASS_NEG_DENORM	CRAY T90 (IEEE), CRAY T3E, SPARC
IEEE_CLASS_NEG_ZERO	CRAY T90 (IEEE), CRAY T3E, SPARC
IEEE_CLASS_POS_ZERO	CRAY T90 (IEEE), CRAY T3E, SPARC
IEEE_CLASS_POS_DENORM	CRAY T90 (IEEE), CRAY T3E, SPARC
IEEE_CLASS_POS_NORM_NONZERO	CRAY T90 (IEEE), CRAY T3E, SPARC
IEEE_CLASS_POS_INFINITY	CRAY T90 (IEEE), CRAY T3E, SPARC

Section 3

3.1.5.2 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 exisiting 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
```

Miscellaneous procedures (CF90 compiler extensions)

3.1.6

Most of the CF90 compiler intrinsic functions that are extensions to the Fortran 90 standard are elemental intrinsic functions. 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
`MY_PE`	Returns the number of the processing element (PE) that is executing

Section 3

Name	Information returned or operation
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

Argument keywords

3.2

Intrinsic procedure references can use keyword arguments, as described in subsection 2.5.4, page 81. 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.

Representation models

3.3

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 compiler running on any platform. The values that go into these models, however, are platform-specific. Please see the `models`(3) man page information on how these models apply to the various platforms that the CF90 compiler supports.

Intrinsic procedure summary

3.4

This subsection 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 compiler . 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 compiler supports 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.

Table 1. CF90 intrinsic procedures

Procedure name	Arguments
`ABS`	`([A=]a)`
`IABS`	`([A=]`*a*`)`
`DABS`	`([A=]`*a*`)`
`CABS`	`([A=]`*a*`)`
`CDABS`	`([A=]`*a*`)`

Table 1. CF90 intrinsic procedures (continued)

Procedure name	Arguments
`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*`)`

Section 3

Table 1. CF90 intrinsic procedures (continued)

Procedure name	Arguments
BIT_SIZE	([I=]*i*)
BTEST	([I=]*i*, [POS=]*pos*)
CEILING	([A=]*a*)
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*)

Section 3

Table 1. CF90 intrinsic procedures (continued)

Procedure name	Arguments
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*)
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*)

Table 1. CF90 intrinsic procedures (continued)

Procedure name	Arguments
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*)
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*)

Table 1. CF90 intrinsic procedures (continued)

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*])

Table 1. CF90 intrinsic procedures (continued)

Procedure name	Arguments
INT	([A=]*a* [,[KIND=]*kind*])
IFIX	([A=]*a* [,[KIND=]*kind*])
IDINT	([A=]*a* [,[KIND=]*kind*])
INT_MULT_UPPER	([I=]*i*, [J=]*j*)
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*)
LEADZ	([I=]*i*)
LBOUND	([ARRAY=]*array* [,[DIM=]*dim*])
LEN	([STRING=]*string*)
LENGTH	([I=]*i*)
LEN_TRIM	([STRING=]*string*)
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*)

Table 1. CF90 intrinsic procedures (continued)

Procedure name	Arguments
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	()
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*] ...)

Table 1. CF90 intrinsic procedures (continued)

Procedure name	Arguments
MAXEXPONENT	([X=]*x*)
MAXLOC	([ARRAY=]*array* [,[MASK=]*mask*])
MAXVAL	([ARRAY=]*array* [,[DIM=]*dim*] [,[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* [,[MASK=]*mask*])
MINVAL	([ARRAY=]*array* [,[DIM=]*dim*] [,[MASK=]*mask*])
MOD	([A=]*a*, [P=]*p*)
AMOD	([A=]*a*, [P=]*p*)
DMOD	([A=]*a*, [P=]*p*)
MODULO	([A=]*a*, [P=]*p*)
MY_PE	()
MVBITS	([FROM=]*from* , [FROMPOS=]*frompos* , [LEN=]*len* , [TO=]*to*, [TOPOS=]*topos*)
NEAREST	([X=]*x*, [S=]*s*)

Section 3

Table 1. CF90 intrinsic procedures (continued)

Procedure name	Arguments
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*)
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*])
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*)

Section 3

Table 1. CF90 intrinsic procedures (continued)

Procedure name	Arguments
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*)
SET_IEEE_INTERRUPTS	([STATUS=]*status*)
SET_IEEE_ROUNDING_MODE	([ROUNDING_MODE=]*rounding_mode*)
SET_IEEE_STATUS	([STATUS=]*status*)
SHAPE	([SOURCE=]*source*)

Section 3

Table 1. CF90 intrinsic procedures (continued)

Procedure name	Arguments
`SHIFT`	`([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* `])`
`SYSTEM_CLOCK`	`([[COUNT=]`*count* `] [,[COUNT_RATE=]`*count_rate* `] [,[COUNT_MAX=]`*count_max* `])`
`TAN`	`([X=]`*x*`)`
`DTAN`	`([X=]`*x*`)`

Section 3

Table 1. CF90 intrinsic procedures (continued)

Procedure name	Arguments
TANH	([X=]*x*)
DTANH	([X=]*x*)
TEST_IEEE_EXCEPTION	([EXCEPTION=]*exception*)
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	()

Man pages

3.5

The rest of this section contains copies of the man pages that describe the CF90 intrinsic procedures. On SPARC systems, the underlying math library differs from that on Cray PVP systems and CRAY T3E systems. Please see your math library documentation for information on the intrinsic procedures implemented as part of the math library.

These man pages can be accessed online through the man(1) command.

Scope, Association, and Definition [4]

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 section 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 *CF90 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.

Scope

4.1

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 12. 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.

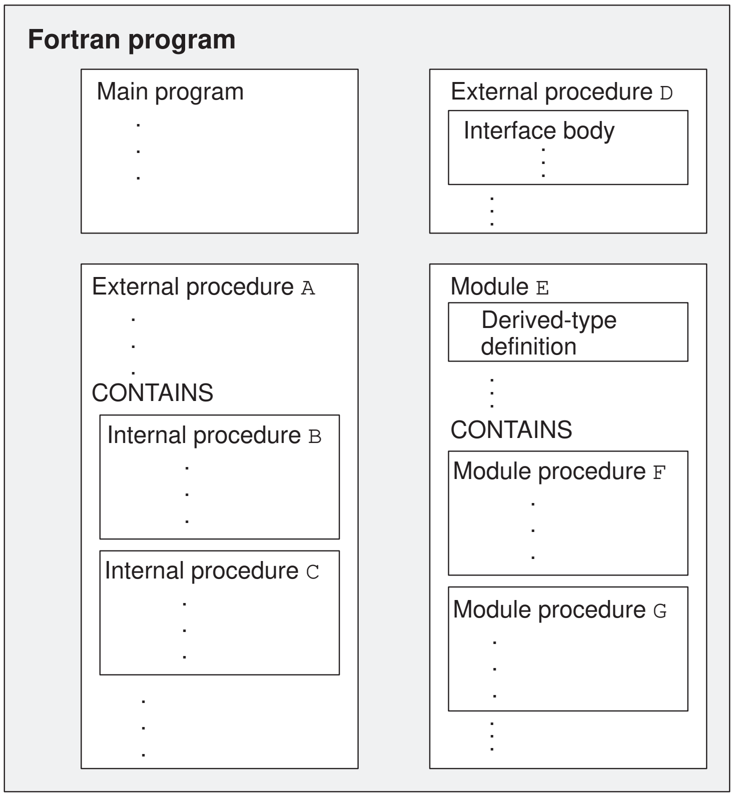


Figure 12. Scoping units

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.

Scope of names

4.1.1

A name has one of the following scopes:

- *Global* , which is the scope of an executable program (for example, an external function name)
- *Local* , which is the scope of a scoping unit (for example, an array name in a subroutine subprogram)
- *Statement* , 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)

Names as global entities

4.1.1.1

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.

Names as local entities

4.1.1.2

There are three classes of names having 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.

Section 4

- 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.

Names as statement entities
4.1.1.3

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. Similarily, 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.

Section 4

Scope of labels
4.1.2

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.

Scope of input/output (I/O) units
4.1.3

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.

Scope of operators
4.1.4

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.

Scope of assignment
4.1.5

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.

Unambiguous procedure references
4.1.6

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.

Resolving procedure references

4.1.7

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. Subsections 2.5.7, page 86, 2.6.5, page 104, and 4.1.6, page 430, 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. Subsections 2.5.7, page 86, 2.6.4, page 101, and 4.1.6, page 430, 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. Subsection 2.5, page 67, 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.

Association

4.2

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.

Association is the mechanism 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 13 and Figure 14 illustrate the various kinds of association in an executable program.

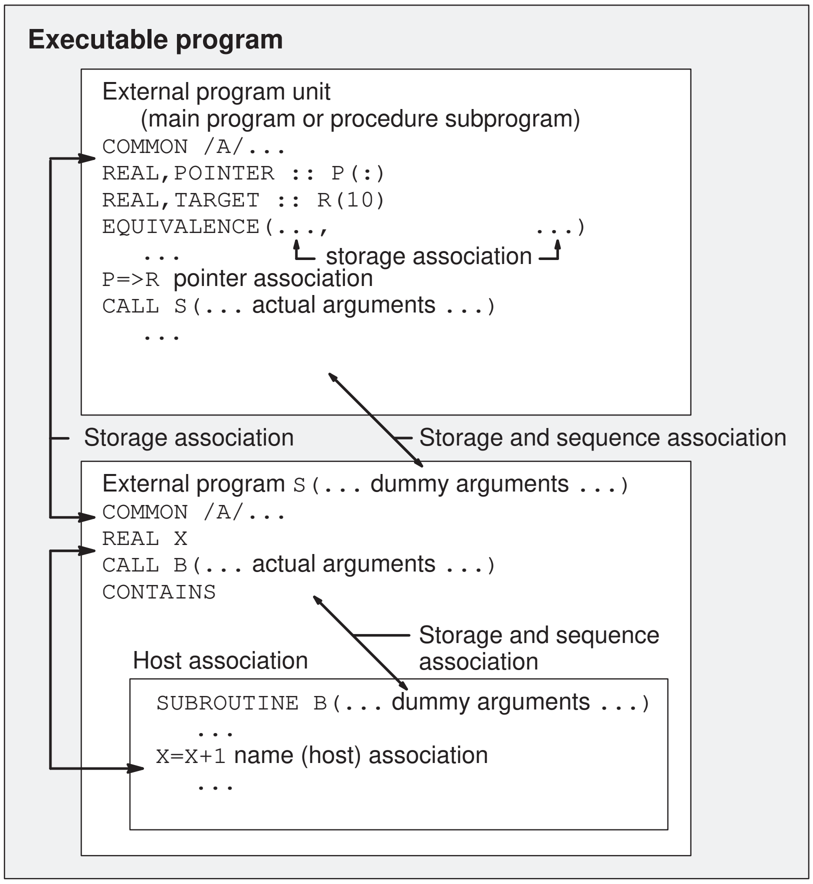


Figure 13. Associations between two nonmodule scoping units

Section 4

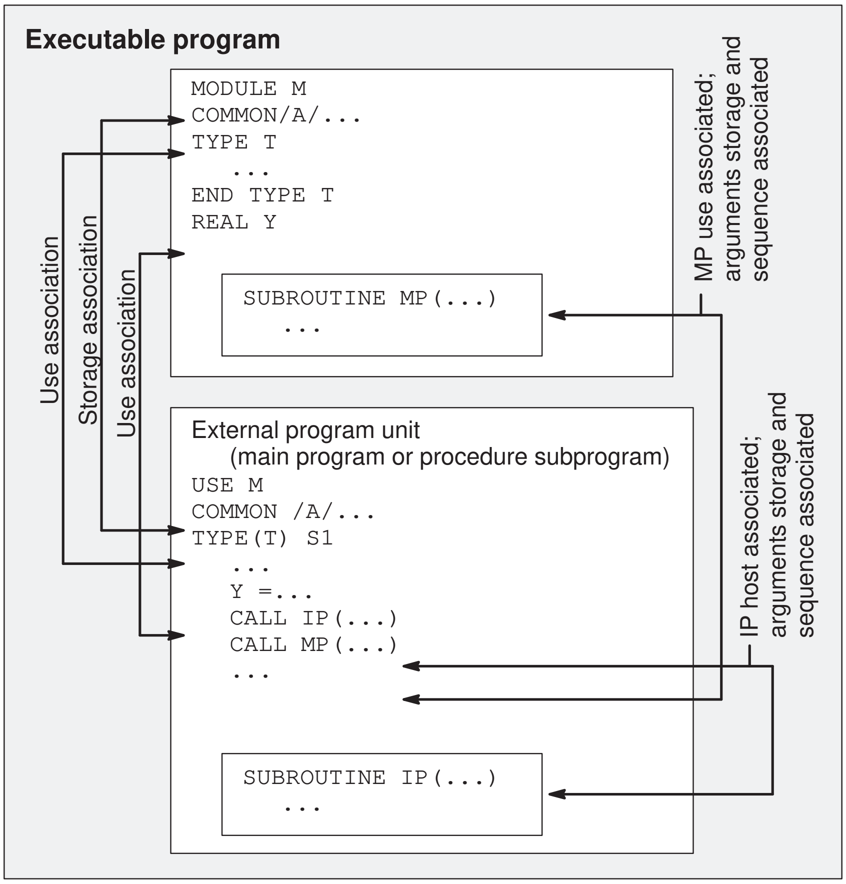


Figure 14. Associations between a module scoping unit and a nonmodule scoping unit

Name association
4.2.1

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.

Section 4

Argument association
4.2.1.1

Argument association is explained in detail in subsection 2.5, page 67. 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.

Use association
4.2.1.2

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 subsection 1.6.4.5, page 28.

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 subsection 1.6.4.5, page 28, for an exception to this).

Host association
4.2.1.3

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 subsection 1.4, page 11.

Pointer association
4.2.2

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).

Section 4

Storage association
4.2.3

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.

Storage units and storage sequence
4.2.3.1

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.

Fortran data and storage units
4.2.3.2

This subsection 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 *CF90 Fortran Language Reference Manual, Volume 1*, publication SR–3902.
- Each `ENTRY` statement in a function subprogram has a storage sequence as described in subsection 2.4.3, page 62.
- `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.*

Section 4

Partial association
4.2.3.3

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.

Examples 4.2.3.4

This subsection 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):

X(0)	X(1)	X(2)	X(3)	X(4)	X(5)
			C*r*	C*i*	

Figure 15. 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:

`A(1,1)(1:1)`	`A(1,1)(2:2)`	`A(2,1)(1:1)`	`A(2,1)(2:2)`	`A(1,2)(1:1)`	`A(1,2)(2:2)`	`A(2,2)(1:1)`	`A(2,2)(2:2)`
	`B(1)(1:1)`	`B(1)(2:2)`	`B(1)(3:3)`	`B(2)(1:1)`	`B(2)(2:2)`	`B(2)(3:3)`	
`C(1:1)`	`C(2:2)`	`C(3:3)`	`C(4:4)`	`C(5:5)`			

Figure 16. Example 2

Sequence association

4.2.4

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 subsection 2.5.2, page 71.

Definition status

4.3

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.

Definition status of subobjects
4.3.1

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.

Events that affect definition status of variables
4.3.2

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 subsections.

Events that cause variables to become defined
4.3.3

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.

Section 4

4.3.4 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

    ANSI/ISO: 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 compiler retains 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.

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.

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.

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. See also *regular expression*.

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.

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.

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 share 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 .

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 a 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`.

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.

regular expression

(1) In POSIX, a pattern (sequence of characters or symbols) constructed according to the defined rules or defined RE rules (see *regular expression rules* in the online glossary). (2) A character string that contains operating system wildcards (metacharacters) that can match multiple substitution characters. Such a string will match more than one literal string; it is used to filter a specified string or type of string from an input of multiple strings. The main editors in UNIX, `ed` and `vi`, use regular expressions.

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.

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.

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.

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`).

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

An optimization technique used on vectorizing machines in which a vector or array is broken down into chunks the size of the machine's vector register set for simultaneous processing.

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.

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, a 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

A

B

C

D

J

K

L

M

N

O

P

U

V

W

X

Reader's Comment Form

CF90 Fortran Language Reference Manual, Volume 2 SR–3903 2.0

Your reactions to this manual will help us provide you with better documentation. Please take a moment to complete the following items, and use the blank space for additional comments.

List the operating systems and programming languages you have used and the years of experience with each.

Your experience with computer systems: _____0-1 year _____1-5 year _____5+years

How did you use this manual: _____in a class _____as a tutorial or introduction _____as a procedural guide _____as a reference _____for troubleshooting _____other

Please rate this manual on the following criteria:

	Excellent			Poor
Accuracy	4	3	2	1
Appropriateness (correct technical level)	4	3	2	1
Accessibility (ease of finding information)	4	3	2	1
Physical qualities (binding, printing, illustrations)	4	3	2	1
Terminology (correct, consistent, and clear)	4	3	2	1
Number of examples	4	3	2	1
Quality of examples	4	3	2	1
Index	4	3	2	1

Please use the space below for your comments about this manual. Please include general comments about the usefulness of this manual. If you have discovered inaccuracies or omissions, please specify the number of the page on which the problem occurred.

Name ______________________
Title ______________________
Company ______________________
Telephone ______________________
Today's date ______________________

Address ______________________
City ______________________
State/Country ______________________
Zip code ______________________
Electronic mail address ______________________

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO 6184 ST. PAUL, MN

POSTAGE WILL BE PAID BY ADDRESSEE

CRAY
RESEARCH, INC.

ATTN: Software Publications Group
655 LONE OAK DR BLDG F
EAGAN MN 55121-9957

Fold

Cut along this line

Reader's Comment Form

CF90 Fortran Language Reference Manual, Volume 2 SR–3903 2.0

Your reactions to this manual will help us provide you with better documentation. Please take a moment to complete the following items, and use the blank space for additional comments.

List the operating systems and programming languages you have used and the years of experience with each.

Your experience with computer systems: _____0-1 year _____1-5 year _____5+years

How did you use this manual: _____in a class _____as a tutorial or introduction _____as a procedural guide _____as a reference _____for troubleshooting _____other

Please rate this manual on the following criteria:

	Excellent			Poor
Accuracy	4	3	2	1
Appropriateness (correct technical level)	4	3	2	1
Accessibility (ease of finding information)	4	3	2	1
Physical qualities (binding, printing, illustrations)	4	3	2	1
Terminology (correct, consistent, and clear)	4	3	2	1
Number of examples	4	3	2	1
Quality of examples	4	3	2	1
Index	4	3	2	1

Please use the space below for your comments about this manual. Please include general comments about the usefulness of this manual. If you have discovered inaccuracies or omissions, please specify the number of the page on which the problem occurred.

Name ______________________________
Title ______________________________
Company ______________________________
Telephone ______________________________
Today's date ______________________________

Address ______________________________
City ______________________________
State/Country ______________________________
Zip code ______________________________
Electronic mail address ______________________________

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO 6184 ST. PAUL, MN

POSTAGE WILL BE PAID BY ADDRESSEE

CRAY RESEARCH, INC.

ATTN: Software Publications Group
655 LONE OAK DR BLDG F
EAGAN MN 55121-9957

Fold

Cut along this line