CF90™ Commands and Directives Reference Manual

SR–3901 2.0

Cray Research, Inc.

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

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

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 Commands and Directives Reference Manual SR–3901 2.0

The CF90 2.0 compiler 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. The CF90 2.0 compiler is not supported on CRAY T3D systems. (Note that implementation on CRAY T3E systems is deferred.)

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

The following `f90`(1) command-line options have changed:

- The `taskcommon` argument has been added to the `-a` option. When specified, it converts all common blocks in the compilation to task common blocks. This option is available on Cray PVP systems and SPARC systems only.
- Most array syntax statements now vectorize when `-G 0` is specified. Specifying `-G 1` allows both array syntax statements and inner loops to vectorize. This functionality is available on Cray PVP systems only.
- `46`, which is the default argument to the `-i` option on Cray PVP systems, is obsolescent. The `-i 46` option will be removed, for all platforms, in a later release. It will be replaced by the `-i 64` option. To achieve functionality equivalent to specifying `-i 46`, you can specify `-i 64` and `-O fastmd`.
- The `-k` *prof* option now works with `gprof`(1) and `prof`(1). This option is available on SPARC systems only.
- The `-N` option now accepts `132` as an argument, which lets you specify `132` as the default line length for fixed-format source code.
- The optimizations performed when `-O 0` | `1` | `2` | `3` is specified have been changed on SPARC systems.
- Module files on SPARC systems are now stored in files suffixed with `.o` and work like module files on Cray PVP systems. The `-p` option has been extended to allow directories. The compiler can search the directories for module files.

- Output from the -t, -v, and -V options differs from that generated for previous releases on Cray PVP systems. The output from the -v and -V options differs from that generated for previous releases on SPARC systems.

- Listings are placed in files suffixed with .lst.

The following f90(1) command-line options are new:

- -D *var* and -U *var* define and undefine variables used to control conditional compilation.

- -e / -d A generates the code necessary for using the MPP Apprentice tool. This option is available on CRAY T3E systems only.

- -e /-d P and -e/-d Z have been added to support conditional compilation.

- -e / -d R compiles all functions and subroutines as if they contained the RECURSIVE keyword.

- -e / -d u rounds floating-point division results upwards. It is now supported on SPARC systems.

- -e / -d I treats all variables as if an IMPLICIT NONE statement had been specified.

- The -O inline*n*, -O inlinefrom, and -O (no)modinline command-line options have been added to support the inlining feature. These options are available on Cray PVP systems and SPARC systems only.

- -O (no)fastint allows you to specify faster multiplication and division for INTEGER(KIND=8) values. This option is available on Cray PVP systems only.

- -O (no)msgs writes optimization messages to stderr.

- -O (no)negmsgs writes messages to stderr that describe why certain constructs did not optimize.

- -O (no)taskinner allows you to specify Autotasking for innermost loops and requests a threshold test prior to Autotasking loop nests. This option is available on Cray PVP systems and SPARC systems only.

- -O (no)threshold requests threshold testing to determine whether or not there is sufficient work in a loop nest before Autotasking is attempted. This option is available on Cray PVP systems and SPARC systems only.

- `-O unroll`*n* specifies an amount of loop unrolling.
- `-O (no)vsearch` vectorizes search loops. This option is available on Cray PVP systems only.
- `-P` *code_gen* generates position-independent code. This option is available on SPARC systems only.
- `-s` *size* allows you to change the default size of variables, literal constants, and intrinsic functions declared as type real, integer, logical, complex, double complex, or double precision. This option is available on CRAY T3E systems and SPARC systems only.

The following compiler directives are new:

- `CACHE_ALIGN` aligns each specified variable on a cache line boundary. This directive is available on CRAY T3E systems only.
- `COMMON` overrides the `-O taskcommon` command-line specification and causes common blocks to remain as common blocks. This directive is available on Cray PVP systems and SPARC systems only.
- `INLINE` and `NOINLINE` control whether automatic inlining analysis is performed. These directives are available on Cray PVP systems and SPARC systems only.
- `INLINE NEVER` and `INLINE ALWAYS` allow you to specify whether inlining should be attempted for a particular procedure. These directives are available on Cray PVP systems and SPARC systems only.
- `MODINLINE` and `NOMODINLINE` allow you to specify that a particular module should be compiled so that its contained procedures can be inlined when the module is used. These directives are available on Cray PVP systems and SPARC systems only.
- `PREFERTASK` allows single loops with large iteration counts to be considered as candidates for tasking. No threshold checks are generated for these loops. This directive is available on Cray PVP systems and SPARC systems only.
- `PREFERVECTOR` allows you to mark a loop as a candidate for vectorization. This directive is available on Cray PVP systems only.
- `STACK` causes the default storage allocation to be to the stack.

- SYMMETRIC declares that a PE-private stack variable has the same local address on all PEs. This directive is available on CRAY T3E systems only.
- UNROLL and NOUNROLL specify that the DO loop that follows the directive should be (or should not be) unrolled.
- VSEARCH and NOVSEARCH specify the vectorization of search loops. This directive is available on Cray PVP systems only.

The following Autotasking directives are new:

- CNCALL allows a loop to be autotasked by asserting that subroutines called from the loop have no recursive side effects. This directive is available on Cray PVP systems and SPARC systems only.
- MAXCPUS controls the number of CPUs allocated to the next parallel region. After the region is finished, the number of CPUs returns to the previous value. This directive is available on Cray PVP systems and SPARC systems only.
- NUMCPUS controls the number of CPUs allocated to the upcoming parallel regions. The number of CPUs stays constant until another NUMCPUS directive is encountered. This directive is available on Cray PVP systems and SPARC systems only.
- PERMUTATION declares that an integer array has no repeated values. This directive is available on Cray PVP systems and SPARC systems only.

The following new !MIC DO ALL parameters have been added:

- AUTOSCOPE specifies that all variables that have not been scoped explicitly should be scoped according to the default rules. This parameter is available on Cray PVP systems and SPARC systems only.
- NCPUS_CHUNKS specifies that iterations should be divided into the same number of chunks as the values of the NCPUS environment variable. This parameter is available on SPARC systems only and is the default implementation for SPARC systems.

Several new environment variables have been added.

The information previously contained in appendix sections A and B of this manual now appears in the *CF90 Fortran Language Reference Manual, Volume 3*, publication SR–3905.

Record of Revision

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

Version	***Description***
1.0	December 1993. Original printing. This document supports the CF90 compiler release 1.0 running on CRAY Y-MP systems.
1.1	June 1994. This document supports the CF90 compiler release 1.0 running on both CRAY Y-MP systems and on SPARC systems, including those from CRS.
2.0	November 1995. 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.

Preface

This reference manual describes the commands and directives for using the CF90 compiler, which is invoked through the `f90` command. The `f90` command can also invoke a source lister and the loader.

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 processing 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*.

  Note: Implementation of the CF90 2.0 compiler on CRAY T3E systems is deferred.

- SPARC systems, including Cray Research Superserver systems, running the Solaris 2.3, or later, operating system. This manual refers to these systems as *SPARC systems*.

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 previous Fortran 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 outcome of the outstanding interpretations to the standard.

Related CF90 publications

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

- *Introducing the CF90 Programming Environment*, publication IN–2155
- *CF90 Ready Reference*, publication SQ–3900
- *CF90 Ready Reference for SPARC Systems*, publication SQ–3904
- *CF90 Fortran Language Reference Manual, Volume 1*, publication SR–3902
- *CF90 Fortran Language Reference Manual, Volume 2*, publication SR–3903
- *CF90 Fortran Language Reference Manual, Volume 3*, publication SR–3905

If you are using the CF90 compiler on a SPARC system, *Introducing the CF90 Programming Environment* and the *CF90 Ready Reference for SPARC Systems* are included in the CF90 software package in hard copy form. The CF90 Fortran language reference manuals, volumes 1 through 3, are included in online form, but you can order hard copies of these and other publications from the Distribution Center in Mendota Heights, MN. See *Ordering publications* for more information.

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.

Section numbers appear in parentheses after man page names. Man pages are referenced in text by entry name and section number, as shown in the following example:

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

Related Fortran publications

The *CF90 Fortran Language Reference Manual, Volume 1*, publication SR–3902, and the *CF90 Fortran Language Reference Manual, Volume 2*, publication SR–3903, and *CF90 Fortran Language Reference Manual, Volume 3*, publication SR–3905, describe the Cray Research implementation of the Fortran 90 standard. Several commercial reference texts also contain information on the Fortran 90 language. The following two books are among the group of documents that can be useful to you in understanding the Fortran 90 language:

- Adams, J., W. Brainerd, J. Martin, B. Smith, and J. Wagener . 1992. *Fortran 90 Handbook – Complete ANSI/ISO Reference*. New York, NY: Intertext Publications/Multiscience Press, Inc.

- Metcalf, M. and J. Reid. 1990. *Fortran 90 Explained*. Oxford, UK: Oxford University Press.

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 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.

  Note: Depending on your system hardware or software, not all of these publications may be appropriate or available 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–201 1	*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*

Publication	Title
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*
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.
  655 Lone Oak Drive
  Eagan, Minnesota 55121
  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

The following conventions are used throughout this manual:

Convention	Meaning
`command`	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.
variable	Italic typeface denotes variable entries and words or concepts being defined.

Convention	Meaning
`user input`	This bold fixed-space font denotes literal items that the user enters in interactive sessions. Output is shown in nonbold, fixed-space font.
[ ]	Brackets surround optional portions of a syntax statement.
...	Ellipses indicate that a preceding command-line element can be repeated.

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 Internet, using the following Internet addresses:

  `publications@timbuk.cray.com`

- 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 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 Customer Support Call Center, using either of the following numbers:

  1–800–950–2729 (toll free from the United States and Canada)

  +1–612–683–5600

- Send a facsimile of your comments to the attention of “Software Publications” 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

Introduction [1]

Section 1

The CF90 compiler was developed to support Fortran code that conforms to the new Fortran standards adopted by the American National Standards Institute (ANSI) and the International Standards Organization (ISO). These standards, commonly referred to as Fortran 90 standards, are ANSI X3.198–1992 and ISO/IEC 1539:1991. Because the Fortran 90 standard is a superset of the FORTRAN 77 standard, the CF90 compiler will compile code written to the FORTRAN 77 standard.

Sections in this manual

1.1

This manual is organized into the following sections:

- Section 1 introduces the content of this manual and provides a general description of the compiler.
- Section 2 describes the `f90` command, which you use to invoke the compiler . This section includes information about the options you can use on the `f90` command line and environment variables that affect the CF90 compiler .
- Section 3 describes the compiler directives that the CF90 compiler recognizes.
- Section 4 describes Autotasking directives.
- Section 5 provides an overview of the other software products available in the CF90 programming environment.

CF90 overview

1.2

The CF90 compiler is invoked through the `f90`(1) command. The `f90` command also invokes `cflint`(1), listers, and a loader, as follows:

- If you are running CF90 on a Cray PVP system, the `f90` command invokes the loader `segldr`(1).
- If you are running CF90 on a CRAY T3E system, the `f90` command invokes the loader `mppldr`(1).
- If you are running CF90 on a SPARC system, the `f90` command invokes the loader `ld`(1).

In the most basic case, the `f90` command invokes the CF90 compiler, processes the input files named on the command line, and generates a binary file. The loader loads the binary file and generates an executable output file (the default output file is `a.out`). The lister generates the program's listing file.

In the following simple example, the `f90` command is used to invoke the compiler. Option `-r s` is specified to generate a source listing. File `pgm.f` is the input file. You run the program by entering the name of the output file name as a command; in this example, the default output file name, `a.out`, is used. Figure 1 illustrates this example.

```
% f90 -r s pgm.f
% ./a.out
```

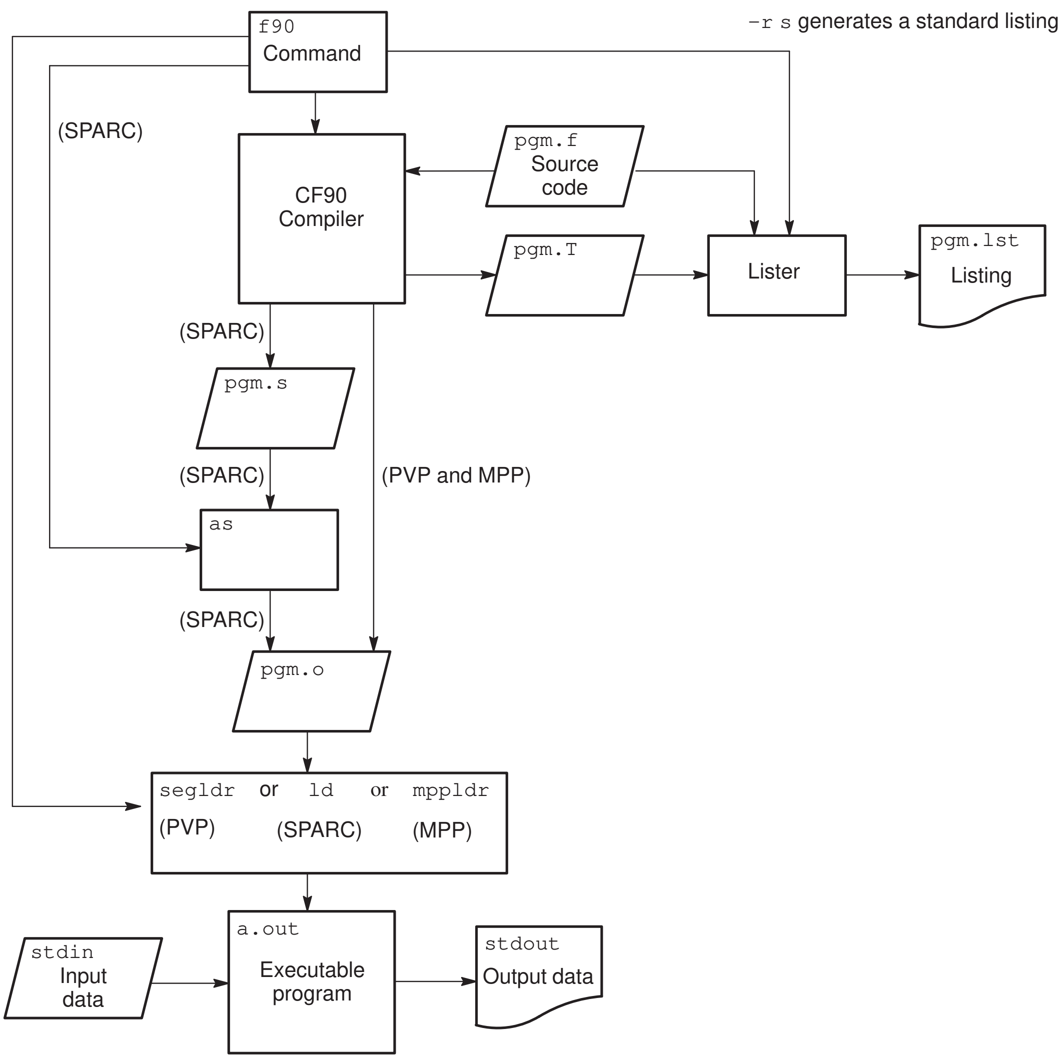

Figure 1. `f90`(1) command example

You can use the options on the `f90` command line to modify the default actions; for example, you can disable the load step. For more information on `f90` command-line options, see section 2.

Invoking CF90 [2]

This section describes the `f90`(1) command, which invokes the CF90 compiler. This section also describes environment variables used to determine shell characteristics when working with the CF90 compiler. The `f90`(1) online man page contains information from this section in an abbreviated form.

Setting up the CF90 programming environment

2.1

The installation and configuration of the CF90 programming environment uses a utility called *Modules*, which is provided and installed as part of the release package. File `/opt/ctl/doc/README` was distributed in the release package. It contains information on initializing the `module`(1) command and initializing the environment.

The default programming environment is available to you after you have entered the following command:

```
module load modules PrgEnv
```

If you have questions on setting up the programming environment, contact your system support staff.

The `f90` command

2.2

The `f90`(1) command invokes the CF90 compiler. This command has several options, and some options are used more often than others. The most commonly used options are as follows:

`f90` [`-d` *disable*] [`-D` *var*[`,`*var*]] [`-e` *enable*] [`-f` *source_form*]
[`-I` *incldir*] [`-m` *msg_lvl*] [`-O` *opt*[`,` *opt*...]] [`-p` *module_site*]
[`-r` *list_opt*] [`-U` *var*[`,`*var*]] [`--`] *file*`.`*suffix*[`90`] [*file*`.`*suffix*[`90`]] ...

Section 2

The full syntax of the command is as follows:

```
f90 [-a alloc ] [-b bin_obj_file ] [-c] [-C cifopts ] [-d disable ]
    [-D var [,var ]] [-e enable ] [-f source_form ] [-F] [-g]
    [-G debug_lvl ] [-i intlen ] [-I incldir ] [-k prof] [-l lib ] [-L dir ]
    [-m msg_lvl ] [-M msgs ] [-N col] [-o out_file ] [-O opt [, opt ...]]
    [-p module_site ] [-P code_gen ] [-r list_opt ] [-R runchk ] [-s size ]
    [-S source_file ] [-t num ] [-T] [-U var [,var ]] [-v] [-V]
    [-Wl"loader_opt "] [-Wr"cflist_opt "] [-x dirlist ] [-X npes ] [--]
    file .suffix [90] [file .suffix [90]] ...
```

Note: Some options are not available on all platforms. If you specify an option that is not supported on your platform, a message is issued and compilation continues when possible.

Note: On Cray PVP systems, some default values shown for `f90` command options may have been changed by your site. See your system support staff for further details.

–a *alloc* (SPARC systems and Cray PVP systems only)

2.2.1

The `-a` *alloc* option allows you to specify a storage allocation mechanism. Enter one of the following for *alloc* :

alloc	Action
`taskcommon`	Converts all common blocks in the compilation to task common blocks. Local variables named in `SAVE` statements are stored in taskcommon in that each processor (task) has a private copy of saved variables. This includes variables initialized in `DATA` statements or data declaration statements. For a given procedure, a `COMMON` compiler directive can override this command-line specification. For more information on the `COMMON` compiler directive, see subsection 3.2.7.3, page 115.

alloc	Action
`dalign`	Aligns double-precision and complex data on double word (64-bit) boundaries. If `-s default64` or `-s float64` are specified, all 64-bit objects are aligned on double word boundaries. By default, data is aligned on 32-bit word boundaries. This option is valid on SPARC systems only.

To specify both `taskcommon` and `dalign`, you must specify multiple `-a` options on the `f90` command line.

`-b` *bin_obj_file*

2.2.2

The `-b` *bin_obj_file* option disables the load step and saves the binary object file version of your program in *bin_obj_file* .

The `-b` *bin_obj_file* specification is ignored if it appears on the command line with either the `-e S` or the `-S` *source_file* options.

The `-c` option is redundant when specified with `-b` *bin_obj_file* because `-b` *bin_obj_file* automatically disables the load step.

Only one input file is allowed when the `-b` option is specified. If you have more than one input file, you can use the `-c` option to disable the load step and save the binary files to their default file names. If you have only one input file and neither the `-b` nor the `-c` options are specified, the binary version of your program is not saved.

By default, the binary file is saved in *file* `.o`, where *file* is the name of the source file and `.o` is the suffix used.

−c
2.2.3

The `−c` option disables the load step and saves the binary object file version of your program in *file* `.o`, where *file* is the name of the source file and `.o` is the suffix used. Using this option in conjunction with other options can lead to the following possible outcomes:

- If `−b` *bin_obj_file* is specified in conjunction with `−c`, the binary file is placed in *bin_obj_file* .
- If `−c` is specified in conjunction with `−o` *out_file* and there is only one input file, the binary file is saved in *out_file* .
- If you specify the `−c` option and there is more than one input file, a *file* `.o` is created for each input file specified.

If only one input file is processed and neither the `−b` nor the `−c` options are specified, the binary version of your program is not saved after the load is completed.

−C *cifopts*
2.2.4

The `−C` *cifopts* option creates a compiler information file (CIF) and places it in *file* `.T`, where *file* is the name of the source file and `.T` is the suffix used. If both the `−r` and `−C` options are specified, `−r` overrides `−C`.

By default, the `f90` command does not create a CIF. The `−C` option is used to create a CIF, which is required by various tools that analyze CF90 programs, such as `cflint`, `cflist`, and `xbrowse`. The user interface to the CIF information is through the CIF library, `libcif`.

The most common specification is `−C a`, which writes all CIF information possible. For information on the *cifopts* available, see section 4 of the *Compiler Information File (CIF) Reference Manual*, publication SR–2401.

`-d` *disable* and `-e` *enable*

2.2.5

The `-d` *disable* and `-e` *enable* options disable or enable compiling options. To specify more than one compiling option, enter the options without separators between them; for example, `-e af`. Enter one or more of the following for *disable* or *enable* :

Table 1. Compiling options

disable *enable*	Platforms	Default	Action if enabled
`a`	PVP, CRAY T3E, SPARC	Disabled	Aborts compilation after encountering the first error.
`f`	PVP	Disabled	Generates an output file during execution that is suitable for processing by `flowview`.
`i`	PVP	Disabled	Generates a run-time error when an uninitialized local variable is used in a floating-point operation or array subscript. This option causes allocated but uninitialized local storage to be set to an undefined value. When the `-e i` option is specified, variables are reset for each invocation of a subprogram, even in static storage. Therefore, a `SAVE` statement is needed to preserve the value of a variable between invocations.

Section 2

Table 1. Compiling options
(continued)

disable enable	Platforms	Default	Action if enabled
			This option does not apply to variables in common blocks (or those equivalenced to such variables), variables in a `DATA` statement, dummy arguments, or saved variables. However, the loader offers this feature for saved variables that were not allocated in an `ALLOCATE` statement. If the compiler detects use of an uninitialized variable, it generates a message; this feature is unrelated to the –e i option and does not apply to arrays.
`j`	PVP, SPARC	Disabled	Executes `DO` loops at least once.
`n`	PVP, CRAY T3E, SPARC	Disabled	Generates messages to note all nonstandard usage.
`p`	PVP, CRAY T3E, SPARC	Enabled	Enables double precision arithmetic. When disabled, variables declared on a `DOUBLE PRECISION` statement and constants specified with the `D` exponent are implicitly converted to default real type. This causes arithmetic operations and intrinsics involving these variables to have a default real type rather than a double-precision real type. Similarly, variables declared on a `DOUBLE COMPLEX` statement and complex constants specified with the `D` exponent are implicitly mapped to the complex type in which each part has a default real type. (The implementation of double precision arithmetic is deferred on CRAY T3E systems.)

Table 1. Compiling options (continued)

disable enable	Platforms	Default	Action if enabled
`q`	PVP, CRAY T3E, SPARC	Enabled	Aborts compilation if 100 or more errors are generated.
`r`	PVP	Enabled	Rounds multiplication results.
`t`	PVP, SPARC	Enabled, Disabled	Determines the memory allocation method used for automatic variables. An *automatic variable* is a variable that is not a dummy argument but whose declaration depends on a nonconstant expression (array bounds and/or character length). Storage is allocated for automatic variables and array temporaries upon entry to a procedure. Storage is deallocated upon exit from the procedure. When enabled, the compiler attempts to limit the amount of memory requested from the heap for automatic variables; that is, an attempt is made to allocate storage for them on the stack. If not enough stack memory is available, the heap is used.

Table 1. Compiling options (continued)

disable *enable*	Platforms	Default	Action if enabled
			On Cray PVP systems, by default, the CF90 compiler requests 16,000 words of stack space, for automatic arrays and array temporaries, with a stack increment size of 8,000 words. Initial size may be increased, and increment size may be increased or decreased through directives passed to the loader. Initial stack and increment sizes can affect performance because of the number of stack overflows that might occur. Multitasking can increase the occurrences of stack overflows and decrease optimal performance. Stack overflow occurrences can be minimized and performance increased by tuning initial stack and increment values.

`u`	PVP,
	CRAY T3E,
	SPARC
		Disabled,
		Disabled,
		Enabled
			Rounds floating-point division results upwards.
			
			When disabled, faster code sequences are generated for floating-point divides. This can result in slightly less accurate results because when results of floating-point divides are later assigned to integer variables, truncation errors can occur. For example, the following code fragment may result in the value of `i`, an integer , being truncated to an unexpected value:
			
			`x = 6.0`
			`r = x/3.0`
			`...`
			`i = r`

Table 1. Compiling options
(continued)

disable *enable*	Platforms	Default	Action if enabled
			On Cray PVP systems, excluding CRAY T90 systems that support IEEE floating-point arithmetic, the result of the division may yield a result of 1.99999.... When assigned to `i`, the value is truncated, which produces a value of 1 instead of the expected result of 2. On CRAY T3E systems, CRAY T90 systems that support IEEE floating-point arithmetic, and SPARC systems, floating-point division may not produce correctly rounded results when disabled because of the faster divide sequence that is used. This may affect the results when converting to an integer value or when comparing results with those obtained from another system that supports IEEE floating-point arithmetic. When enabled, additional instructions are generated that produce more accurate results. On systems that support IEEE floating-point arithmetic, a true IEEE divide sequence is generated (instead of possibly calculating a reciprocal and multiply sequence). On Cray PVP systems other than CRAY T90 systems that support IEEE floating-point arithmetic, results of floating-point divides are strongly rounded by multiplying the result by a value slightly greater than one.

Section 2

Table 1. Compiling options (continued)

disable *enable*	Platforms	Default	Action if enabled
`v`	PVP, CRAY T3E, SPARC	Disabled	Allocates variables to static storage. These variables are treated as if they had appeared in a `SAVE` statement. Two types of variables are not allocated to static storage: variables allocated in an `ALLOCATE` statement and local variables in explicit recursive procedures. Variables in explicit recursive procedures consist of those in functions, in subroutines, and in internal procedures within functions and subroutines that have been declared with the `RECURSIVE` attribute. The `STACK` compiler directive overrides `-e v`; for more information on this compiler directive, see subsection 3.2.7.4, page 116.
`A`	CRAY T3E	Disabled	Generates code necessary to use the MPP Apprentice tool.
`B`	PVP, CRAY T3E, SPARC	Enabled	Generates binary output. If disabled, inhibits all optimization and allows only syntactic and semantic checking.
`I`	PVP, CRAY T3E, SPARC	Disabled	Treats all variables as if an `IMPLICIT NONE` statement had been specified. Does not override any `IMPLICIT` statements or explicit type statements. All variables must be typed.
`P`	PVP, CRAY T3E, SPARC	Disabled	Performs source preprocessing on *file*`.f[90]` or *file*`.F[90]` but does not compile. Generates *file*`.i`.
`R`	PVP, CRAY T3E, SPARC	Disabled	Compiles all functions and subroutines as if they contained a `RECURSIVE` keyword.

Table 1. Compiling options
(continued)

disable *enable*	Platforms	Default	Action if enabled
`S`	PVP, CRAY T3E, SPARC	Disabled	Generates assembly language output and saves it in *file* `.s`. When the `-e S` option is specified on the command line with the `-S` *source_file* option, the `-S` *source_file* option overrides the `-e S` option. If `-e S` is specified with the `-b` *bin_obj_file* option, the `-b` *bin_obj_file* option is ignored.
`X`	PVP, SPARC	Disabled	Generates additional run-time code needed to support the Autotasking Expert System, ATExpert. To use ATExpert, you must have specified either `-O 3`, `-O task2`, or `-O task3` on the command line or you must have included Autotasking (`!MIC`) directives in your source code. See the `atexpert`(1) man page for more information about ATExpert.
`Z`	PVP, CRAY T3E, SPARC	Disabled	Performs source preprocessing and compilation on *file* `.f[90]` or *file* `.F[90]`. Generates *file* `.i`.

`-D` *var* [`,` *var*] ...
2.2.6

Defines variables used for conditional compilation. If the input file name ends in `.F` or `.F90`, GPP is invoked and it recognizes the `-D` and `-U` options on the `f90`(1) command line. If GPP is not invoked, these options are recognized directly by the CF90 compiler. For more information on GPP, see `gpp`(1).

The `-U` option undefines variables used for conditional compilation. For more information on the `-U` option, see subsection 2.2.29, page 58.

For more information on conditional compilation, see the *CF90 Fortran Language Reference Manual, Volume 3*, publication SR–3905.

`–f` *source_form*
2.2.7

The `–f` *source_form* option specifies whether the Fortran source file is written in fixed source form or free source form. For *source_form* , enter `free` or `fixed`. The *source_form* specified here overrides any source form implied by the source file suffix.

The default source form is `fixed` for input files that end with a `.f` or `.F` suffix. The default source form is `free` for input files that end with a `.f90` or `.F90` suffix.

If the file ends in `.F` or `.F90`, GPP is invoked. For more information on GPP, see `gpp`(1).

`–F`
2.2.8

The `–F` option enables macro expansion throughout the source file. Typically, macro expansion occurs only on directive lines. If the source file suffix is neither `.F` nor `.F90`, this option is ignored.

`–g`
2.2.9

The `–g` option provides debugging support identical to specifying the `–G 0` option.

`–G` *debug_lvl*
2.2.10

The `–G` *debug_lvl* option generates a debug symbol table and establishes a debugging level. The debugging level determines the points at which breakpoints can be set. The frequency and position of breakpoints can curtail optimization partially or totally. At higher debugging levels, fewer breakpoints can be set, but optimization is increased. Enter one of the following for *debug_lvl* :

debug_lvl	Support
`0`	Default debugging support. Breakpoints can be set at each line. This level of debugging can be specified when `-O 0` is specified. Disables scalar optimization, tasking, and most vectorization; array syntax statements vectorize at this level. This level can also be obtained by specifying the `-g` option.
`1`	Performs block-by-block debugging, with the exception of innermost loops. You can place breakpoints at statement labels on executable statements and at the beginning and end of block constructs (such as `IF/THEN/ELSE` blocks, `DO/END DO` blocks, and at `SELECT CASE/END SELECT` blocks). This level of debugging can be specified when `-O 0` or `-O 1` is specified. Disables some scalar optimization and all loop nest restructuring. Only user tasking is enabled. On Cray PVP systems, this *debug_lvl* allows vectorization of some inner loops and most array syntax statements. Vectorization is equal to that performed when `-O vector1` is in effect.

`-i` *intlen*
2.2.11

The `-i` *intlen* option specifies 64-bit, 46-bit, or 32-bit integer arithmetic for default integers. Enter `64`, `46`, or `32` for *intlen* . The default *intlen* values are as follows:

intlen default	Platform
46	Cray PVP systems (obsolescent)
64	CRAY T3E systems
32	SPARC systems

Warning: Specifying `-i 46` is obsolescent. It causes the compiler to generate a warning message. This functionality will be removed in the CF90 3.0 release, and the new Cray PVP system default will be `-i 64`.

This option specifies 64-bit, 46-bit, or 32-bit integer arithmetic for operations involving multiplication, division, conversion to or from real, and comparisons. Specifying 32- or 46-bit integer arithmetic does not limit all representations to 32 or 46 bits, but it enables the compiler to generate faster code for some expressions. Specifying `64` generates a full 64-bit code sequence and performance is slower. To increase performance, you can specify `-O fastint` in conjunction with `-i 64`. For more information on `-O fastint`, see subsection 2.2.20.20, page 39.

A value specified in an `INTEGER` compiler directive overrides this option. The `INTEGER` compiler directive is described in subsection 3.2.5.3, page 110.

If a module is compiled separately from a program unit that uses the module, they both must be compiled with the same *intlen* . If program units are compiled separately and linked, they must all be compiled with the same *intlen* .

`-I` *incldir*

2.2.12

The `-I` *incldir* option specifies a directory to be searched for files named in `INCLUDE` lines in the Fortran source file. You must specify an `-I` option for each directory you want searched. Directories can be specified in *incldir* as full path names or as path names relative to the working directory.

The following example causes the compiler to search for files included within `earth.f` in the directories `/usr/local/sun` and `../moon`:

```
f90 -I /usr/local/sun -I ../moon earth.f
```

If the `INCLUDE` line in the source file specifies an absolute name, that is, one that begins with a slash (`/`), that name is used, and no other directory is searched. If the `INCLUDE` uses a relative name, that is, one that does not begin with a slash (`/`), the compiler searches for the file in the directory of the source file containing the `INCLUDE` line. If this directory contains no file of that name, the compiler then searches the directories named by the `-I` options, as specified on the command line, from left to right.

Section 2

`-k` *prof* (SPARC systems only)

2.2.13

The `-k` *prof* option prepares the object file for use with the `prof`(1) or `gprof`(1) profiling tool. Enter one of the following for *prof*:

prof	Profiling feature
`s`	Prepares the object files for profiling with `prof`(1). This allows for profiling by procedure, which shows the number of calls to each procedure and the percent of time used by each procedure. File `mon.out` is written. You can then run an execution profile using `prof`.
`g`	Prepares the object files for profiling with `gprof`(1). This allows for profiling by procedure, which shows the number of calls to each procedure and the percent of time used by each procedure. This option produces counting code in the manner of `-k s`, but it invokes a run time recording mechanism that keeps more extensive statistics. This option writes file `gmon.out`. You can then run an execution profile using `gprof`.

`-l` *lib*

2.2.14

The `-l` *lib* option directs the loader to search for the specified object library file when loading an executable file.

- On Cray PVP systems and CRAY T3E systems, if *lib* begins with a period (`.`) or a slash (`/`), it is assumed to be a full path name, and the loader uses it as is. Otherwise, the loader searches for a file named `lib`*lib*`.a` in each directory specified in the library search path. For more information on library search rules, see subsection 2.2.15.
- On SPARC systems, the loader searches for a file named `lib`*lib*`.so` or `lib`*lib*`.a` in each directory specified in the library search path. Precedence is given to `lib`*lib*`.so`. For example, if `-l m` is specified, the loader searches for a file named `libm.so` or `libm.a`. For more information on library search rules, see subsection 2.2.15.

–L *dir*
2.2.15

The –L *dir* option directs the loader to search for object library files in the specified directory before searching in the standard directories. The loader searches for library files specified by –l options in the directories specified by preceding –L options.

- On Cray PVP systems and CRAY T3E systems, *dir* can take the form of a comma-separated list of directories. The loader searches for library files in directory *dir* before checking in the standard directories. The standard system directories are /lib and /usr/lib.

  For example, if –L ../mylibs,/loclib and –l m are specified on a Cray PVP system, the loader searches for the following files and uses the first one found:

  ```
  ../mylibs/libm.a
  /loclib/libm.a
  /opt/ctl/craylibs/craylibs
  /lib/libm.a
  /usr/lib/libm.a
  ```

  See segldr(1) or mppldr(1) for more information on library searches. Note that the f90(1) command or /opt/ctl/bin/segldr adds /opt/ctl/craylibs/craylibs to your search path.

- On SPARC systems, the loader searches for library files in directory *dir* before checking in the standard directories, which are /opt/craysoft/f90e/lib, /usr/ccs/lib, and /usr/lib. If the LD_LIBRARY_PATH environment variable is set to a list of directories, these are searched after those specified by any –L options but before the standard directories. See subsection 2.4.9, page 66, for more information on the LD_LIBRARY_PATH environment variable.

  By default, the CF90 programming environment is installed in /opt/craysoft/f90e/lib. If this directory does not exist, see your system administrator for the alternate location.

For example, if `LD_LIBRARY_PATH` is set to `/tools/lib:/loclib` and `-L ../mylibs` and `-l m` are specified on the command line, the loader searches for the following files and uses the first one found:

```
../mylibs/libm.so
../mylibs/libm.a
/tools/lib/libm.so
/tools/lib/libm.a
/loclib/libm.so
/loclib/libm.a
/opt/craysoft/f90e/lib/libm.so
/opt/craysoft/f90e/lib/libm.a
/usr/ccs/lib/libm.so
/usr/ccs/lib/libm.a
/usr/lib/libm.so
/usr/lib/libm.a
```

See `ld`(1) for more information on library searches.

For information on specifying modules, see subsection 2.2.21, page 48.

-m ***msg_lvl***

2.2.16

The `-m` *msg_lvl* option specifies the minimum compiler message levels to enable. The following list shows the integers to specify in order to enable each type of message and which messages are generated by default.

msg_lvl	Message types enabled
0	Error , warning, caution, note, and comment
1	Error , warning, caution, and note
2	Error , warning, and caution
3	Error and warning (default)
4	Error

Caution and warning messages denote, respectively, possible and probable user errors.

By default, messages are sent to the standard error file, `stderr`, and are displayed on your terminal. If the `-r` option is enabled, messages are also sent to the listing file.

To see more detailed explanations of messages, use the `explain` command. This command retrieves message explanations and displays them online. For example, to obtain documentation on message 500, enter the following command:

```
explain cf90-500
```

The default *msg_lvl* is `3`, which suppresses messages at the comment, note, and caution level.

`-M` *msgs*
2.2.17

The `-M` *msgs* option suppresses specific messages. For *msgs*, specify one or more integer numbers that correspond to the CF90 messages you want to suppress. If you want to specify more than one message number, enter a comma (but no spaces) between the message numbers. For example, to disable messages `cf90-100` and `cf90-200`, specify `-M 100,200`.

`-N` *col*
2.2.18

The `-N` *col* option specifies the line width for fixed-format source lines. For *col*, enter `72` to specify 72-column lines, enter `80` to specify 80-column lines, and enter `132` to specify 132-column lines.

By default, fixed-format source lines are 72 characters wide.

`-o` *out_file*
2.2.19

The `-o` *out_file* option overrides the default executable file name, `a.out`, with *out_file*. If this option is specified on the command line with the `-c` option, the binary object file is written to *out_file* and the load step is disabled. If this option is specified on the command line with the `-e S` or the `-S` options, the `-o` *out_file* option has no effect.

Section 2

***–O opt*[, *opt*]...**

2.2.20

You can use the `–O` *opt* option of the `f90`(1) command line to specify the following optimization features:

- General optimization levels. The `–O 0`, `–O 1`, `–O 2`, and `–O 3` options allow you to specify a general level and type of optimization. The range is from no optimization, at level `–O 0`, to aggressive optimization, at `–O 3`. Generally , as the level increases, so does execution performance. Each level implies a certain level of scalar optimization, tasking, and vectorization.

  With increased execution performance, however, you may notice increases in compile time and compile size. You may also notice slight differences in results at different optimization levels.

- Specific optimization levels. The specific *opt* values, such as `vector1`, `scalar3`, `task2`, and so on, determine the extent of the vectorization, scalar optimization, and Autotasking that the compiler will perform.

  The `scalar`*n*, `vector`*n*, and `task`*n* *opt* levels must be specified in logical combinations. For example, `vector0` is compatible with `task0` and `task1` only. If you specify incompatible levels, the compiler issues an error message. If you leave a level unspecified, the compiler raises or lowers the level to be compatible with those specified and issues a warning message. For example, specifying `–O task3` causes the compiler to raise the scalar optimization level to `scalar3`. Likewise, specifying `–O scalar0` causes the compiler to lower the vector optimization level to `vector0`.

  Note: You can use only one method of specifying scalar optimization, vectorization, or Autotasking. The *opt* values `0`, `1`, `2`, and `3` cannot be specified in conjunction with *opt* values `scalar0`, `scalar1`, `scalar2`, `scalar3`, `vector0`, `vector1`, `vector2`, `vector3`, `task0`, `task1`, `task2`, or `task3`.

- Specific optimization characteristics. The `–O` option accepts arguments to control bottom loading of loops, pattern matching, zero incrementing, and several other optimization features. Some of these features can also be controlled through compiler directives.

Cray PVP system defaults: `2`, `scalar2`, `task1`, `vector2`, `noaggress`, `nobl`, `fastint`, `noloopalign`, `nomsgs`, `nomodinline`, `nonegmsgs`, `nooverindex`, `pattern`, `recurrence`, `notaskinner`, `threshold`, `unroll2`, `vsearch`, and `nozeroinc`.

CRAY T3E system defaults: `2`, `scalar2`, `noaggress`, `nobl`, `jump`, `nomsgs`, `nonegmsgs`, `nooverindex`, `nozeroinc`, `pattern`, `recurrence`, `split1`, and `unroll0`.

SPARC system defaults: `1`, `scalar1`, `noaggress`, `nomsgs`, `nomodinline`, `nonegmsgs`, `nooverindex`, `pattern`, `recurrence`, `taskinner`, `threshold`, `unroll0`, and `nozeroinc`. Certain optimization features available on Cray PVP systems are not available on SPARC systems. The following optimization features are in effect at all times on SPARC systems: `vector0`, `nobl`, `nofastmd`, `nojump`, `noloopalign`, and `novsearch`.

Table 2 summarizes the optimization features enabled when different *opt* levels are specified. The optimization specifications available differ depending on your platform. If your command line includes an option that is not supported on your platform, the compiler issues a message and compilation continues.

Table 2. `-O` *opt* summary

opt	Description	Platforms	For more information
`0`	No optimization.	PVP, CRAY T3E, SPARC	page 30
`1`	Conservative optimization. Default general optimization level on SPARC systems.	PVP, CRAY T3E, SPARC	page 31

Table 2. `-O` *opt* summary (continued)

opt	Description	Platforms	For more information
`2`	Moderate optimization. Default general optimization level on Cray PVP systems and CRAY T3E systems.	PVP, CRAY T3E, SPARC	page 31
`3`	Aggressive optimization and Autotasking.	PVP, CRAY T3E, SPARC	page 32
`scalar0`	No scalar optimization.	PVP, CRAY T3E, SPARC	page 32
`scalar1`	Conservative scalar optimization. Default scalar optimization level on SPARC systems.	PVP, CRAY T3E, SPARC	page 32
`scalar2`	Moderate scalar optimization. Default scalar optimization level on Cray PVP systems and CRAY T3E systems.	PVP, CRAY T3E, SPARC	page 33
`scalar3`	Aggressive scalar optimization.	PVP, CRAY T3E, SPARC	page 33
`task0`	No Autotasking.	PVP, SPARC	page 34
`task1`	User Autotasking. Default tasking optimization level.	PVP, SPARC	page 34

Table 2. `-O` *opt* summary (continued)

opt	Description	Platforms	For more information
`task2`	Moderate Autotasking.	PVP, SPARC	page 34
`task3`	Aggressive Autotasking.	PVP, SPARC	page 35
`vector0`	Vectorization of array syntax statements only.	PVP	page 35
`vector1`	Conservative vectorization.	PVP	page 35
`vector2`	Moderate vectorization. Default vectorization optimization level.	PVP	page 36
`vector3`	Aggressive vectorization.	PVP	page 36
`inline0`	No inlining.	PVP, SPARC	page 36
`inline1`	Conservative inlining.	PVP, SPARC	page 36
`inline2`	Moderate inlining.	PVP, SPARC	page 36
`inline3`	Aggressive inlining.	PVP, SPARC	page 36
`aggress`, `noaggress`	`aggress` raises the limits for internal tables, which increases opportunities for optimization. The default is `noaggress`.	PVP, CRAY T3E, SPARC	page 39

Table 2. `-O` *opt* summary (continued)

opt	Description	Platforms	For more information
`bl`, `nobl`	`bl` enables full bottom loading of scalar operands in loops. The default is `nobl`.	PVP, CRAY T3E	page 39
`fastint`, `nofastint`	`fastint` performs faster multiplication, division, and compare sequences for `INTEGER(KIND=8)` and `INTEGER*8` data objects. The default is `fastint`.	PVP	page 39
`inlinefrom=`*source*	Specifies a file or directory that contains procedures for inline code expansion in the source file being compiled.	PVP, SPARC	page 39
`jump`, `nojump`	`jump` generates jumps instead of branches to external functions. The default is `jump`.	CRAY T3E	page 40
`loopalign`, `noloopalign`	`loopalign` causes the compiler to attempt to align `DO` and `DO WHILE` loops on buffer boundaries, comparing buffer length with the number of generated instructions in each loop. The default is `noloopalign`.	PVP	page 41

Table 2. -O *opt* summary (continued)

opt	Description	Platforms	For more information
`modinline`, `nomodinline`	`modinline` compiles modules so that any procedures contained within them can be inlined when they are used. The default is `nomodinline`.	PVP, SPARC	page 41
`msgs`, `nomsgs`	`msgs` causes the compiler to write optimization messages to `stderr`. The default is `nomsgs`.	PVP, CRAY T3E, SPARC	page 42
`negmsgs`, `nonegmsgs`	`negmsgs` writes negative optimization messages to `stderr`. The default is `nonegmsgs`.	PVP, CRAY T3E, SPARC	page 42
`overindex`, `nooverindex`	`nooverindex` declares that there are no array subscripts that index a dimension of an array that are outside the declared bounds of that dimension. The default is `nooverindex`.	PVP, CRAY T3E, SPARC	page 42
`pattern`, `nopattern`	`pattern` enables pattern matching for library substitution. The default is `pattern`.	PVP, CRAY T3E, SPARC	page 43

Section 2

Table 2. -O *opt* summary
(continued)

opt	Description	Platforms	For more information
`recurrence`, `norecurrence`	On Cray PVP systems, `recurrence` enables vectorization of reduction loops. On CRAY T3E systems and SPARC systems, `recurrence` rewrites multiplication operations to be a series of addition operations. The default is `recurrence`.	PVP, CRAY T3E, SPARC	page 44
`split0`, `split1`, `split2`	Specifies various levels of loop splitting. The default is `split1`.	CRAY T3E	page 44
`taskinner`, `notaskinner`	`taskinner` specifies Autotasking for innermost loops and requests that a threshold test be performed prior to Autotasking. On Cray PVP systems, the default is `notaskinner`. On SPARC systems, the default is `taskinner`.	PVP, SPARC	page 45
`threshold`, `nothreshold`	`threshold` performs threshold testing to determine whether there is sufficient work in a loop nest before Autotasking is attempted. The default is `threshold`.	PVP, SPARC	page 45

Table 2. -O *opt* summary
(continued)

opt	Description	Platforms	For more information
`unroll0`, `unroll1`, `unroll2`	Specifies various levels of loop unrolling. On CRAY T3E systems and SPARC systems, the default is `unroll0`. On Cray PVP systems, `unroll2` is enabled at all times.	CRAY T3E, SPARC	page 45
`vsearch`, `novsearch`	`vsearch` vectorizes search loops. The default is `vsearch`.	PVP	page 46
`zeroinc`, `nozeroinc`	`zeroinc` specifies that constant increment variables (CIVs) can be incremented by zero. The default is `nozeroinc`.	PVP, CRAY T3E, SPARC	page 46

For information on Autotasking, see section 4, page 119.

The following subsections describe the effects of various `-O` specifications. Some optimization specifications are not available on all platforms. If your command line includes an option that is not supported on your platform, the compiler issues a message and compilation continues.

2.2.20.1 `-O 0`

No optimization. Characteristics include low compile time, small compile size, and no global scalar optimization.

On Cray PVP systems and SPARC systems, all tasking is disabled.

On Cray PVP systems, most array syntax statements are vectorized, but all other vectorization is disabled.

`-O 1`
2.2.20.2

Conservative optimization. Characteristics include moderate compile time and size, global scalar optimizations, and no loop nest restructuring. Results may differ from the results obtained when `-O 0` is specified because of operator reassociation. No optimizations will be performed that might create false exceptions.

On Cray PVP systems, only array syntax statements and inner loops are vectorized, and the system does not perform some vector reductions. All tasking is disabled.

On SPARC systems, this is the default optimization level. All tasking is disabled.

`-O 2`
2.2.20.3

Moderate optimization. Characteristics include moderate compile time and size, global scalar optimizations, pattern matching, and loop nest restructuring. The system does not perform optimizations that might create false exceptions; for example, on Cray PVP systems, it employs safe bottom loading.

Results may differ from results obtained when `-O 1` is specified. On SPARC systems, this is because of possible loop nest restructuring. On Cray PVP systems, this is because of vector reductions.

On Cray PVP systems and CRAY T3E systems, this is the default level of optimization.

On Cray PVP systems and SPARC systems, user tasking is enabled, so `!MIC$` directives are recognized.

`-O 3`
2.2.20.4

Aggressive optimization and Autotasking. Characteristics include a potentially larger compile size, longer compile time, global scalar optimizations, possible loop nest restructuring, pattern matching, and Autotasking. The optimizations performed might create false exceptions in rare instances; for example, on Cray PVP systems, the system performs full bottom loading.

Note that Autotasking is supported on Cray PVP systems and SPARC systems.

Results may differ from the results obtained when `-O 1` is specified. On SPARC systems, this is because of possible loop nest restructuring. On Cray PVP systems, this is because of vector reductions.

Autoscoping rules are in effect. For more information on these rules, see subsection 4.3, page 133.

`-O scalar0`
2.2.20.5

No scalar optimization. Characteristics include low compile time and size. Compatible with `task0` or `task1`.

On Cray PVP systems, `scalar0` is compatible with `vector0`.

`-O scalar1`
2.2.20.6

Conservative scalar optimization. Moderate compile time and size. Results can differ from the results obtained when `-O scalar0` is specified because of operator reassociation. No optimizations are performed that could create false exceptions; for example, on Cray PVP systems, the system performs only safe bottom loading.

On SPARC systems, analysis that determines whether a variable is used before it is defined is disabled at this level. Compatible with `task0` or `task1`. This is the default scalar optimization level.

On Cray PVP systems, `scalar1` is compatible with `vector0` or `vector1` and `task0` or `task1`.

2.2.20.7 `-O scalar2`

Moderate scalar optimization. Characteristics include moderate compile time and size. Results can differ slightly from the results obtained when `-O scalar1` is specified because of possible loop nest restructuring.

Generally , no optimizations are done that could create false exceptions. For example, on Cray PVP systems and CRAY T3E systems, only safe bottom loading is performed.

On SPARC systems, analysis that determines whether a variable is used before it is defined is enabled at this level. Compatible with all tasking levels.

On Cray PVP systems, `scalar2` is compatible with all vectorization and tasking levels.

On Cray PVP systems and CRAY T3E systems, this is the default scalar optimization level.

2.2.20.8 `-O scalar3`

Aggressive scalar optimization. Potentially greater compile time and size. Results can differ from the results obtained when `-O scalar1` is specified because of possible loop nest restructuring.

The optimization techniques used can create false exceptions in rare instances; for example, on Cray PVP systems and CRAY T3E systems, full bottom loading is performed. Analysis that determines whether a variable is used before it is defined is enabled at this level.

The `scalar3` optimization level is never enabled automatically, even when `-O 3` is specified. This scalar optimization level must be requested specifically on the command line.

On SPARC systems, `scalar3` is compatible with all tasking levels.

On Cray PVP systems, `scalar3` is compatible with all tasking and vectorization levels.

On CRAY T3E systems, CRAY T90 systems that support IEEE arithmetic, and SPARC systems, strength reduction of floating-point values is performed.

Section 2

2.2.20.9 `-O task0` (Cray PVP systems and SPARC systems only)

No tasking. Characteristics include low compile time and size. All tasking is disabled. `!MIC$` directives are ignored.

On Cray PVP systems, `task0` is compatible with all vectorization and scalar optimization levels.

On SPARC systems, `task0` is compatible with all scalar optimization levels.

2.2.20.10 `-O task1` (Cray PVP systems and SPARC systems only)

User tasking. Characteristics include low compile time and size. User tasking is enabled, so `!MIC$` directives are recognized. No level for scalar optimization is enabled automatically. This is the default optimization level for Autotasking.

On Cray PVP systems, `task1` is compatible with all vectorization and scalar optimization levels.

On SPARC systems, `task1` is compatible with all scalar optimization levels.

2.2.20.11 `-O task2` (Cray PVP systems and SPARC systems only)

Moderate Autotasking. Characteristics include moderate compile time and size. Possible loop nest restructuring. No optimizations that can create exceptions or differing results are performed.

On Cray PVP systems, `task2` is compatible with `scalar2` or `scalar3` and `vector2` or `vector3`.

On SPARC systems, `task2` is compatible with `scalar2` or `scalar3`.

Autoscoping rules are in effect. For more information on these rules, see subsection 4.3, page 133.

2.2.20.12 `-O task3` (Cray PVP systems and SPARC systems only)

Aggressive Autotasking. Characteristics include a potentially high compile time and size. Possible loop nest restructuring. Results can differ slightly from those obtained when `-O task2` is specified, and from run to run, because of parallel reductions.

On Cray PVP systems, `task3` is compatible with `scalar2` or `scalar3` and `vector3`.

On SPARC systems, `task3` is compatible with `scalar2` or `scalar3`.

Autoscoping rules are in effect. For more information on these rules, see subsection 4.3, page 133.

2.2.20.13 `-O vector0` (Cray PVP systems only)

Low vectorization. Characteristics include low compile time and small compile size. Vector code is generated for most array syntax statements but not for user-coded loops.

This level of vectorization is compatible with `task0` or `task1` and with all scalar optimization levels.

2.2.20.14 `-O vector1` (Cray PVP systems only)

Conservative vectorization. Characteristics include moderate compile time and size. No loop nests are restructured. Only inner loops are vectorized. Not all vector reductions are performed, so results do not differ from results obtained when `-O vector0` is specified. No vectorizations that might create false exceptions are performed.

This level of vectorization is compatible with `scalar1`, `scalar2`, or `scalar3` and `task0` or `task1`.

2.2.20.15 `-O vector2` (Cray PVP systems only)

Moderate vectorization. Characteristics include moderate compile time and size. Loop nests are restructured. Results can differ slightly from results obtained when `-O vector1` is specified because of vector reductions. No vectorizations that might create false exceptions will be performed. Pattern matching is enabled.

This level of vectorization is compatible with `scalar2` or `scalar3` and `task0`, `task1`, or `task2`. This is the default vectorization level.

2.2.20.16 `-O vector3` (Cray PVP systems only)

Aggressive vectorization. Characteristics include potentially high compile time and size. Loop nests are restructured. Results can differ slightly from results obtained when `-O vector1` is specified because of vector reductions. Vectorizations that might create false exceptions in rare cases may be performed. Pattern matching is enabled.

This level of vectorization is compatible with `scalar2` or `scalar3` and with all tasking levels.

2.2.20.17 `-O inline0`, `-O inline1`, `-O inline2`, `-O inline3` (Cray PVP systems and SPARC systems only)

The `-O inline`*n* options let you specify the amount of automatic inline code expansion desired. Inlining is the process of replacing a user subroutine or function call with the definition itself. This saves subprogram call overhead. A loop that contains a call that has been inlined becomes a candidate for vectorization or tasking.

Certain general criteria must be met in order for any inlining to occur:

- The procedure to be inlined must be available in an inlinable form. This can be accomplished in any of the following ways:
  - The procedure is in the same file being compiled as the call site. A *call site* is the place at which a `CALL` statement that names the procedure appears or is the place at which a function reference that names the procedure appears.

  - If the source is a module procedure that is use associated into the calling procedure, an inlinable form of the procedure is available if the module was compiled with `-O modinline`.
  - The procedure is available in a file or directory that was compiled using the `-O inlinefrom` option.

- The program unit that calls the procedure must have been compiled with `-O inline1`, `-O inline2`, `-O inline3`, or `-O inlinefrom` specified on the `f90` command line.
- The procedure being called must not be specified in a `!DIR INLINE NEVER` directive.
- The call site must not be within the range of a `!DIR$ NOINLINE` directive.

The following coding constructs can inhibit inlining:

- An assumed-size character item in the argument list
- Multiple entry points in the called routine
- Dummy argument types and sizes that differ from corresponding actual argument types and sizes
- The number of dummy arguments declared differs from the number of actual arguments
- The result type declared in the called subprogram differs from what is expected to be returned at the call site
- A function that contains a declaration of a parallel region and a call to that subprogram from another parallel region
- Modification of local static data inside a subprogram that was included in the compilation with an `INLINEFROM` compiler directive

The `-O inline0` option disables all inline code expansion, and the `-O inline3` option specifies aggressive inline code expansion. Table 3 explains the levels in more detail:

Table 3. Inlining specifications

Level	Description	Notes
`-O inline0`	No inlining	All inlining disabled. The `!DIR$ INLINE` and `!DIR$ NOINLINE` compiler directives are ignored. Default.
`-O inline1`	Conservative inlining	Inlining attempted if the procedure being called is specified on a `!DIR$ INLINE ALWAYS` directive.
`-O inline2`	Moderate inlining	Inlining attempted on calls described in `-O inline1`. In addition, medium-sized procedures that were not mentioned on a `!DIR$ INLINE ALWAYS` directive are inlined.
`-O inline3`	Aggressive inlining	Inlining attempted for all procedures in the compilation unit.

Certain other command-line options and compiler directives also affect inlining. The following list shows these features and indicates where they are described:

Feature	For more information, see:
`-O inlinefrom`	subsection 2.2.20.21, page 39
`-O modinline`, `-O nomodinline`	subsection 2.2.20.24, page 41
`INLINE`, `NOINLINE`	subsection 3.2.2.1, page 92
`INLINE NEVER`, `INLINE ALWAYS`	subsection 3.2.2.2, page 92
`MODINLINE`, `NOMODINLINE`	subsection 3.2.2.3, page 96

2.2.20.18 `-O aggress`, `-O noaggress`

During the optimizing phase of program compilation, `aggress` causes the compiler to raise the limits for internal tables. The higher limits cause large loops to be optimized but can also increase compile time and space. The default is `noaggress`.

On Cray PVP systems, this allows the compiler to vectorize some loops that might have been inhibited by those limits.

On CRAY T3E systems, this causes more aggressive register assignment and instruction scheduling.

2.2.20.19 `-O bl`, `-O nobl` (Cray PVP systems and CRAY T3E systems)

`bl` enables full bottom loading of scalar operands in loops. Bottom loading is enabled only when scalar optimization is enabled. `nobl` specifies safe bottom loading. The default is `nobl`. This feature is also available through compiler directives; for more information, see subsection 3.2.4.2, page 100.

2.2.20.20 `-O fastint`, `-O nofastint` (Cray PVP systems only)

`fastint` performs faster multiplication and division sequences for `INTEGER(KIND=8)` and `INTEGER*8` data objects. On CRAY T90 systems that support IEEE floating-point arithmetic, `fastint` also performs fast integer comparisons.

`nofastint` performs full 64-bit integer operations. The default is `fastint`.

Note that `fastint` does not produce a result that is precise to a full 64 bits of precision.

2.2.20.21 `-O inlinefrom=`*source* (Cray PVP systems and SPARC systems only)

Specifies inline code expansion of all subprograms contained in *source*. For *source*, enter the name of a file that contains one or more subprograms or enter the name of a directory that contains one or more source files.

Every procedure specified in the file or directory is inlined unless the call is within the scope of a `NOINLINE` directive or unless the procedure is specified in an `INLINE NEVER` directive.

If no inlining level is specified in conjunction with this option, `inline1` is used.

Certain other command-line options and compiler directives also affect inlining. The following list shows these features and indicates where they are described:

Feature	For more information, see:
`-O inline`*n*	subsection 2.2.20.17, page 36
`-O modinline`, `-O nomodinline`	subsection 2.2.20.24, page 41
`INLINE`, `NOINLINE`	subsection 3.2.2.1, page 92
`INLINE NEVER`, `INLINE ALWAYS`	subsection 3.2.2.2, page 92
`MODINLINE`, `NOMODINLINE`	subsection 3.2.2.3, page 96

2.2.20.22 `-O jump`, `-O nojump` (CRAY T3E systems only)

`jump` causes jumps, instead of branches, to external functions. By default, jumps are generated instead of branches because this is safer. Branches are limited in the distance over which they can transfer control; jumps have no such limitations. Large programs may have to use `-O jump` when compiling files that generate calls to functions that are loaded at a large offset from the position of the call site that is invoking the procedure. A loader message similar to the following alerts you to the necessity of compiling with `-O jump`:

```
mppldr-331 _mppldr:  CAUTION
DEX expression 7 module 'xxx$f' calculated a relative
branch target too distant.  The last symbol
referenced was 'abc'.
```

The default is `jump`.

2.2.20.23 `-O loopalign`, `-O noloopalign` (Cray PVP systems only)

`loopalign` causes the compiler to attempt to align `DO` and `DO WHILE` loops on buffer boundaries, comparing buffer length with the number of generated instructions in each loop. Loop alignment is useful when program execution is dominated by specific blocks of code. If such a block crosses a buffer boundary, the overhead caused by frequent reloading of instruction buffers degrades program performance.

When `loopalign` is specified, the compiler counts the number of generated instructions in each `DO` or `DO WHILE` loop and compares this to the length of the instruction buffers. If the loop body fits in the buffers and would otherwise cross over the buffer boundary, the loop is aligned on a buffer boundary . Short loops, unwound loops, and loops containing external references are not aligned.

This option does not interact with the `ALIGN` compiler directive. The default is `noloopalign` (no attempt to align loops).

2.2.20.24 `-O modinline`, `-O nomodinline` (Cray PVP systems and SPARC systems only)

`modinline` directs the compiler to create the information it needs in order to generate inlinable code for specified module procedures. When `-O modinline` is specified, the `MODINLINE` and `NOMODINLINE` directives are recognized.

Using the `-O modinline` option increases compile size. The default is `-O nomodinline`.

Certain other command-line options and compiler directives also affect inlining. The following list shows these features and indicates where they are described:

Feature	For more information, see:
`-O inline`*n*	subsection 2.2.20.17, page 36
`-O inlinefrom`	subsection 2.2.20.21, page 39
`INLINE`, `NOINLINE`	subsection 3.2.2.1, page 92
`INLINE NEVER`, `INLINE ALWAYS`	subsection 3.2.2.2, page 92
`MODINLINE`, `NOMODINLINE`	subsection 3.2.2.3, page 96

`-O msgs`, `-O nomsgs` 2.2.20.25

`msgs` causes the compiler to write optimization messages to `stderr`. These messages include `VECTOR`, `SCALAR`, and `TASK` messages. The default is `nomsgs`, in which case you must specify that a listing be produced so that you can see the optimization messages in the listing.

`-O negmsgs`, `-O nonegmsgs` 2.2.20.26

`negmsgs` causes the compiler to generate messages that indicate why optimizations such as vectorization or tasking did not occur in a given instance.

The default is `nonegmsgs`.

`-O overindex`, `-O nooverindex` 2.2.20.27

`nooverindex` declares that there are no array subscripts that index a dimension of an array and that are outside the declared bounds of that dimension. On Cray PVP systems, shortloop code generation occurs when the extent does not exceed the maximum vector length of the machine.

Specifying `overindex` declares that the program contains code that makes array references with subscripts that exceed the defined extents. This prevents the compiler from performing the shortloop optimizations described in the preceding paragraph.

Overindexing is nonstandard, but it compiles correctly as long as data dependencies are not hidden from the compiler. This technique *collapses* loops; that is, it replaces a loop nest with a single loop. An example of this practice is as follows:

```
DIMENSION A(20, 20)
DO I = 1, N
   A(I, 1) = 0.0
END DO
```

Assuming that `N` equals 400 in the previous example, the compiler can generate more efficient code than a doubly nested loop. However, incorrect results can occur in this case if `nooverindex` is in effect. The practice can be found in user codes, but it is especially prevalent in codes generated by FPP. (FPP is the dependency analysis phase of the Cray Research CF77 compiling system.)

You do not need to specify `overindex` if the overindexed array is a Cray pointee, has been equivalenced, or if the extent of the overindexed dimension is declared to be 1 or *. In addition, the `-O overindex` option is enabled automatically for the following CF90 extension code, where the number of subscripts in an array reference is less than the declared number:

```
DIMENSION A(20, 20)
DO I = 1, N
   A(I) = 0.0   ! 1-dimension reference;
                ! 2-dimension array
END DO
```

The default is `nooverindex`.

2.2.20.28 `-O pattern`, `-O nopattern`

`pattern` enables pattern matching. The pattern matching feature searches your code for specific code patterns and replaces them with calls to scientific library routines. On Cray PVP systems and CRAY T3E systems, the scientific library used is `libsci.a`. On SPARC systems, the scientific library used is the CraySoft `libsci`. These routines are highly optimized and may contain multitasked code.

`pattern` is enabled only for optimization levels `-O 2`, `-O vector2` or higher; there is no way to force pattern matching for lower levels.

On CRAY T3E systems, only PE-private data is supported.

Specifying `nopattern` disables pattern matching. The default is `pattern`.

2.2.20.29 `-O recurrence`, `-O norecurrence`

On Cray PVP systems, `recurrence` enables vectorization for all reduction loops.

On CRAY T3E systems and SPARC systems, `recurrence` may rewrite some multiplication operations to be a series of addition operations.

The default is `recurrence`. This feature is also available through compiler directives; for more information, see subsection 3.2.1.5, page 82.

2.2.20.30 `-O split0`, `-O split1`, `-O split2` (CRAY T3E systems only)

The `-O split0`, `-O split1`, and `-O split2` options specify loop splitting, as follows:

- `split0`. Disables loop splitting and directs the compiler to ignore `SPLIT` compiler directives.
- `split1`. Causes the compiler to split only the loops that are preceded by a `SPLIT` compiler directive. Default
- `split2`. Causes the compiler to evaluate all loops in the compilation as candidates for splitting except those that are preceded by a `NOSPLIT` directive.

The default is split1. This feature is also available through compiler directives. For more information, see subsection 3.2.4.5, page 105.

2.2.20.31 `–O taskinner`, `–O notaskinner` (Cray PVP systems and SPARC systems only)

`taskinner` specifies Autotasking for innermost loops and requests that a threshold test be performed prior to Autotasking. The threshold test determines whether there is sufficient parallel work in a loop nest before Autotasking is attempted. Autotasking must be enabled for this directive to take effect. Autotasking is enabled when `–O 3`, `–O task2`, or `–O task3` is specified.

The default is `notaskinner`.

2.2.20.32 `–O threshold`, `–O nothreshold` (Cray PVP systems and SPARC systems only)

`threshold` generates a runtime threshold test to determine whether there is sufficient work in a loop nest before Autotasking is attempted. Autotasking must be enabled for this directive to take effect. Autotasking is enabled when `–O 3`, `–O task2`, or `–O task3` is specified.

The default is `threshold`.

2.2.20.33 `–O unroll0`, `–O unroll1`, `–Ounroll2` (CRAY T3E systems and SPARC systems only)

The unrolling arguments specify loop unrolling, as follows:

- `unroll0`. Disables loop unrolling and directs the compiler to ignore `UNROLL` compiler directives. Default.
- `unroll1`. Causes the compiler to unroll only the loops that are preceded by an `UNROLL` compiler directive.
- `unroll2`. Causes the compiler to evaluate all loops in the compilation as candidates for unrolling except those that are preceded by a `NOUNROLL` compiler directive.

On Cray PVP systems, `unroll2` is enabled at all times.

The default is `unroll0`. This feature interacts with the `UNROLL` compiler directive. For more information, see subsection 3.2.1.8, page 86.

Section 2

`-O vsearch`, `-O novsearch` (Cray PVP systems only)
2.2.20.34

`vsearch` vectorizes search loops. `novsearch` argument disables vectorization of search loops. A *search loop* is one that can be exited by means of an `IF` statement or `EXIT` statement.

`vsearch` is the default when `vector2` or `vector3` are enabled. `novsearch` is the default when `vector0` or `vector1` are enabled.

This feature is also available through compiler directives; for more information, see subsection 3.2.1.11, page 91.

`-O zeroinc`, `-O nozeroinc`
2.2.20.35

`zeroinc` causes the compiler to assume that constant increment variables (CIVs) can be incremented by zero. A CIV 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`, where `K` can be equal to zero, `J` is a CIV. `zeroinc` can cause less strength reduction to occur in loops that have variable increments.

The default is `nozeroinc`, which means that you must prevent zero incrementing.

Optimization values
2.2.20.36

Figure 2 shows the relationships between some of the *opt* values on Cray PVP systems.

> **Note:** In Figure 2, vectorization information applies to Cray PVP platforms only. Tasking information applies to Cray PVP systems and SPARC systems only.

	scalar0	scalar1	scalar2	scalar3	vector0	vector1	vector2	vector3	task0	task1	task2	task3
Low compile cost	✘				✘				✘	✘		
Moderate compile cost		✘	✘			✘	✘				✘	
Potentially high compile cost				✘				✘				✘
†No numerical differences from serial execution	✘	✘										
No numerical differences from serial execution (no vector reductions)					✘	✘						
Potential numerical differences from serial execution (vector reductions)							✘	✘				
Potential numerical differences from serial execution and from run to run due to parallel reductions												✘
Potential numerical differences from serial execution (operator reassociation)		✘	✘	✘								
No optimizations that may create exceptions (safe bottom loading)	✘	✘	✘									
No optimizations that may create exceptions	✘	✘	✘		✘	✘	✘		✘	✘	✘	
Optimizations that may create exceptions (full bottom loading)				✘								
Implies at least scalar1						✘				✘		
Implies at least scalar2							✘	✘			✘	✘
No loop nest restructuring					✘	✘			✘	✘		
Loop nest restructuring							✘	✘			✘	✘
Vectorize array syntax statements					✘	✘	✘	✘				
Vectorize only inner loops						✘						
No vectorization may create exceptions					✘	✘	✘					
Vectorization that may create exceptions								✘				
All tasking disabled									✘			
User tasking enabled										✘	✘	✘

†Left-to-right evaluation is deferred

Figure 2. Optimization values

Section 2

−p ***module_site***
2.2.21

The `−p` *module_site* option allows you to specify files or directories that contain modules.

The *module_site* specifies the name of a binary file or directory to search for modules. The *module_site* specified can be an archive file, build file (`bld` file), or binary file.

When searching files, the compiler searches files suffixed with `.o` (*file* `.o`) or library files suffixed with `.a` (*lib* `.a`) containing one or more modules. When searching directories, the compiler searches files in the named directory that are suffixed with `.o` or `.a`.

> **Note:** When using the CF90 1.0 compiler on a SPARC system, if you specified `−p` *file* `.o`, the compiler automatically opened *file* `.M`. The CF90 2.0 compiler reads and searches for `.M` files created by the CF90 1.0 compiler . The compiler's ability to search for and recognize `.M` files will be removed in a later release.

The module files are not interchangeable. For example, it is not possible to create a module with a CF90 compiler based on a SPARC system and read the file on a Cray PVP system.

File name substitution (such as `*.o`) is not allowed. If the path name begins with a slash (`/`), the name is assumed to be an absolute path name. Otherwise, it is assumed to be a path name relative to the working directory . If you need to specify multiple binary files, library files, or directories, you must specify a `−p` option for each *module_site* . There is no limit on the number of `−p` options that you can specify. The compiler searches the binary file, library files, and directories in the order specified.

Example 1: Consider the following command line:

```
f90  −p steve.o  −p mike.o  joe.f
```

Assume that `steve.o` contains a module called `Rock` and `mike.o` contains a module called `Stone`. A reference to use `Rock` in `joe.f` causes the compiler to use `Rock` from `steve.o`. A reference to `Stone` in `joe.f` causes the compiler to use `Stone` from `mike.o`.

Example 2: The following example specifies binary file `murphy.o` and library file `molly.a`:

```
f90  -p murphy.o  -p molly.a  prog.f
```

Example 3: In this example, assume that the following directory structure exists in your home directory:

```
                programs
               /    |    \
          tests   one.f   two.f
            |
         use_it.f
```

The following module is in file `programs/one.f`, and the compiled version of it is in `programs/one.o`:

```
MODULE one
INTEGER i
END MODULE
```

The next module is in file `programs/two.f`, and the compiled version of it is in `programs/two.o`:

```
MODULE two
INTEGER j
END MODULE
```

The following program is in file `programs/tests/use_it.f`:

```
PROGRAM demo
USE one
USE two
. . .
END PROGRAM
```

To compile `use_it.f`, enter the following command from your home directory , which contains the subdirectory `programs`:

```
f90  -p programs  programs/tests/use_it.f
```

Example 4: In the next set of program units, a module is contained within the first program unit and accessed by more than one program unit. The first file, `progone.f`, contains the following code:

```
MODULE split
INTEGER k
REAL a
END MODULE

PROGRAM demopr
USE split
INTEGER j
j = 3
k = 1
a = 2.0
CALL suba(j)
PRINT *, 'j=',j
PRINT *, 'k=',k
PRINT *, 'a=',a
END
```

The second file, `progtwo.f`, contains the following code:

```
SUBROUTINE suba(l)
USE split
INTEGER l
l = 4
k = 5
CALL subb(l)
RETURN
END

SUBROUTINE subb(m)
USE split
INTEGER m
m = 6
a = 7.0
RETURN
END
```

Use the following command line to compile the two files with one `f90` command and a relative path name:

```
f90  -p progone.o  progone.f  progtwo.f
```

`-P` *code_gen* (SPARC systems only)

2.2.22

Generates position-independent code (PIC). This option allows you to create library code that can be linked dynamically. Enter one of the following for *code_gen* :

code_gen	Compiler action
`s`	Produces PIC. Each reference to a global datum is generated as a dereference of a pointer in the global offset table. Each function call is generated in program counter (PC)-relative addressing mode through a procedure linkage table. The size of the global offset table is limited to 8K.
`l`	Produces code similar to that generated when `-P s` is in effect, but `-P l` allows the global offset table to span the range of 32-bit addresses. Specify `-P l` if your program has too many global data objects for `-P s`.

For any dynamic library, use all `-P s` or all `-P l`.

`-r` *list_opt*

2.2.23

The `-r` *list_opt* option causes a listing file to be produced. If the `-C` option is specified along with the `-r` option, the `-C` option is overridden and a warning message is generated. If both the `-r` option and the `-Wr` options are specified, the listing options specified by all the `-r` options are placed on the `cflist` command line before those specified by the `-Wr` option.

By default, if only one input file is specified on the `f90`(1) command line, the listing is placed in *file*`.lst`. If more than one input file is specified on the command line, the listing is placed in file `cflist.out`.

The `-r` option allows you to generate a listing directly from the `f90` command. The *list_opt* argument produces listings with commonly needed information. If you need a customized report, you can use `cflist` directly.

The `-r` option accepts only one argument unless you are specifying the `T` argument; `T` can be specified in conjunction with one other argument.

The *list_opt* values are as follows:

list_opt	Listing type
`0`	Listing includes the standard `cflist` listing information. Includes global reports when you have more than one compilation unit.
`1`	Listing includes the source listing, loop marks, and parallel marks. Regardless of the number of compilation units, no global reports are included. This is a minimal listing.
`2`	Listing includes the source listing, loop marks, parallel marks, considerata, and compiler messages. Regardless of the number of compilation units, no global reports are included. Produces a short, standard listing.
`3`	Listing includes the source listing, loop marks, parallel marks, an argument report, a parallelism report, identifier cross-references, considerata, and compiler messages. Regardless of the number of compilation units, no global reports are included.
`4`	Listing includes all subprogram items that can be listed with the default `cflist` options. Regardless of the number of compilation units, no global reports are included.
`5`	Listing includes all subprogram items that can be listed with the default `cflist` options. If more than one compilation unit is present, the Global Considerata report is included.
`6`	Listing includes all items that can appear in a listing. All `cflist` options are enabled. This option produces the maximum amount of listing information.
`c`	Listing includes all common blocks and all members of each common block.

list_opt	Listing type
`g`	Saves the generated binary code and its assembly language equivalent to *file*`.L`. Unlike most other `-r` arguments, specifying `-r g` does not create a CIF. On SPARC systems, specifying `-r g` does not generate a relocatable object file (*file*`.o`) and does not call the loader.
`l`	Invokes `cflint`(1) with default `cflint` options. You cannot pass option information directly to `cflint`. If you require a particular `cflint` analysis, you must call it separately. If more than one source file is listed on the command line, the output is written to file `cflint.out`.
`m`	Produces a source listing with loopmark information. Loopmark information will not be displayed if the `-d B` option has been specified.
`s`	Lists source code. Messages are interspersed with the source lines. Produces 80-column output by default.
`w`	Produces 132-column output, which, when specified in conjunction with `-r s` or `-r x`, overrides the 80-column output that those options produce by default.
`x`	Generates a cross-reference listing. Produces 80-column output by default.
`T`	Retains *file*`.T` after processing rather than deleting it. This option may be specified in addition to any of the other options. For more information on *file*`.T`, see the `-C` option.

−R *runchk*

2.2.24

The `−R` *runchk* option lets you specify any of a group of run-time checks for your program. The run-time checks available are as follows:

runchk	Checking performed
`a`	Compares the number and types of arguments passed to a procedure with the number expected.
`b`	Enables checking of array bounds. If a problem is detected at run time, a message is issued but execution continues. The `BOUNDS` and `NOBOUNDS` compiler directives override this option. For more information on `BOUNDS` and `NOBOUNDS`, see subsection 3.2.5.1, page 107.
`c`	Enables conformance checking of array operands in array expressions. Even without the `−R` option, such checking is performed during compilation when the dimensions of array operands can be determined.
`s`	Enables checking of character substring bounds. This option behaves similarly to option `−Ra` or `−Rb`.
`C`	Passes a descriptor for the actual arguments as an extra argument to the called routine and sets a flag to signal the called routine that this descriptor is included.
`E`	Creates a descriptor for the dummy arguments at each entry point and tests the flag from the caller to see if argument checking should be performed. If the flag is set, the argument checking is done.

If argument checking is to be done for a particular call, the calling routine must have been compiled with either `−R a` or `−R C` and the called routine must have been compiled with either `−R a` or `−R E`. `−R a` is equivalent to `−R CE`. The separation of `−R a` into `−R C` and `−R E` allows some control over which calls are checked.

Libraries can be compiled with `−R E`. If the program that is calling the libraries is compiled with either `−R a` or `−R C`, library calls are checked. If the calling routines are not compiled with `−R a` or `−R C`, no checking occurs.

Slight overhead is added to each entry sequence compiled with `-RE` or `-Ra` and to each call site compiled with `-RC` or `-Ra`. If a call site passes the extra information to an entry that is compiled to perform checking, the checking itself costs a few thousand clock periods per call. This cost depends on the number of arguments at the call.

Some nonstandard code behaves differently when argument checking is used. Different behavior can include run-time aborts or changed results. The following example illustrates this problem:

```
CALL SUB1(10,15)
CALL SUB1(10)
END

SUBROUTINE SUB1(I,K)
PRINT *,I,K
END
```

Without argument checking, if the two calls in this example share the same stack space for arguments, subroutine `SUB1` prints the values 10 and 15 for both calls. However, with argument checking enabled, an extra argument is added to the argument list, overwriting any previous information that was there. In this case, the second call to `SUB1` prints 10, followed by an incorrect value.

If full argument checking is enabled by `-Ra`, a message reporting the mismatch in the number of arguments is issued. This problem occurs only with nonstandard code in which the numbers of actual and dummy arguments do not match.

Section 2

−s *size*
2.2.25

The `−s` *size* option allows you to modify the sizes of variables, literal constants, and intrinsic function results declared as type real, integer, logical, complex, double complex, or double precision. For *size*, enter one of the following:

size	Action
`default32`	Adjusts the default sizes as follows: real, integer, and logical are set to 32 bits; complex and double precision are set to 64 bits; double complex is set to 128 bits. This *size* option is valid on CRAY T3E systems only.
`default64`	Adjusts the default sizes as follows: real, integer, and logical are set to 64 bits; complex and double precision are set to 128 bits; double complex is set to 256 bits. This *size* option is valid on SPARC systems only.
`float64`	Causes real values to be treated as double precision values and sized as follows: real is set to 64 bits; complex and double precision are set to 128 bits; double complex is set to 256 bits. Programs that make assumptions based on storage association should not be compiled with `−s float64`. This *size* option is valid on SPARC systems only.
`i`	Relaxes the strict data typing rules of Fortran 90 when more than one kind exists for a type. Specifying `−s i` causes data types to map to only the standard intrinsic FORTRAN 77 types and provides compatibility with the Cray Research CF77 compiling system. This *size* option is valid on Cray PVP systems and CRAY T3E systems only.

If you specify `−s` *size* in conjunction with data declarations that include size in your source code, the default size is not used. For example, if you specify `REAL(KIND=`*n*`) R` or `COMPLEX(KIND=`*n*`) Z`, the values specified in *n* are used.

Note: Programs relying on storage association of integer and real values, whether through `EQUIVALENCE` statements, overindexing, or redeclaration of `COMMON` blocks, are likely to fail if compiled with `-i 64` or `-s float64`. Only `-s default64` supports such associations.

Intrinsic procedures are interpreted accordingly; for example, `SQRT` (a generic) is treated as `DSQRT` (a specific) when a `-s default64` or `-s float64` option is used. If you are using a specific intrinsic, this option does not affect the intrinsic because the specific entry point is chosen. If you are using a generic, it is affected, and the correct entry point for that input type is chosen.

If both `-s default64` and `-s float64` are specified on the `f90`(1) command line, `-s default64` is used.

On SPARC systems, code compiled with the `-s` option executes more slowly because of the increased precision.

`-S` *source_file*

2.2.26

The `-S` *source_file* option specifies the assembly language output file name. When `-S` *source_file* is specified on the command line with either the `-e S` or `-b` *bin_obj_file* options, the `-e S` and `-b` *bin_obj_file* options are overridden.

`-t` *num* (Cray PVP systems only)

2.2.27

The `-t` *num* option specifies the number of bits to be truncated on floating-point operations. For *num*, enter an integer in the range $0 \leq num \leq 47$. The default is 0.

This option is not valid on CRAY T3E systems or SPARC systems. If specified, it is ignored, and a value of 0 is used.

–T
2.2.28

The –T option disables the compiler but displays all options currently in effect. The CF90 compiler generates information identical to that generated when the –v option is specified on the command line; when –T is specified, however , no processing is performed. When this option is specified, output is written to the standard error file (stderr).

–U var [, var] ...
2.2.29

The –U option undefines variables used for conditional compilation. If the input file name ends in .F or .F90, GPP is invoked and it recognizes the –D and –U options on the f90(1) command line. If GPP is not invoked, these options are recognized directly by the CF90 compiler. For more information on GPP, see gpp(1).

The –D option undefines variables used for conditional compilation. For more information on the –D option, see subsection 2.2.6, page 15.

For more information on conditional compilation, see the *CF90 Fortran Language Reference Manual, Volume 3*, publication SR–3905.

–v
2.2.30

The –v option sends compilation information to the standard error file (stderr). The information generated indicates the compilation phases, as they occur, and all options and arguments being passed to each processing phase.

–V
2.2.31

The –V option directs each compilation phase to send a message containing version information to the standard error file (stderr). Unlike all other command-line options, you can specify this option without specifying an input file name; that is, specifying f90 –V is valid.

–Wl"*loader_opt*"

2.2.32

The `–Wl"`*loader_opt* `"` option passes *loader_opt* directly to the loader. For example, specifying `–Wl"–D MAP=FULL"` passes the argument `MAP=FULL` directly to the loader's `–D` option. For more information on loader options, see `segldr`(1), `mppldr`(1), or `ld`(1).

–Wr"*cflist_opt*"

2.2.33

The `–Wr"`*cflist_opt* `"` option passes *cflist_opt* directly to `cflist`(1). For example, specifying `–Wr"–o cfile.o"` passes the argument `cfile.o` directly to the `cflist –o` option; this directs `cflist` to override the default output listing and put the output file in `cfile.o`. If you specify the `–Wr` option, you must specify the `–r` option. For more information on the `cflist` options, see `cflist`(1).

–x *dirlist*

2.2.34

The `–x` *dirlist* option disables specified directives or specified classes of directives. For *dirlist* , enter one of the following:

dirlist	Directives disabled
`all`	All compiler directives and Autotasking directives. For descriptions of the compiler directives, see section 3, page 73. For descriptions of the Autotasking directives, see section 4, page 119.
`dir`	All compiler directives.
directive	One or more compiler directives or Autotasking directives. If specifying more than one, separate them with commas.
`mic`	All Autotasking directives.

–X *npes* (CRAY T3E systems only)

2.2.35

The number of processor elements (PEs) in the partition.

Section 2

– –
2.2.36

The -- symbol signifies the end of options. After this symbol, you can specify files to be processed. This symbol is optional. It may be useful if your input files begin with one or more dash (-) characters.

***file*. *suffix*[90] [*file*. *suffix*[90]] ...**
2.2.37

This option names the file or files to be processed, where *suffix* is either an uppercase `F` or a lowercase `f`. The file suffixes indicate the content of each file and determine whether the compiler, assembler, or loader will be invoked.

If the file ends in `.F` or `.F90`, GPP is invoked. For more information on GPP, see `gpp`(1).

Files containing uncompiled Fortran code must be in one of the following forms:

- *file* `.f` or *file* `.F`, which indicates that this is a fixed source form file
- *file* `.f90` or *file* `.F90`, which indicates that this is a free source form file

The source form specified on the `-f` option overrides the source form implied by the file suffixes.

By default, several files are created during processing. The CF90 compiler adds a suffix to the *file* portion of the file name and places the files it creates into your working directory.

Assembly language output is sent to *file* `.s`. File names ending with `.s` are assembled, and the assembled code is written to the corresponding *file* `.o`.

The compiled code is written to *file* `.o` in the current directory.

You can specify precompiled `.o` files as input files. Input file names ending with `.o` are passed to the loader. The following information may be useful to you if you are using precompiled files:

- On Cray PVP and CRAY T3E systems, if your input files include precompiled CF90, CFT77, Cray Standard C, Cray C++, Cray Pascal, or Cray Ada files, they should be specified as files suffixed with `.o` on the command line. Files suffixed with `.o`, including any `.o` files written by the CF90 compiler, are passed to the loader in the order in which they appear on the `f90` command line. If, however, loading has been disabled by specifying options `-c` or `-b` *bin_obj_file* , no files are passed to the loader . Note that Cray Pascal and Cray Ada are available only for Cray PVP systems.
- On SPARC systems, if your input files include precompiled CF90 files or precompiled files from another compiler that complies to the SPARC Application Binary Interface (ABI), they should be specified as files suffixed with `.o` on the command line. Files suffixed with `.o`, including any `.o` files written by the CF90 compiler, are passed to the loader in the order in which they appear on the `f90` command line. If, however, loading has been disabled by specifying options `-c` or `-b` *bin_obj_file* , no files are passed to the loader. Module files are stripped from `.o` files, so your `a.out` file may be larger if you load your program separately from the `f90`(1) command.

The loader produces an executable file; by default, `a.out`. See the `-o` *out_file* option for information on specifying a different executable file. If only one source file is specified on the command line, the `.o` file is created and deleted. To retain the `.o` file, use the `-c` option to disable the loader.

The following is a file summary:

File	Type
`a.out`	Executable output file.
file `.a`	Library files to be searched for external references.
file `.f` or *file* `.F`	Input Fortran source file in fixed source form. If *file* ends in `.F` or `.F90`, GPP is invoked. For more information on GPP, see `gpp`(1).

Section 2

File	Type
file .f90 or *file* .F90	Input Fortran source file in free source form. If *file* ends in .F or .F90, GPP is invoked. For more information on GPP, see gpp(1).
file .lst	Listing file.
file .o	Relocatable object file.
file .s	Assembly language file.
file .L	File containing binary code and generated assembly language output.
file .T	CIF output file.
cflint.out	cflint(1) output file created when the -r l option is specified and there are multiple source files.
cflist.out	cflist(1) output file created when the -r option is specified to invoke cflist(1) and there are multiple source files.

The loader allows other file types. See the segldr(1), mppldr(1), and ld(1) man pages for more information on these files.

2.3 Loading programs on SPARC systems

To load specific libraries, you can specify the libraries on the f90(1) command line as follows:

f90 -o multi multi.o -L<*dir containing unbundled* libm> -lm

or

f90 -o multi multi.o -l<*dir containing unbundled* libm>/libm.a

Multitasked programs on SPARC systems must be linked with the system thread library, `/usr/lib/libthread.so`. This library is automatically included if a link step is performed and if the optimization level is at least `-O 1`, `-O scalar2`, or `-O task1`.

It is possible for a program to contain multitasking calls, and require `libthread.so`, at lower levels of optimization. An example would be a program that contains macrotasking calls and is compiled at `-O 0`. In such a case, `-l thread` must be specified on the `f90`(1) command line for the link step. This links in the proper Solaris multitasking support routines. An alternative would be to compile the program from source to object form using `-c` and the desired low optimization option. You would then need to run the compiler in a separate pass with a higher optimization level to link the object files into an executable program. The separate pass at the higher optimization level would have to be high enough to force automatic inclusion of the `libthread.so` library.

2.4 Environment variables

Environment variables are predefined shell variables, taken from the execution environment, that determine some of your shell characteristics. Several environment variables pertain to the CF90 compiler. The CF90 compiler recognizes general and multiprocessing environment variables. On Cray PVP systems, it also recognizes a variable for CPU targeting.

The multiprocessing variables in the following subsections affect the way your program will perform on multiple processors. Many of these control the same keywords as those of the `TSKTUNE`(3) multitasking library call. Using environment variables lets you tune the system for parallel processing without rebuilding libraries or other system software.

The environment variables apply to all compilations in a session. The following example shows how to set an environment variable:

With the standard shell, enter:

```
TARGET=cpu
export TARGET
```

Section 2

With the C shell, enter:

```
setenv TARGET cpu
```

The following subsections describe the environment variables recognized by the CF90 compiler.

> **Note:** Many of the environment variables described in this section refer to the default system locations of programming environment components. If the CF90 programming environment has been installed in a nondefault location, see your system support staff for path information.

2.4.1 AS (SPARC systems only)

Path name of `as`, the SPARC assembler. If both the `AS` and `CAL` environment variables are set, the value in `AS` overrides the value in `CAL`. The default is `/usr/ccs/bin/as`.

2.4.2 AUXBUF (Cray PVP systems only)

The `AUXBUF` environment variable specifies the number of buffers, for auxiliary arrays, that are held in memory. The default value is 64 buffers.

2.4.3 AUXPAGE (Cray PVP systems only)

The `AUXPAGE` environment variable specifies the size of each `AUX` buffer. The size is given in units of 512 words each and must be a power of 2. The default value is 2, indicating 1024 words per buffer.

2.4.4 CAL

Path name of the assembler.

On Cray PVP systems, except for CRAY T90 systems that support IEEE floating-point arithmetic, the default is `/bin/as`.

On CRAY T3E systems and on CRAY T90 systems that support IEEE floating-point arithmetic, the default is `/opt/ctl/bin/as`.

On SPARC systems, this environment variable is included for compatibility with Cray PVP systems. If both the AS and CAL environment variables are set, the value in AS overrides the value in CAL. The default is /usr/ccs/bin/as.

2.4.5 CAL_M (CRAY T3E systems only)

Path name of the MPP assembler. The default is /opt/ctl/bin/cam.

2.4.6 CFLIST

Path name of cflist(1) and cflint(1).

On Cray PVP systems and CRAY T3E systems, the default is /opt/ctl/craytools/craytools/bin/cflist.

On SPARC systems, the default location is /opt/craysoft/f90e/bin/cflist.

2.4.7 CRI_F90_OPTIONS

The CRI_F90_OPTIONS environment variable specifies additional options to be attached to the command line. These options are added following the options specified directly on the command line. File names cannot appear. This file is inserted at the rightmost portion of the command line before the input files and binary files are listed.

This allows you to set the environment variable once and have the specified set of options used in all compilations. This is especially useful for adding options to compilations done with build tools.

For example, assume that this variable was set as follows:

```
setenv CRI_F90_OPTIONS -G0
```

With the variable set, a command-line specification of f90 -c t.f is equivalent to a command-line specification of f90 -c t.f -G0.

LD (SPARC systems only)

2.4.8

Path name of the SPARC loader. If both the LD and SEGLDR environment variables are set, the value in LD overrides the value in SEGLDR. The default is /usr/ccs/bin/ld.

LD_LIBRARY_PATH (SPARC systems only)

2.4.9

A list of directories to be searched for libraries specified with the -l option. If multiple directories are specified, use a colon (:) to separate directory names. For more information on this, see the ld(1) man page.

LISTIO_PRECISION

2.4.10

Controls the number of digits of precision printed by list-directed output. The LISTIO_PRECISION environment variable can be set to FULL, PRECISION, or YMP80.

- FULL prints full precision (default).
- PRECISION prints x or x+1 decimal digits, where x is value of the PRECISION intrinsic function for a given real value. This is a smaller number of digits, which usually ensures that the last decimal digit is accurate to within 1 unit. This number of digits is usually insufficient to assure that subsequent input will restore a bit-identical floating-point value.
- YMP80 causes list-directed and namelist output of real values to be of the format used in UNICOS 8.0 and previous library versions. This setting is available on Cray PVP systems only. This functionality will be removed in a later release.

MP_DEDICATED (Cray PVP systems and SPARC systems only)

2.4.11

Autotasking variable. The MP_DEDICATED environment variable specifies the environment. If set to 1, it specifies that you are the only user on the system, which allows library scheduling to take advantage of the dedicated environment.

On Cray PVP systems, if MP_DEDICATED is set to 0 or not set at all, slave processors return to the operating system after waiting in user space for 50,000 clock periods.

On SPARC systems, if MP_DEDICATED is set to 0 or not set at all, slave processors return to the operating system after waiting in user space for MP_HOLDTIME clock periods.

On both Cray PVP systems and SPARC systems, if MP_DEDICATED is set to a value other than 0 or 1, the behavior is undefined.

2.4.12 MP_HOLDTIME (Cray PVP systems and SPARC systems only)

Autotasking variable. The MP_HOLDTIME environment variable specifies the number of clock periods (CPs) to hold a processor before giving up the CPU when no parallel work is available.

In nondedicated mode, when MP_HOLDTIME clock periods have gone by without there being any parallel work for a processor, that processor yields its CPU back to the kernel for use by other processes in the system. This yield is temporary; the processors continue to exist, but they do not use as many system resources while yielded. They reacquire a CPU when the next parallel region occurs in the program.

On SPARC systems, a 0 value for MP_HOLDTIME causes the program to completely halt its parallel processors immediately after each parallel region. This has the following effect: the parallel processors consume no system resources at all while executing the sequential regions, but the cost of starting them up again from scratch is accrued on every parallel region (not just the first). Setting MP_HOLDTIME to 0 may be useful when running parallel programs on heavily loaded systems or when running any program whose parallel regions are separated by long periods of sequential execution.

On Cray PVP systems, the default is 50,000 CPs.

On SPARC systems, the default is 25,000 CPs.

2.4.13 MP_SAMPLE (Cray PVP systems only)

Autotasking variable. The MP_SAMPLE environment variable specifies the sample rate at which the ready mask is read when in the hold loop. The default is 150 CPs, which means that a process checks for a task every 150 CPs while it is waiting for parallel work.

2.4.14 MP_SLVSIN (Cray PVP systems only)

Autotasking variable. The MP_SLVSIN environment variable specifies the stack increment, in words, for microtasking or Autotasking slave processes.

2.4.15 MP_SLVSSZ (Cray PVP systems and SPARC systems only)

Autotasking variable.

On Cray PVP systems, the MP_SLVSSZ environment variable specifies the initial task size, in words, for microtasking or Autotasking slave processes.

On SPARC systems, the MP_SLVSSZ environment variable specifies the total stack space for Autotasking slave processes.

2.4.16 MPP_SEGDIR (CRAY T3E systems only)

mppldr(1) directives. This variable contains one or more strings separated by semicolons. Each string may be either a mppldr(1) directive or the name of a file containing mppldr(1) directives.

2.4.17 MSG_FORMAT (Cray PVP systems only)

Describes a format specification similar to that of the C library routine printf; this specification can be used to alter message displays. For more information on this variable, see the *UNICOS Message System Programmer's Guide*, publication SG–2121.

2.4.18 NCPUS (Cray PVP systems only)

Sets the number of CPUs to use when running parallel code.

On Cray PVP systems, the default is the number of hardware CPUs available.

On SPARC systems, the default is 1. This variable has the same effect as the PARALLEL variable. If both NCPUS and PARALLEL are set, NCPUS overrides PARALLEL.

NLSPATH
2.4.19

Message system variable. The NLSPATH environment variable affects compiler interactions with the message system. For more information on this environment variable, see the catopen(3) man page.

NPROC
2.4.20

Maximum number of processes to be run in the background. Compilations can be multiprocessed by setting the NPROC environment variable to a value greater than 1. You can set NPROC to any value, but large values can overload the system. For debugging purposes, NPROC should be set to 1. By default, NPROC is 1.

PARALLEL (SPARC systems only)
2.4.21

Sets the number of CPUs to use when running parallel code.

On SPARC systems, the default is 1. This variable has the same effect as the NCPUS variable. If both NCPUS and PARALLEL are set, NCPUS overrides PARALLEL.

SEGDIR (Cray PVP systems only)
2.4.22

segldr(1) directives. This variable contains one or more strings separated by semicolons. Each string may be either a segldr(1) directive or the name of a file containing segldr(1) directives.

2.4.23 **`SEGLDR`** (Cray PVP systems only)

Path name of the loader.

On Cray PVP systems, the default is `/bin/segldr`. The PE 2.0 release products call `/bin/segldr` from `/opt/ctl/bin/segldr`.

On SPARC systems, this environment variable is included for compatibility with Cray PVP systems. If both the `LD` and `SEGLDR` environment variables are set, the value in `LD` overrides the value in `SEGLDR`. The default is `/usr/ccs/bin/ld`.

2.4.24 **`SEGLDR_M`** (CRAY T3E systems only)

Path name of the CRAY T3E loader. The default is `/opt/ctl/bin/mppldr.`

2.4.25 **`STACKSIZE`** (SPARC systems only)

Stack size, in bytes, for Autotasking slaves.

2.4.26 **`TARGET`** (Cray PVP and CRAY T3E systems only)

Cross-compiling environment variable. *Cross-compiling* is compiling a program on one system to execute on another.

The `TARGET` environment variable allows you to cross-compile from most Cray PVP systems to other Cray Research systems. Cross-compiling from a CRAY T90 system that supports IEEE floating-point arithmetic to other Cray PVP systems is not possible. Cross compiling from a CRAY T3E system to a Cray PVP system is not possible.

The target system is the type of machine upon which the code will be executed. The `TARGET` environment variable recognizes the following values:

Value	Generates code for
`cray-t3e`	CRAY T3E systems
`cray-ts`	CRAY T90 systems without support for IEEE floating-point arithmetic
`cray-ts,ieee`	CRAY T90 systems with support for IEEE floating-point arithmetic
`cray-c90`	CRAY C90 systems
`cray-ymp`	CRAY Y-MP E systems and CRAY Y-MP M90 systems
`cray-el`	CRAY EL systems
`cray-j90`	CRAY J90 systems

See the `target`(1) man page for more information on setting this environment variable.

It is recommended that you load your binary program on the target system. If you load it on the host system, issues over differences between the libraries needed on the target and host systems may arise. The `TARGET` settings do not automatically cause the correct library to be selected. In many cases, host and target libraries will be compatible, but there may be incompatibilities between host and target systems.

The need for a particular set of libraries, as well as the correct specification to identify the set, must be determined for your particular site. If you need to specify a set of libraries contained in a directory, use the `f90 -L` option, as follows:

```
f90 -L /lib/ylib pgm.f   # For a library directory
```

The following example shows how to compile a program on a CRAY EL system and execute it on a CRAY C90 system.

On the CRAY EL system:

1. Set the `TARGET` environment variable to `cray-c90` using the procedure described in this subsection.

2. Compile your program using a command line that includes the `-c` option, as follows:

   ```
   f90 -c bigfile.f
   ```

   The `-c` option disables the load step and returns `bigfile.o`.

On the CRAY C90 system:

1. Move file `bigfile.o` from the CRAY EL system to the CRAY C90 system.

2. Use the `f90` command to link `bigfile.o` with the correct libraries:

   ```
   f90 bigfile.o
   ```

   Note: The `TARGET` environment variable applies until the end of an interactive session, so it affects other commands that use this environment variable to modify different CPU types. You must cancel these settings if you do not want to use them. You can prevent unexpected effects by setting the environment variable in a script or in a `make` file; the setting is effective only within that file.

TMPDIR
2.4.27

Directory to contain temporary files. By default, the CF90 compiler creates temporary files in `/var/tmp`. You can specify a different location by setting `TMPDIR` to your chosen directory. If `TMPDIR` is not a valid directory, the CF90 compiler uses `/var/tmp`.

Compiler Directives & Optimization [3]

Compiler directives are lines inserted into source code that specify actions to be performed by the compiler. They are not Fortran 90 statements.

This section describes all CF90 compiler directives and notes whether or not particular directives are supported on specific platforms. If you specify a compiler directive while running on a system that does not support that particular directive, the compiler generates a message and continues with the compilation.

Table 4 categorizes the CF90 compiler directives according to purpose and platform. It also indicates the pages that contain the main descriptions of the individual directives.

Table 4. Compiler Directives

Purpose and Name	Platforms	Description
Vectorization and tasking:		
`IVDEP`	PVP	page 80
`NEXTSCALAR`	PVP	page 80
`PREFERTASK`	PVP, SPARC	page 81
`PREFERVECTOR`	PVP	page 82
`RECURRENCE`, `NORECURRENCE`	PVP, CRAY T3E, SPARC	page 82
`SHORTLOOP`, `SHORTLOOP128`	PVP	page 85
`TASK`, `NOTASK`	PVP, SPARC	page 86
`UNROLL`, `NOUNROLL`	PVP, CRAY T3E, SPARC	page 86
`VECTOR`, `NOVECTOR`	PVP	page 89
`VFUNCTION`	PVP	page 90
`VSEARCH`, `NOVSEARCH`	PVP	page 91

Section 3

Table 4. Compiler Directives (continued)

Purpose and Name	Platforms	Description
Inlining:		
`INLINE`, `NOINLINE`	PVP, SPARC	page 92
`INLINE NEVER`, `INLINE ALWAYS`	PVP, SPARC	page 92
`MODINLINE`, `NOMODINLINE`	PVP, SPARC	page 96
External names:		
`NAME`	PVP, CRAY T3E, SPARC	page 99
Scalar optimization:		
`ALIGN`	PVP	page 100
`BL`, `NOBL`	PVP, CRAY T3E	page 100
`NO SIDE EFFECTS`	PVP	page 102
`SPLIT`, `NOSPLIT`	CRAY T3E	page 105
`SUPPRESS`	PVP, CRAY T3E, SPARC	page 105
Local use of compiler features:		
`BOUNDS`, `NOBOUNDS`	PVP, CRAY T3E, SPARC	page 107
`FREE`, `FIXED`	PVP, CRAY T3E, SPARC	page 109
`INTEGER`	PVP, CRAY T3E, SPARC	page 110
Flowtracing:		
`FLOW`, `NOFLOW`	PVP	page 111

Table 4. Compiler Directives
(continued)

Purpose and Name	Platforms	Description
Storage:		
AUXILIARY	PVP	page 112
CACHE_ALIGN	CRAY T3E	page 114
COMMON	PVP, CRAY T3E, SPARC	page 115
STACK	PVP, CRAY T3E, SPARC	page 116
SYMMETRIC	CRAY T3E	page 117
TASKCOMMON	PVP, SPARC	page 117

Using compiler directives
3.1

The following subsections describe how to use compiler directives and the effects they have on programs.

Compiler directive lines
3.1.1

A *compiler directive line* begins with the characters `CDIR$` or `!DIR$`. How you specify compiler directives depends on the source form you are using.

- If you are using fixed source form, indicate a compiler directive line by placing the characters `CDIR$` or `!DIR$` in columns 1 through 5. If the compiler encounters a nonblank character in column 6, the line is assumed to be a compiler directive continuation line. Columns 7 and beyond can contain one or more compiler directives. Characters in compiler directives entered in columns beyond the default column width are ignored.

- If you are using free source form, indicate a compiler directive by the characters `!DIR$`, followed by a space, and then one or more compiler directives. If the position following the `!DIR$` contains a character other than a blank, tab, or newline character, the line is assumed to be a continuation line. The `!DIR$` need not start in column 1, but it must be the first text on a line.

In the following example, an asterisk (`*`) appears in column 6 to indicate that the second line is a continuation of the preceding line:

```
!DIR$ Auxiliary
!DIR$*ab
```

The `FIXED` and `FREE` compiler directives must appear alone on a compiler directive line and cannot be continued.

If you want to specify more than one compiler directive on a line, separate each directive with a comma. Some compiler directives require that you specify one or more arguments; when specifying a compiler directive of this type, no other compiler directive can appear on the line.

Spaces can precede, follow, or be embedded within a compiler directive, regardless of source form.

Code portability is maintained despite the use of compiler directives. In the following example, the ! symbol in column 1 causes other compilers to treat the CF90 compiler directive as a comment.

```
A=10.
!DIR$ NOVECTOR
DO 10,I=1,10
...
```

If no format block is shown, directives described in this section consist of the directive name only, with no parameters.

Range and placement of directives
3.1.2

The range and placement of compiler directives is as follows:

- The `CACHE_ALIGN`, `FIXED`, and `FREE` compiler directives can appear anywhere in your source code. All other compiler directives must appear within a program unit.
- The `FLOW`, `INTEGER`, `NOFLOW`, and `STACK` compiler directives each apply only to the program unit in which they appear. The directives that are pairs do not toggle their respective features within the program unit; rather , the last directive of each pair encountered within the program unit applies to the whole program unit.
- The `BL`, `BOUNDS`, `INLINE`, `MODINLINE`, `NOBL`, `NOBOUNDS`, `NOINLINE`, `NOMODINLINE`, `NORECURRENCE`, `NOTASK`, `NOVECTOR`, `RECURRENCE`, `TASK`, and `VECTOR` compiler directives toggle a compiler feature on or off at the point where the directive appears in the code. These directives are in effect until the next directive appears or until the end of either the main program or external procedure is encountered, at which time the command-line settings become the default for the remainder of the compilation.
- Similar to the previous grouping, the `SUPPRESS` compiler directive applies at the point at which it appears, but it causes no continuing condition that can be toggled.
- The `ALIGN`, `IVDEP`, `NEXTSCALAR`, `NOUNROLL`, `NOSPLIT`, `NOVSEARCH`, `PREFERTASK`, `PREFERVECTOR`, `SHORTLOOP`, `SHORTLOOP128`, `SPLIT`, `UNROLL`, and `VSEARCH` compiler directives apply only to the next loop encountered lexically.
- `AUXILIARY`, `COMMON`, `INLINE ALWAYS`, `INLINE NEVER`, `NAME`, `NO SIDE EFFECTS`, `SYMMETRIC`, `TASKCOMMON`, and `VFUNCTION` are declarative compiler directives that do not apply to particular ranges of code but alter the status of entities in ways that affect compilation.

3.1.3 Interaction of compiler directives with the `-x` command-line option

The `f90 -x` option accepts one or more compiler directives as arguments. The compiler directives named as arguments to `-x` are ignored by the CF90 compiler when your input is compiled. If you specify `-x all` or `-x dir`, all compiler directives are ignored.

For more information on the `-x` option, see subsection 2.2.34, page 59.

3.1.4 Command-line options and compiler directives

Some features activated by compiler directives can also be specified on the `f90` command line; a directive applies only to the program unit in which it appears, but the command-line option applies to the entire compilation.

Vectorization, scalar optimization, and tasking can be controlled through both command-line options and compiler directives. If a compiler optimization feature is disabled by default or is disabled by an argument to the `f90 -O` option, the associated `!DIR$` directives are ignored. The following list shows CF90 compiler optimization features, related command-line options, and related compiler directives.

Specifying the `-O 0` option on the command line disables all optimization; all scalar optimization, vectorization, and tasking compiler directives are ignored. The arguments to the `-O` option that specify particular types of optimization also affect compiler directives, as follows:

- Specifying the `-O scalar0` option disables scalar optimization and causes the compiler to ignore all scalar optimization and all vectorization directives.
- Specifying the `-O vector0` option causes the compiler to ignore all vectorization directives. Specifying the `NOVECTOR` compiler directive in a program unit causes the compiler to ignore subsequent compiler directives in that program unit that may specify vectorization.

  Note: On CRAY T3E systems and SPARC systems, `vector0` is in effect at all times, and no other level of vectorization is recognized.

- Specifying the `-O task0` option disables tasking and causes the compiler to ignore tasking compiler directives and Autotasking directives, which are described in section 4, which begins on page 119.

Compiler directive syntax

3.2

The following subsections describe compiler directive syntax and the effects of compiler directives on CF90 programs.

Vectorization and tasking directives

3.2.1

The following subsections describe the compiler directives used to control vectorization and tasking, which are as follows:

- `IVDEP`
- `NEXTSCALAR`
- `PREFERTASK`
- `PREFERVECTOR`
- `RECURRENCE`, `NORECURRENCE`
- `SHORTLOOP`, `SHORTLOOP128`
- `TASK`, `NOTASK`
- `UNROLL`, `NOUNROLL`
- `VECTOR`, `NOVECTOR`
- `VFUNCTION`
- `VSEARCH`, `NOVSEARCH`

The `-O 0`, `-O scalar0`, `-O task0`, and `-O vector0` options on the `f90`(1) command override these compiler directives.

Section 3

3.2.1.1 Ignoring dependencies: IVDEP (Cray PVP systems only)

When the IVDEP directive appears before a loop, the compiler ignores vector dependencies, including explicit dependencies, in any attempts to vectorize the loop. IVDEP applies to the first DO loop or DO WHILE loop that follows the directive. The directive applies to only the first loop that appears after the directive within the same program unit. Whether or not IVDEP is used, conditions other than vector dependencies can inhibit vectorization. The format for this compiler directive is as follows:

```
!DIR$ IVDEP [SAFEVL=vlen ]
```

vlen Specifies a vector length in which no dependency will occur. If *vlen* is not specified, the vector length used is the maximum possible for the target machine.

IVDEP is ignored if vectorization is disabled.

When the CF90 compiler vectorizes a loop, it may reorder the statements in the source code to remove vector dependencies. When IVDEP is specified, the statements in the loop are assumed to contain no dependencies as written, and the CF90 compiler does not reorder loop statements.

Section 3

3.2.1.2 Specifying scalar processing: NEXTSCALAR (Cray PVP systems only)

The NEXTSCALAR directive disables vectorization for the first DO loop or DO WHILE loop that follows the directive. The directive applies to only one loop, the first loop that appears after the directive within the same program unit. NEXTSCALAR is ignored if vectorization has been disabled. The format for this compiler directive is as follows:

```
!DIR$ NEXTSCALAR
```

3.2.1.3 Designate nested loops: PREFERTASK (Cray PVP systems and SPARC systems only)

This directive allows loops with large iteration counts to be considered as candidates for tasking. The compiler analyzes loops that follow a PREFERTASK directive to determine whether the loop is suitable for Autotasking.

The PREFERTASK directive disables threshold checking. For more information on threshold checking, see subsection 2.2.20.32, page 45.

This directive can be used if there is more than one loop in the nest that can be autotasked. Autotasking must be enabled for this directive to take effect. The format for this compiler directive is as follows:

```
!DIR$ PREFERTASK
```

In the following example, both loops can be autotasked, but the PREFERTASK directive directs the compiler to autotask the inner DO J loop. Without the directive and without any knowledge of N and M, the compiler would task the outer DO I loop. With the directive, the loops are interchanged, to increase parallel granularity, and the resulting outer DO J loop is autotasked.

```
      DO I = 1, N
!DIR$ PREFERTASK
        DO J = 1, M
          E(J,I) = F(J,I) + G(J,I)
        END DO
      END DO
```

3.2.1.4 Designate loop nest for vectorization: PREFERVECTOR (Cray PVP systems only)

For cases in which the compiler could vectorize more than one loop, the `PREFERVECTOR` compiler directive indicates that the loop following the directive should be vectorized.

This directive can be used if there is more than one loop in the nest that could be vectorized. The format for this compiler directive is as follows:

```
!DIR$ PREFERVECTOR
```

In the following example, both loops can be vectorized, but the compiler generates vector code for the outer `DO I` loop. Note that the `DO I` loop is vectorized even though the inner `DO J` loop was specified with an `IVDEP` directive.

```
!DIR$ PREFERVECTOR
      DO I = 1, N
!DIR$ IVDEP
        DO J = 1, M
          A(I) = A(I) + B(J,I)
        END DO
      END DO
```

3.2.1.5 Indicating reduction loops: RECURRENCE, NORECURRENCE

The effect of the `RECURRENCE` and `NORECURRENCE` directives differs depending on your platform. Regardless of platform, however, the formats for these directives are as follows:

```
!DIR$ RECURRENCE
!DIR$ NORECURRENCE
```

The following subsections describe the effects of the `RECURRENCE` and `NORECURRENCE` directives on different systems.

Using RECURRENCE and NORECURRENCE on Cray PVP systems

3.2.1.5.1

The RECURRENCE and NORECURRENCE directives control vectorization for all reduction loops within a program unit and override the -O recurrence option on the f90(1) command. The RECURRENCE directive causes loops that contain reductions to be vectorized; the NORECURRENCE directive causes some loops that contain reductions to go unvectorized. Either directive applies until the end of a program unit or until the opposite directive is encountered.

A *reduction loop* is a loop that contains at least one statement that reduces an array to a scalar value by doing a cumulative operation on many of the array elements. This involves including the result of the previous iteration in the expression of the current iteration.

The NORECURRENCE directive disables vectorization of any loop that contains a reduction in which the order of evaluation is numerically significant. The specific reductions that are disabled are floating point, double precision, complex summation and product reductions, and alternating value computations.

Example 1: The NORECURRENCE directive disables vectorization of the following loop:

```
!DIR$ NORECURRENCE
      SUM = 0.0
      DO I = 1,1000
           SUM = SUM + A(I)
      END DO
```

Section 3

Example 2: In the following code, an alternating value computation is not strictly a reduction, but it is included under this heading because the vector version may generate more precise results than the scalar version. The loop in this code fragment would vectorize with the CF90 compiler default settings, but because of the NORECURRENCE directive, it does not vectorize.

```
      SUM     = 0.0
      PRODUCT = 1.0
      TOGGLE  = 5.0
!DIR$ NORECURRENCE
      DO I = 1,1000
        SUM     = SUM + A(I)      ! Summation reduction
        PRODUCT = PRODUCT * A(I)  ! Product reduction
        TOGGLE  = 5.0 - TOGGLE    ! Alternating value
        B(I)    = TOGGLE          !        computation
      END DO
```

Both directives are ignored if vectorization is disabled or if `-O norecurrence` is specified on the `f90` command line.

Section 3

3.2.1.5.2 Using `RECURRENCE` and `NORECURRENCE` on CRAY T3E systems and SPARC systems

The `RECURRENCE` and `NORECURRENCE` directives convert floating-point operations that involve multiplication or exponentiation by an induction variable to be a series of additions or multiplications of a value. This produces faster code but can create numeric differences.

These directives apply to all reduction loops within a program unit, and they override the `-O recurrence` option on the `f90`(1) command. The `RECURRENCE` directive rewrites multiplication operations. Either directive applies until the end of a program unit or until the opposite directive is encountered.

The `NORECURRENCE` directive stops the compiler from rewriting multiplication operations for loops in which the order of evaluation is numerically significant.

Both directives are ignored if `-O norecurrence` is specified on the `f90` command line or if scalar optimization has been disabled.

3.2.1.6 Indicating loops with low trip counts: `SHORTLOOP`, `SHORTLOOP128` (Cray PVP systems only)

The `SHORTLOOP` directive, used before a `DO` or `DO WHILE` loop with a low trip count, allows the compiler to generate code that improves program performance by eliminating run-time tests for determining whether a vectorized `DO` loop has been completed. The formats for these compiler directives are as follows:

```
!DIR$ SHORTLOOP
!DIR$ SHORTLOOP 128
```

You can specify either of the preceding formats.

- If you specify `!DIR$ SHORTLOOP`, the loop trip count must be in the range 1 ≤ *trip_count* ≤ 64. If *trip_count* exceeds 64, results are unpredictable.
- If you specify `!DIR$ SHORTLOOP 128`, the loop trip count must be in the range 1 ≤ *trip_count* ≤ 128. If *trip_count* exceeds 128, results are unpredictable.

`SHORTLOOP` is ignored in the following cases:

- If vectorization is disabled.
- If the code in question is an array syntax assignment statement.
- If the loop trip count is known at compile time and is greater than the target machine's vector length. The vector length of CRAY C90 systems and CRAY T90 systems is 128. The vector length of all other Cray PVP systems is 64.

Section 3

3.2.1.7 Enabling and disabling tasking: `TASK` and `NOTASK` (Cray PVP systems and SPARC systems only)

The `NOTASK` compiler directive suppresses compiler attempts to task loops or array syntax statements and disables recognition of Autotasking directives. `NOTASK` takes effect at the next statement and applies to the rest of the program unit unless it is superseded by a `TASK` directive. These directives are disabled if tasking is disabled. The formats for these compiler directives are as follows:

```
!DIR$ TASK
!DIR$ NOTASK
```


When `!DIR$ NOTASK` has been used within the same program unit, `!DIR$ TASK` causes the compiler to resume its attempts to task loops. After a `TASK` directive is specified, the compiler again attempts to autotask loops and array syntax statements and `!MIC$` directives are again recognized.

The `TASK` directive affects subsequent loops. The `NOTASK` directive also affects subsequent loops, but if it is specified within the body of a loop, it affects the loop in which it is contained and all subsequent loops.

3.2.1.8 Loop unrolling: `UNROLL` and `NOUNROLL`

Loop unrolling can improve program performance by revealing cross-iteration memory optimization opportunities such as read-after-write and read-after-read. The effects of loop unrolling also include:

- Improved loop scheduling by increasing basic block size
- Reduced loop overhead
- Improved chances for cache hits

The formats for these compiler directives are as follows:

```
!DIR$ UNROLL [n]
!DIR$ NOUNROLL
```

n Specifies the total number of loop body copies to be generated. *n* must be a positive integer.

If you specify a value for *n*, the compiler does not attempt to determine the number of copies to generate based on the number of inner loops in the loop nest.

The subsequent `DO` loop is not unrolled if you specify `UNROLL0`, `UNROLL1`, or `NOUNROLL`.

The `UNROLL` compiler directive should be placed immediately before the `DO` statement of the loop that should be unrolled.

Warning: If placed prior to a noninnermost loop, the `UNROLL` directive asserts that the following loop has no dependencies across iterations of that loop. If dependencies exist, incorrect code could be generated.

Note: `UNROLL` compiler directives are ignored when `-O unroll0` is specified on the command line. No unrolling is the default setting, so you must specify `-o unroll1` or `-o unroll2` to enable loop unrolling. For more information on the unrolling options, see subsection 2.2.20.33, page 45.

The `UNROLL` compiler directive can be used only on loops whose iteration counts can be calculated before entering the loop. If `UNROLL` is specified on loop that is not the innermost loop in a loop nest, the inner loops must be nested perfectly. That is, all loops in the nest can contain only 1 loop, and the innermost loop can contain work.

The `NOUNROLL` compiler directive inhibits loop unrolling and overrides a `-o unroll2` command line specification.

Example 1: Assume that the outer loop of the following nest will be unrolled by two:

```
!DIR$ UNROLL 2
      DO I = 1, 10
        DO J = 1,100
              A(J,I) = B(J,I) + 1
        END DO
      END DO
```

With outer loop unrolling, the compiler produces the following nest, in which the two bodies of the inner loop are adjacent to each other:

```
      DO I = 1, 10, 2
        DO J = 1,100
              A(J,I) = B(J,I) + 1
        END DO
        DO J = 1,100
              A(J,I+1) = B(J,I+1) + 1
        END DO
      END DO
```

The compiler *jams* , or *fuses* , the inner two loop bodies together, producing the following nest:

```
      DO I = 1, 10, 2
        DO J = 1,100
              A(J,I)   = B(J,I) + 1
              A(J,I+1) = B(J,I+1) + 1
        END DO
      END DO
```

Example 2: Outer loop unrolling is not always legal because the transformation can change the semantics of the original program. For example, unrolling the following loop nest on the outer loop would change the program semantics because of the dependency between `A(...,I)` and `A(...,I+1)`:

```
!DIR$ UNROLL 2
      DO I = 1, 10
        DO J = 1,100
              A(J,I) = A(J-1,I+1) + 1
        END DO
      END DO
```

3.2.1.9 Enabling and disabling vectorization: `VECTOR` and `NOVECTOR` (Cray PVP systems only)

The `NOVECTOR` directive suppresses compiler attempts to vectorize loops. `NOVECTOR` takes effect at the beginning of the next loop and applies to the rest of the program unit unless it is superseded by a `VECTOR` directive. These directives are ignored if vectorization or scalar optimization have been disabled. The formats for these compiler directives are as follows:

```
!DIR$ VECTOR
!DIR$ NOVECTOR
```

When `!DIR$ NOVECTOR` has been used within the same program unit, `!DIR$ VECTOR` causes the compiler to resume its attempts to vectorize loops. After a `VECTOR` directive is specified, vectorization is attempted for loops with a trip count of 3 or more, or with an unknown trip count.

The `VECTOR` directive affects subsequent loops. The `NOVECTOR` directive also affects subsequent loops, but if it is specified within the body of a loop, it affects the loop in which it is contained and all subsequent loops.

3.2.1.10 Specifying a vectorizable function: `VFUNCTION` (Cray PVP systems only)

The `VFUNCTION` directive declares that a vector version of an external function exists. The `VFUNCTION` directive must precede any statement function definitions or executable statements in a program. `!DIR$ VFUNCTION` cannot be specified for internal or module procedures. `!DIR$ VFUNCTION` cannot be specified for functions within interface blocks. The format for this compiler directive is as follows:

```
!DIR$ VFUNCTION f[,f]...
```

f Symbolic name of a vector external function. The maximum length is 29 characters because the `%` character is added at the beginning and end of the name as part of the calling sequence. If the function is named `FUNC`, the CAL vector version is spelled `%FUNC%`. (The scalar version is `FUNC%`.)

The following rules and recommendations apply to any function *f* named as an argument in a `VFUNCTION` directive:

- *f* cannot be declared in an `EXTERNAL` statement or have its interface specified in an interface body.
- *f* must be written in CAL and must use the call-by-register sequence.
- Arguments to *f* must be either vectorizable expressions or scalar expressions; array syntax and array expressions are not allowed.
- A call to *f* can pass a maximum of seven single-word items or one four-word item (complex (`KIND=KIND(0.0D0)`)). No structures or character arguments can be passed. These can be mixed in any order with a maximum of seven words total.
- *f* should not change the value of its arguments or variables in common blocks or modules. Any changed value should be for variables that are distinct from the arguments.
- *f* should not reference variables in common blocks or modules that are also used by a program unit in the calling chain.

- A call to *f* cannot occur within a `WHERE` statement or `WHERE` block.
- *f* must not have side effects or perform I/O.

Arguments to *f* are sent to the V registers that have numbers that match the arguments' ordinal numbers in the argument list: `X=VFUNC(`*v1*`,`*v2*`,`*v3*`,`*v4*`).` (The scalar version uses the same convention with the S registers.)

If the argument list for *f* contains both scalar and vector arguments in a vector loop, the scalar arguments are broadcast into the appropriate vector registers. If all arguments are scalar or the function reference is not in a vector loop, *f* is called with all arguments passed in S registers.

Vectorizing search loops: `VSEARCH` and `NOVSEARCH` (Cray PVP systems only)
3.2.1.11

The `VSEARCH` compiler directive indicates that the compiler should vectorize the search loop that follows. The `NOVSEARCH` directive disables vectorization of search loops. The formats for these directives are as follows:

```
!DIR$ VSEARCH
!DIR$ NOVSEARCH
```

A *search loop* is one that can be exited by means of an `IF` statement or `EXIT` statement.

The `VSEARCH` and `NOVSEARCH` directives are enabled only when vectorization is enabled. These directives override `-o vsearch`, but they do not override `-o novsearch`.

Section 3

3.2.2 Inlining directives (Cray PVP systems and SPARC systems only)

The inlining compiler directives allow you to specify whether the compiler should attempt to inline certain subprograms or procedures. They are as follows:

- `INLINE`, `NOINLINE`
- `INLINE NEVER`, `INLINE ALWAYS`
- `MODINLINE`, `NOMODINLINE`

These directives work in conjunction with the following command-line options:

- `-O inline`*n*, described in subsection 2.2.20.17, page 36.
- `-O inlinefrom`, described in subsection 2.2.20.21, page 39.
- `-O modinline` and `-O nomodinline`, described in subsection 2.2.20.24, page 41.

Section 3

3.2.2.1 Disable or enable inlining for a block of code: `INLINE` and `NOINLINE` (Cray PVP systems and SPARC systems only)

The `INLINE` and `NOINLINE` compiler directives control whether automatic inlining analysis is performed. One of these directives remains in effect until the opposite directive is encountered. These directives are ignored if `-O inline0` is in effect.

```
!DIR$ INLINE
!DIR$ NOINLINE
```

3.2.2.2 Specify inlining for a procedure: `INLINE NEVER` and `INLINE ALWAYS` (Cray PVP systems and SPARC systems only)

The `INLINE ALWAYS` directive forces attempted inlining of specified procedures. The `INLINE NEVER` directive suppresses inlining of specified procedures. An `INLINE NEVER` directive overrides an `INLINE ALWAYS` directive.

Certain general criteria must be met in order for any inlining to occur:

- The procedure to be inlined must be available in an inlinable form. This can be accomplished in any of the following ways:
  - The procedure is in the same file being compiled as the call site. A *call site* is the place at which a `CALL` statement that names the procedure appears or is the place at which a function reference that names the procedure appears.
  - If the source is a module procedure that is use associated into the calling procedure, an inlinable form of the procedure is available if the module was compiled with `-O modinline`.
  - The procedure is available in a file or directory that was compiled using the `-O inlinefrom` option.
- The program unit that calls the procedure must have been compiled with `-O inline1`, `-O inline2`, `-O inline3`, or `-O inlinefrom` specified on the `f90` command line.
- The procedure being called must not be specified in a `!DIR INLINE NEVER` directive.
- The call site must not be within the range of a `!DIR$ NOINLINE` directive.

The formats for these directives are as follows:

```
!DIR$ INLINE ALWAYS procedure [, procedure ]...
!DIR$ INLINE NEVER procedure [, procedure ]...
```

procedure The name of a procedure.

Section 3

The following rules determine the scope of these directives:

- If the directive is specified for the definition of the procedure, the directive applies wherever the procedure is called and inlining is being attempted.
- If the directive is specified for a referenced procedure, the directive applies only to the scope of the program unit containing the directive.
- For a recursive procedure, if the directive is specified for the procedure, it is treated as a definition and nonrecursive references are inlined or not inlined depending on the directive. A recursive procedure cannot be inlined indefinitely because this would create an infinite loop of inlining.

An error message is issued if `INLINE NEVER` and `INLINE ALWAYS` are specified for the same procedure in the same program unit. A caution message is issued if `!DIR$ INLINE NEVER` is specified for the definition of a program unit, but `!DIR$ INLINE ALWAYS` is specified for the reference of a program unit.

Example: the following file is compiled with `-O inline1`:

```
      SUBROUTINE S()
!DIR$ INLINE ALWAYS S  ! This says inline S at all
                       ! calls (if normal
                       ! inlining criteria are met.)
         ...
      END SUBROUTINE

      SUBROUTINE T
!DIR$ INLINE NEVER S   ! Do NOT inline any calls to S
                       ! in subroutine T.
      CALL S()
      ...
      END SUBROUTINE

      SUBROUTINE V
!DIR$ NOINLINE
      CALL S()        ! Do not inline this call to S.
!DIR$ INLINE
      CALL S()        ! Inline this call to S.
      ...
      END SUBROUTINE

      SUBROUTINE W
      CALL S()        ! Inline this call to S.
      ...
      END SUBROUTINE
```

In the preceding example, all references to `S` are inlined if they meet the following criteria:

- The source for `S` is available (or if `S` is a module procedure and is use associated, `S`'s parent module must have been compiled with `-O modinline`).
- The call to `S` is not within the range of a `!DIR$ NOINLINE` directive.

- `S` has not been specified with `!DIR$ INLINE NEVER S` in the program unit where `S` is being called.
- Inlining is activated in some form for the program unit containing the call to `S`.

In this example, the call to `S` is not inlined in subroutine `T` because `S` has been specified with an `INLINE NEVER` directive.

Request inlining for a module: `MODINLINE` and `NOMODINLINE` (Cray PVP systems and SPARC systems only)
3.2.2.3

The `MODINLINE` directive allows all module procedures to be inlined. It causes the compiler to store more information in the module information table, which allows inlining of any module procedure declared within the scope of the directive.

The `NOMODINLINE` directive causes the compiler to generate a smaller module information table. Any module procedure definitions specified within the scope of this directive cannot be inlined later even if they are incorporated into a different compilation by a `USE` statement.

The formats for these directives are as follows:

```
!DIR$ MODINLINE
!DIR$ NOMODINLINE
```

These directives are in effect for the scope of the program unit in which they are specified, including all contained procedures. If the directive is specified in a contained procedure, the contained procedure's directive overrides the containing procedure's directive. This directive follows the Fortran 90 scoping rules.

The compiler generates an error message if these directives are specified outside of a module.

To use inlinable module procedures, the module being use associated must be compiled with inlining enabled and the module procedures being called must be compiled with `-O modinline`.

The `-O inline`*n* option affects inlinable module procedures as follows:

Option	Behavior
`-O inline0`	No module procedures are inlined.
`-O inline1`	The module procedure will be inlined if the following criteria are met: – The procedure is named in an `INLINE ALWAYS` directive. – The module procedure name is referenced. – The module procedure is within the scope of a `!DIR$ INLINE` directive. – The criteria listed in subsection 2.2.20.17, page 36, are met.
`-O inline2`, `-O inline3`	The compiler attempts to inline the module procedure.

Section 3

Example: The following file is compiled as `f90 -O modinline file.f`:

```
      MODULE BEGIN
!DIR$ MODINLINE
      ...
      CONTAINS
        SUBROUTINE S()      ! Uses SUBROUTINE S's !DIR
!DIR$   NOMODINLINE
        ...
        CONTAINS
          SUBROUTINE INSIDE_S()  ! Uses SUBROUTINE S's !DIR
          ...
          END SUBROUTINE
        END SUBROUTINE

        SUBROUTINE T()           ! Uses MODULE BEGIN's !DIR
        ...
        CONTAINS
          SUBROUTINE INSIDE_T()  ! Uses MODULE BEGIN's !DIR
          ...
          END SUBROUTINE
            SUBROUTINE MORE_INSIDE_T
!DIR$       NOMODINLINE
            ...
            END SUBROUTINE
        END SUBROUTINE
      END MODULE
```

In the preceding example, the subroutines are affected as follows:

- Inlining information is not produced for `S`, `INSIDE_S`, or `MORE_INSIDE_T`.
- Inlining information is produced for `T` and `INSIDE_T`.

Section 3

External name mapping directive: NAME
3.2.3

The NAME directive allows you to specify a case-sensitive external name, or a name that contains characters outside of the Fortran character set, in a Fortran program. Rules for Fortran naming do not apply to the *external_name* string; any character sequence is valid. You can use this directive, for example, when writing calls to C routines. The case-sensitive external name is specified on the NAME directive, in the following format:

```
!DIR$ NAME (fortran_name ="external_name "
  [, fortran_name ="external_name "] ... )
```

fortran_name The name used for the object throughout the Fortran program.

external_name The external form of the name.

Example:

```
      PROGRAM MAIN
!DIR$ NAME (FOO="XyZ")
      CALL FOO            ! XyZ is really being called
      END PROGRAM
```

Scalar optimization directives
3.2.4

The ALIGN, BL, NOBL, NO SIDE EFFECTS, NOSPLIT, SPLIT and SUPPRESS compiler directives control aspects of scalar optimization. The following subsections describe these compiler directives.

Section 3

3.2.4.1 Aligning loops on buffer boundaries: ALIGN (Cray PVP systems only)

The `ALIGN` directive causes a block of code to begin on an instruction buffer boundary . Placing the directive in front of a `DO` or `DO WHILE` statement causes the body of the loop to begin on an instruction buffer boundary. Placing the directive immediately before a statement containing a referenced label (for example, the first statement of an `IF` loop) causes the generated code for that statement to begin on an instruction buffer boundary . The format for this compiler directive is as follows:

```
!DIR$ ALIGN
```

The `ALIGN` directive is useful when program execution is dominated by specific small blocks of code. If such a block crosses a buffer boundary, the overhead caused by frequent reloading of instruction buffers hurts program performance. By aligning such a dominant loop on a buffer boundary, the `ALIGN` directive can decrease this overhead.

3.2.4.2 Bottom loading operands: BL and NOBL (Cray PVP systems and CRAY T3E systems only)

When scalar optimization is enabled, the `NOBL` directive causes the use of safe bottom loading within loops; the `BL` directive causes the compiler to use standard bottom loading. Safe bottom loading involves a small cost in performance but removes a possible cause of run-time errors (indicated by an `Operand range error` message). Either directive overrides the `-O bl` option on the `f90`(1) command. The formats for these compiler directives are as follows:

```
!DIR$ BL
!DIR$ NOBL
```

Bottom loading is an optimization technique that improves program performance and, typically, does not cause an error. This subsection describes the purpose of bottom loading and why it can cause an error.

Standard bottom loading, which is used only on eligible scalar loops, consists of prefetching operands during each iteration of a loop for use in the next iteration, so that the operand is available as soon as the first loop instruction executes. A prefetch is performed even during the final loop iteration, before the loop's final jump test has been performed.

The final prefetch, using the next computed address, can access a location outside the array. If this address is outside the user program area, your program can fail, generating an operand range error. An out-of-bounds address typically is accessed in the following situations:

- A loop with a very large increment, for example:

  ```
  DO I=10,1000000,100000
  ... = A(I)
  ```

- An array reference with a variant index that is other than the first subscript, for example:

  ```
  DIMENSION A(1000,1000)
  DO J=1,1000
  ... = A(100,J)
  ```

  An increment of this size might occur unnoticed when it is defined by a variable.

Safe bottom loading disables operand prefetching on the final loop iteration. The run-time test required to detect the final iteration entails a cost in performance but allows bottom loading to be performed on previous iterations. Despite the extra cost, safe bottom loading is faster than no bottom loading. You can enable safe bottom loading by specifying the `-O nobl` option on the `f90` command or the `NOBL` compiler directive. When the `-O scalar0` or `-O 0` option is specified on the `f90` command, no bottom loading is performed.

> **Note:** When a range error is caused by bottom loading, this does not indicate either a fault in your source program or a compiling error. The error results from an attempt to go outside the range of an array address; the problem is not caused by exceeding the array subscript bounds (as specified in your source code), so it cannot be detected.

Section 3

The scope of the BL and NOBL directives is a single program unit. Either directive applies for the remainder of the program unit or until the appearance of the other directive in the same program unit. Both directives can be specified in a single program unit, and both are ignored if scalar optimization is disabled by the -O scalar0 or -O 0 options on the f90 command.

Example:

```
...
REAL ARRAY(0:M)
...
INC = IRESULT
! Potentially large value
...
DO 10 I=0,M,INC
...
Y = X*ARRAY(I) + ...
...
! Fetch next operand, ARRAY(I+INC)
10 CONTINUE
```

Section 3

3.2.4.3 Determining register storage: NO SIDE EFFECTS (Cray PVP systems only)

The NO SIDE EFFECTS directive allows the compiler to keep information in registers across a single call to a subprogram without reloading the information from memory after returning from the subprogram. The directive is not needed for intrinsic functions and VFUNCTIONs.

NO SIDE EFFECTS declares that a called subprogram does not redefine any variables that meet the following conditions:

- Local to the calling program
- Passed as arguments to the subprogram
- Accessible to the calling subprogram through host association
- Declared in a common block or module
- Accessible through USE association

The format for this compiler directive is as follows:

```
!DIR$ NO SIDE EFFECTS f [, f]...
```

f Symbolic name of a subprogram that the user ensures to have no side effects. *f* must not be the name of a dummy procedure, module procedure, or internal procedure.

A procedure declared `NO SIDE EFFECTS` should not define variables in a common block or module shared by a program unit in the calling chain. All arguments should be intent `IN`; that is, the procedure must not modify its arguments. If these conditions are not met, results are unpredictable.

The `NO SIDE EFFECTS` directive must appear in the specification part of a program unit and must appear before the first executable statement.

The compiler may move invocations of a `NO SIDE EFFECTS` subprogram from the body of a `DO` loop to the loop preamble if the arguments to that function are invariant in the loop. This may affect the results of the program, particularly if the `NO SIDE EFFECTS` subprogram calls functions such as the random number generator or the real-time clock.

3.2.4.4 Loop splitting: `SPLIT` and `NOSPLIT` (CRAY T3E systems only)

Loop splitting improves performance by making best use of the six stream buffers of CRAY T3E systems. It achieves this by splitting an inner loop into a set of smaller loops, each of which allocates no more than six stream buffers, thus avoiding stream buffer thrashing. The stream buffer feature reduces memory latency and increases memory bandwidth by prefetching for long, small-strided sequences of memory references.

```
!DIR$ SPLIT
!DIR$ NOSPLIT
```

The `SPLIT` compiler directive should be placed immediately before the `DO` statement of the loop that should be split. The `SPLIT` directive asserts that the loop can profit by splitting. It will not cause incorrect code.

The compiler splits the loop only if it is safe. Generally, a loop is safe to split under the same conditions that a loop is vectorizable. The compiler only splits inner loops. The compiler may not split some loops with conditional code.

The `SPLIT` directive also causes the original loop to be stripmined. This is done to increase the potential for cache hits between the resultant smaller loops.

Loop splitting can reduce the execution time of a loop by as much as 40%. Candidates for loop splitting can have trip counts as low as 40. They must also contain more than six different memory references with strides less than 16.

Note that there is a slight potential for increasing the execution time of certain loops. Loop splitting also increases compile time, especially when loop unrolling is also enabled.

The `NOSPLIT` compiler directive inhibits loop splitting and overrides a `-O split2` command line specification. For more information on the `-O` loop splitting options, see subsection 2.2.20.30, page 44.

Example:

```
!DIR SPLIT
      DO I=1,1000
         A(I) = B(I) * C(I)
         T = D(I) + A(I)
         E(I) = F(I) + T * G(I)
         H(I) = H(I) + E(I)
      ENDDO
```

First, the compiler generates the following loop (notice the expansion of the scalar temporary `T` into the compiler temporary array `TA`):

```
DO I=1,1000
   A(I) = B(I) * C(I)
   TA(I) = D(I) + A(I)
ENDDO
DO I=1,1000
   E(I) = F(I) * TA(I) * G(I)
   H(I) = H(I) + E(I)
ENDDO
```

Finally , the compiler stripmines the loops to increase the potential for cache hits and reduce the size of arrays created for scalar expansion:

```
DO I1=1,1000,256
   I2 = MIN(I1+255, 1000)
DO I=I1,I2
   A(I) = B(I) * C(I)
   TA(I-I1+1) = D(I) + A(I)
ENDDO
DO I=I1,I2
   E(I) = F(I) * TA(I-I1+1) * G(I)
   H(I) = H(I) + E(I)
ENDDO
ENDDO
```

Section 3

3.2.4.5 Suppress scalar optimization momentarily: `SUPPRESS`

The `SUPPRESS` directive suppresses scalar optimization for all variables or only for those specified at the point where the directive appears. This often prevents or adversely affects vectorization of any loop that contains `SUPPRESS`. The format for this compiler directive is as follows:

`!DIR$ SUPPRESS` [*var* [, *var*] ...]

var Variable that is to be stored to memory. If no variables are listed, all variables in the program unit are stored.

At the point at which `!DIR$ SUPPRESS` appears in the source code, variables in registers are stored to memory (to be read out at their next reference), and expressions containing any of the affected variables are recomputed at their next reference after `!DIR$ SUPPRESS`. The effect on optimization is equivalent to that of an external subroutine call with an argument list that includes the variables specified by `!DIR$ SUPPRESS` (or, if no variable list is included, all variables in the program unit).

`SUPPRESS` takes effect only if it is on an execution path. Optimization proceeds normally if the directive path is not executed because of a `GOTO` or `IF`.

Example:

```
      SUBROUTINE SUB (L)
      LOGICAL L
      A = 1.0           ! A is local
      IF (L) THEN
!DIR$ SUPPRESS           ! Has no effect if L is false
        CALL ROUTINE()
      ELSE
      PRINT *, A
      ENDIF
      END
```

In this example, optimization replaces the reference to `A` in the `PRINT` statement with the constant `1.0`, even though `!DIR$ SUPPRESS` appears between `A=1.0` and the `PRINT` statement. The `IF` statement can cause the execution path to bypass `!DIR$ SUPPRESS`. If `SUPPRESS` appears before the `IF` statement, `A` in `PRINT *` is not replaced by the constant `1.0`.

Section 3

Local use of compiler features
3.2.5

The `FREE` and `FIXED` directives allow you to specify the source form used within a source file. The `INTEGER` directive determines the integer length used in a program unit. The `BOUNDS` and `NOBOUNDS` directives control array bounds checking.

The `-f` and `-i` command-line options apply to an entire compilation, but these compiler directives override any command line specifications for source form or integer length. The following subsections describe these compiler directives.

Array bounds checking: `BOUNDS` and `NOBOUNDS`
3.2.5.1

Array bounds checking provides a check of most array references at both compile time and run time to ensure that each subscript is within the array's declared size.

The `-R` command-line option controls bounds checking for a whole compilation. The `BOUNDS` and `NOBOUNDS` directives toggle the feature on and off within a program unit. Either directive can specify particular arrays or can apply to all arrays. The formats for these compiler directives are as follows:

```
!DIR$ BOUNDS [arnm [,arnm ]...]
!DIR$ NOBOUNDS [arnm [,arnm ]...]
```

arnm The name of an array. The name cannot be a subobject of a derived type. When no array name is specified, the directive applies to all arrays.

`BOUNDS` remains in effect for a given array until the appearance of a `NOBOUNDS` directive that applies to that array, or until the end of the program unit. Bounds checking can be enabled and disabled many times in a single program unit.

Section 3

> **Note:** To be effective, these directives must follow the declarations for all affected arrays; it is suggested that they be placed at the end of a program unit's specification statements, unless they are meant to control particular ranges of code.

The bounds checking feature detects any reference to an array element whose subscript exceeds the array's declared size. For example:

```
      REAL A(10)
C  Detected at compile time:
      A(11) = X
C  Detected at run time if IFUN(M) exceeds 10:
      A(IFUN(M)) = W
```

The compiler generates an error message when it detects an out-of-bounds subscript. If the compiler cannot detect the out-of-bounds subscript (for example, if the subscript includes a function reference), a message is issued for out-of-bound subscripts when your program runs, but the program is allowed to complete execution.

Section 3

Bounds checking does not inhibit vectorization but typically increases program run time. If an array's last dimension declarator is `*`, checking is not performed on the last dimension's upper bound. Arrays in formatted `WRITE` statements are not checked.

> **Note:** Array bounds checking does not prevent operand range errors that result when operand prefetching attempts to access an invalid address outside an array. See subsection 3.2.4.2, page 100, for information on bottom loading.
>
> On Cray PVP systems, bounds checking is needed when very large values are used to calculate addresses for memory references.

If bounds checking detects an out-of-bounds array reference, a message is issued for only the first out-of-bounds array reference in the loop. For example:

```
      DIMENSION A(10)
      MAX = 20
      A(MAX) = 2
      DO 10 I = 1, MAX
         A(I) = I
10    CONTINUE
      CALL TWO(MAX,A)
      END

      SUBROUTINE TWO(MAX,A)
      END
```

The following messages are issued for the preceding program:

```
lib-1961 a.out: WARNING
  Subscript 20 is of out of range for dimension 1 for
  array 'A' at line 3 in file 't.f' with bounds:
  Lower bound is 1
  Upper bound is 10

lib-1962 a.out: WARNING
  Subscript 11:20:1 is of out of range for dimension 1
  for array 'A' at line 5 in file 't.f' with bounds:
  Lower bound is 1
  Upper bound is 10
```

Specifying source form: `FREE` and `FIXED` 3.2.5.2

The `FREE` and `FIXED` directives specify whether the source code in the program unit is written in free source form or fixed source form. The `FREE` and `FIXED` directives override the `-f` option, if specified, on the command line. The formats for these compiler directives are as follows:

```
!DIR$ FREE
!DIR$ FIXED
```


Section 3

These directives apply to the source file in which they appear, and they allow you to switch source forms within a source file.

You can change source form within an INCLUDE file. After the INCLUDE file has been processed, the source form reverts back to the source form that was being used prior to processing of the INCLUDE file.

3.2.5.3 Specifying integer length: INTEGER

The INTEGER directive specifies the integer length to be used in multiplication and division operations within one program unit for default integer types. It does not apply to subsequent program units. The !DIR$ INTEGER directive is allowed only in external program units; it cannot be specified in internal or module procedures. This compiler directive must immediately follow the program unit header statement (PROGRAM, FUNCTION, SUBROUTINE, MODULE, or BLOCKDATA). INTEGER applies at both compile time and run time. The format for this compiler directive is as follows:

```
!DIR$ INTEGER=n
```

n Specify 32, 46, or 64 integer length.

Warning: The INTEGER=46 specification is valid only on Cray PVP systems, and it is obsolescent. If you specify INTEGER=46, a warning message is generated. This value for *n* will be removed in the CF90 3.0 release.

If the INTEGER compiler directive is used in a module, program units that use the module must have the same integer length *n*.

This feature is also available through the -i *intlen* command-line option. For more information on this command-line option, see subsection 2.2.11, page 17.

Section 3

Flowtracing directives (Cray PVP systems only)

3.2.6

The `FLOW` and `NOFLOW` directives control the Flowtrace feature and override the `-e f` or `-d f` options on the `f90(1)` command line. Flowtrace monitors your program during execution and generates a dynamic call tree for your program. It also collects calling and timing information for each called procedure. These directives also control the Perftrace utility. Flowtrace and Perftrace are described in the *UNICOS Performance Utilities Reference Manual*, publication SR–2040.

If the `-e f` option is not specified, `!DIR$ FLOW` activates Flowtrace for the subprogram that contains it. If Flowtrace is activated by the presence of the `-e f` option on the command line, `!DIR$ NOFLOW` deactivates Flowtrace for the subprogram that contains it. The two directives cannot be used to toggle Flowtrace on and off within a program unit. The formats for these compiler directives are as follows:

```
!DIR$ FLOW
!DIR$ NOFLOW
```

These directives apply to the entire program unit in which they appear. If both directives appear in the same program unit, the last one encountered takes effect for the entire program unit.

Storage directives

3.2.7

The following compiler directives specify aspects of storing common blocks, variables, or arrays:

- `AUXILIARY`
- `CACHE_ALIGN`
- `COMMON`
- `STACK`
- `SYMMETRIC`
- `TASKCOMMON`

Section 3

The following subsections describe these compiler directives.

3.2.7.1 Allocating to SSD: AUXILIARY (Cray PVP systems only)

The AUXILIARY directive causes the compiler to allocate arrays and common blocks to the SSD solid-state storage device. This directive is intended for programs with very large data structures. The SSD, configured as secondary memory, functions much the same as main memory and considerably faster than a disk drive. When an entity has been allocated to the SSD, that entity becomes *auxiliary* ; for example, an *auxiliary array.*

This directive must appear after the declaration of auxiliary arrays or common blocks but before the first executable statement in any program unit that references them.

If you want to specify auxiliary data in the specification part of a module, the data must also be declared in a common block.

The format for this compiler directive is as follows:

```
!DIR$ AUXILIARY n[,n]...
```

n Symbolic name of an array to be allocated to the SSD.

Note: Any arrays or scalar variables that are in the same common block as an auxiliary array or are equivalenced to an auxiliary array become auxiliary variables.

Example 1: In the following code example, arrays A and B reside on the SSD. In addition, because array T is also specified in !DIR$ AUXILIARY, array U and variables X, Y, and Z are auxiliary because they appear in common block COMBLK along with T. COMBLK contains a mixture of arrays and scalar variables; individual variables can be allocated to the SSD only as members of a common block that contains an auxiliary array.

```
            DIMENSION A(1000000), B(1000000)
            COMMON /COMBLK/ T(1000000), U(1000000), X, Y, Z
      !DIR$ AUXILIARY A, B, T
```

Section 3

Example 2: In the following code example, element `A(1,1)` is equivalenced to element `X(50,50)` of auxiliary array `X`; therefore, `A` becomes an auxiliary array .

```
      COMMON /COM_BLK/ X(100,100), Y(1000), Z(100,100)
      DIMENSION A(100,100)
      EQUIVALENCE (A(1,1), X(50,50))
!DIR$ AUXILIARY X
```

Restrictions 3.2.7.1.1

The `AUXILIARY` directive has the following restrictions:

- The name of an auxiliary array or scalar variable must not appear in an I/O, `DATA`, or Cray pointer statement.
- If an auxiliary array or scalar variable is passed as an actual argument to a subprogram, the corresponding dummy argument must also have been declared auxiliary. This does not apply to intrinsic functions or to functions declared as `VFUNCTION`s.
- The name of an auxiliary array or scalar variable must not appear as an argument to the `LOC` or `CLOC` intrinsic functions.
- Arrays in blank common or those declared by `TASK COMMON` must not be declared auxiliary.
- Character arrays and arrays in common blocks containing character scalar variables or arrays must not be declared auxiliary.
- Auxiliary array elements and vectors should not be used in signal-handling routines.
- Arrays of structures containing character components or pointer components must not be declared `AUXILIARY`.
- An auxiliary array must not have the attributes `POINTER`, `TARGET`, or `ALLOCATABLE`.

Section 3

3.2.7.1.2 Auxiliary arrays and memory

Auxiliary arrays or scalar variables are read from the SSD solid-state storage device and stored in memory in buffers to read and write to the SSD. Two environment variables, `AUXBUF` and `AUXPAGE`, allow you to specify the size and number of these buffers. Both variables are examined at the beginning of program execution. For more information on these environment variables, see subsection 2.4.2, page 64, and subsection 2.4.3, page 64.

For array syntax statements that need compiler-generated temporary arrays (called *temps*), the temps are always allocated on the heap. This means that if the array syntax involves auxiliary arrays, there may be insufficient space on the heap to create the temps to perform the assignment. Temps are sometimes required when the same array appears on both sides of the equal sign or when a function or subroutine call requires an array temp (either for an argument or for a function result).

Example:

```
      DIMENSION A(100000)
!DIR$ AUXILIARY A
      A(1:100000) = A(100000:1:-1)
```

The preceding is processed in the following form, where `T` is a temporary array:

```
T=A(100000:1:-1)
A = T
```

3.2.7.2 Align on cache line boundaries: `CACHE_ALIGN` (CRAY T3E systems only)

The `CACHE_ALIGN` directive aligns each specified variable on a cache line boundary . This is useful for frequently referenced variables. A *cache* is storage that can be accessed more quickly than conventional memory. A *cache line* is a division within a cache. `CACHE_ALIGN` can be used with work-sharing, data-sharing, and message-passing programming styles. (The work-sharing and data-sharing capabilities are deferred.)

The directive's effect is independent of its position in source code. The format for this compiler directive is as follows:

```
!DIR$ CACHE_ALIGN var [, var] ...
```

var List of common block names or PE-private variable names, separated by commas. If specifying a common block name, the names of the items in the common block need not be specified. A common block name must be in the following form:

/*common_block_name* /

var cannot have the `ALLOCATABLE`, `POINTER`, or `TARGET` attributes, and *var* cannot appear in a Cray `POINTER` statement.

3.2.7.3 Declaring common tasks: `COMMON`

This `COMMON` compiler directive overrides the `-a taskcommon` command line specification for a given procedure. This information is not inherited from the parent scoping unit. All common blocks remain as single-copy common blocks when this compiler directive appears in a procedure. The format for this compiler directive is as follows:

```
!DIR$ COMMON b[,b]...
```

b Common block name. The name must also appear as a common block name in a `COMMON` statement; for information on the `COMMON` statement, see the *CF90 Fortran Language Reference Manual, Volume 1*, publication SR–3902.

Common blocks named in this compiler directive cannot also be named in a `TASKCOMMON` compiler directive. All procedures that reference a common block named in this directive that are compiled with `-a taskcommon` muct declare the block in a `COMMON` directive.

For more information on the `-a taskcommon` command-line specification, see subsection 2.2.1, page 6.

3.2.7.4 Requests stack storage: STACK

The `STACK` directive causes the default storage allocation to be to the stack in the program unit that contains the directive. This directive overrides the `-e v` command-line option in specific program units of a compilation unit. For more information on the `-e v` command-line option, see subsection 2.2.5, page 9.

The format for this compiler directive is as follows:

```
!DIR$ STACK
```


Data specified in the specification part of a module or in a `DATA` statement is always allocated to static storage. This directive has no effect on this static storage allocation.

All `SAVE` statements are honored in program units that also contain a `STACK` directive. This directive does not override the `SAVE` statement.

If the compiler finds a `STACK` directive and a `SAVE` statement without any objects specified in the same program unit, a warning message is issued.

The following rules apply when using this directive:

- It must be specified within the scope of a program unit.
- If it is specified in the specification part of a module, an error message is issued. The `STACK` directive is allowed in the scope of a module procedure.
- If it is specified within the scope of an interface body, an error message is issued.

By default on SPARC systems, any variable in a main program that requires more than 32 bytes (8 words) of storage defaults to static storage. This directive overrides that rule and causes the storage allocation to be stack for these items.

3.2.7.5 Declaring local addressing: SYMMETRIC (CRAY T3E systems only)

The `SYMMETRIC` directive declares that a PE-private stack variable has the same local address on all PEs. This is useful for global addressing using the SHMEM library routines. For more information on the SHMEM library routines, see `intro_shmem(3)`. The format for this compiler directive is as follows:

```
!DIR$ SYMMETRIC [ var [, var ... ]]
```

var A variable that is PE private. Variables that are specified are allocated at the same local address on all PEs. There is an implicit barrier before the first executable statement in the routine.

When specified without the optional *var* argument, all PE-private stack variables are declared symmetric. All local variables are allocated at the same local address on all PEs.

All PEs must participate in the allocation of symmetric stack variables.

3.2.7.6 Declaring local tasks: TASKCOMMON (Cray PVP systems and SPARC systems only)

When multitasking is used, some common blocks may need to be local to a task. The `TASKCOMMON` directive declares all variables in a common block to be local to a task; the common block is referred to as a *task common block*. If multiple tasks execute code containing the same task common block, each task will have a separate copy of the block.

The `TASKCOMMON` directive must appear before the first executable statement of a program unit. Task common blocks must be named. A task common block is allocated at task invocation. The format for this compiler directive is as follows:

```
!DIR$ TASKCOMMON cb[,cb]...
```

cb Common block name. Common blocks named in this directive must not also be named in a `COMMON` compiler directive.

If *cb* is declared as `TASKCOMMON`, *cb* must be declared as `TASKCOMMON` in all program units that reference *cb*.

Using `!DIR$ TASKCOMMON` with the `COMMON` statement is equivalent to using the `TASK COMMON` statement, described in the *CF90 Fortran Language Reference Manual, Volume 3*, publication SR–3905. This directive is recommended because it increases code portability; a different compiler would recognize the `COMMON` statement and ignore the directive, for example:

```
      COMMON /CB1/ A,B  /CB2/ C,D
!DIR$ TASKCOMMON CB1, CB2
```

Arrays in a task common block must not be declared as auxiliary. With these exceptions, these variables can be used like other variables in common storage.

Task common blocks are always given stack allocation, regardless of the kind of allocation used for other data. Unlike other stack variables, task common variables can be initially defined in a `DATA` statement or a type declaration statement. Such initialization of `TASKCOMMON` variables is supported with SEGLDR 9.2 and higher.

Autotasking Directives [4]

If your system includes multiple central processing units (CPUs), your program may be able to make use of *multitasking*, or running simultaneously on more than one CPU. This technology speeds up program execution by decreasing elapsed time. You can determine the number of CPUs on your system by entering either the `target`(1) command (on Cray PVP systems) or the `mpstat`(1) command (on SPARC systems).

> **Note:** The features described in this section are not available on CRAY T3E systems.

The CF90 compiler automatically recognizes many parallel coding constructs, and it compiles them for multitasking without requiring additional user input; this capability is called *Autotasking*.

To benefit from the Autotasking software, your program must be suitable for dividing into separate tasks. If you are uncertain whether you can make use of this capability, you can compare execution times of different Autotasked versions of your program by using ATExpert to analyze performance gains. To use ATExpert, specify `-e X` on the `f90`(1) command line.

Autotasking directives let you specify the level of parallelism desired. You can start and end parallel processing at any number of suitable points within a subprogram. These directives are useful when the compiler fails to recognize parallelism that you know exists. This can occur, for example, when you have subroutine calls that can be executed in parallel.

This section provides an overview of the Autotasking directives recognized by the CF90 compiler. For more information on the tasking capabilities available through the command line, see subsection 2.2.20, page 23.

> **Note:** The ability to use Autotasking directives in a subprogram that host associates a variable is deferred. Attempts to do this will result in undefined behavior. This note applies only to Autotasking directives; it does not apply to parallelism detected by the compiler.
>
> For more information on host association, see the *CF90 Fortran Language Reference Manual, Volume 1*, publication SR–3902.

Overview

4.1

Autotasking directives control the way the CF90 compiler multitasks your program. You can insert tasking directive lines directly into your source code. The CF90 compiler supports the following Autotasking directives:

- CASE, END CASE
- CNCALL
- DO ALL
- DO PARALLEL, END DO
- GUARD, END GUARD
- MAXCPUS
- NUMCPUS
- PARALLEL, END PARALLEL
- PERMUTATION

How you specify Autotasking directives depends on the source form you are using.

- If you are using fixed source form, indicate an Autotasking directive line by placing the characters !MIC$ or CMIC$ in columns 1 through 5. If the compiler encounters a nonblank character in column 6, the line is assumed to be a continuation line. Columns 7 and beyond can contain one or more Autotasking directives. Autotasking directives entered in columns beyond the default line widths are ignored.
- If you are using free source form, indicate an Autotasking directive by the characters !MIC$, followed by a space, and then one or more Autotasking directives. If the position following the !MIC$ contains a character other than a blank, tab, or newline character, the line is assumed to be a continuation line. The !MIC$ need not start in column 1, but it must be the first text on the line.

In the following example, an asterisk (*) appears in column 6 to indicate that the second line is a continuation of the preceding line:

```
!MIC$ DO ALL
!MIC$*SHARED(lizzy)
```

If you want to specify more than one Autotasking directive on a line, separate each directive with a comma. Some Autotasking directives require that you specify one or more parameters; when specifying an Autotasking directive of this type, no other compiler directive can appear on the line.

Spaces can precede, follow, or be embedded within an Autotasking directive, regardless of source form.

> **Note:** The CF77 compiling system supports the FMP `CMIC$ SOFT EXIT` directive. The CF90 compiler does not support this directive, but similar functionality can be obtained by using the Fortran 90 `EXIT` statement. For more information on the `EXIT` statement, see the *CF90 Fortran Language Reference Manual, Volume 1*, publication SR–3902.

Autotasking directives

4.2

The following subsections describe the Autotasking directives.

Concurrent blocks: `CASE` and `END CASE`

4.2.1

The `!MIC$ CASE` directive serves as a separator between adjacent code blocks that can be executed concurrently. `!MIC$ END CASE` serves as the terminator for a group of one or more parallel `CASE` directives. These directives can appear only in a parallel region bounded by the `PARALLEL` and `END PARALLEL` Autotasking directives.

The formats for these directives are as follows:

```
!MIC$ CASE
!MIC$ END CASE
```

Section 4

In the following example, CASE directives have been added. Currently, the compiler does not perform interprocedural analysis and would not, therefore, add the CASE directives automatically for this example. Because CASE directives have been added in the following example, subroutines called within the CASE/END CASE directives are concurrently executable:

```
!MIC$ PARALLEL
!MIC$   CASE
        CALL ABC
!MIC$   CASE
        CALL DEF
!MIC$   CASE
        CALL GHI
!MIC$   END CASE
!MIC$ END PARALLEL
```

The work in the subroutine calls must complete before execution continues with the code below the END CASE.

A special form of the CASE/END CASE directive pair allows only one processor to execute a code block in a parallel region, as follows:

```
!MIC$ PARALLEL
!MIC$ CASE
      CALL XYZ
!MIC$ END CASE
       :
!MIC$ DO PARALLEL
      DO I = 1,IMAX
       :
      END DO
!MIC$ END PARALLEL
```

In the preceding example, only one processor calls XYZ, and then all available processors execute the code following the END CASE until the DO PARALLEL directive is reached.

Section 4

Declare lack of side effects: CNCALL

4.2.2

The !MIC$ CNCALL directive allows a loop to be autotasked by asserting that subroutines called from the loop have no loop-related side effects (that is, they do not modify data referenced in other iterations of the loop) and therefore can be called concurrently by separate iterations of the loop. CNCALL is inserted immediately preceding the loop.

The format for this directive is as follows:

```
!MIC$ CNCALL
```

Example:

```
!MIC$ CNCALL
      DO I = 1, N
        CALL CRUNCH ( A(I), B(I) )
      END DO
```

Mark parallel loop: DO ALL

4.2.3

The !MIC$ DO ALL directive indicates that the DO loop beginning on the next line may be executed in parallel by multiple processors. No directive is needed to end a DO ALL loop, (that is, the DO ALL initiates a parallel region that contains only a DO loop with independent iterations). The loop index variable for a DO ALL must be specified as a PRIVATE variable.

Every variable in a parallel region must be declared as PRIVATE or SHARED unless the AUTOSCOPE parameter appears on the directive.

The format of this directive is as follows:

```
!MIC$ DO ALL [parameter ] [, parameter ...]] [work_distribution ]
```

Section 4

parameters	Table 5 describes parameters for the `DO ALL` directive. More than one of these parameters can appear on the directive, and they must be separated by commas or blanks.
work_distribution	Parameters that specify the work distribution policy for iterations of the parallel `DO` loop. Only one can be used for a given `DO` loop. By default, iterations are distributed one at a time (`SINGLE`) for Cray PVP systems, and they are distributed statically (`NCPUS_CHUNKS`) for SPARC systems. Table 6, page 127, describes the work distribution parameters.

Note that the specifications for *parameters* do not have to precede the specifications for *work_distribution*. The *parameters* and the *work_distribution* can be in any order, and they may be intermixed.

Table 5. Autotasking directive *parameters*

Parameter	Description
`AUTOSCOPE`	Specifies that all unscoped variables that have not been explicitly scoped with a `PRIVATE` or `SHARED` declaration are scoped according to the default rules for scoping variables. For more information on the default rules for scoping variables, see subsection 4.3, page 133.
`IF` (*expr*)	Performs a run-time test to choose between uniprocessing and multiprocessing. When not specified, multiprocessing is chosen if the loop was not called from within a parallel region. The logical expression (*expr*) determines (at run time) whether multiprocessing will occur. When *expr* is true, multiprocessing is enabled.

Table 5. Autotasking directive *parameters* (continued)

Parameter	Description

`MAXCPUS (`*n*`)`	Specifies the maximum number of CPUs that the parallel region can use effectively but does not ensure that *n* processors will be assigned; this is the optimal maximum. The *n* argument must be of type integer . Argument *n* can be a constant, a variable, or an expression; both of the following are valid specifications:
	`MAXCPUS (2)`
	`MAXCPUS (NUM)`

`PRIVATE(`*var* `[,`*var* `]`...`)`	Specifies that the variables listed will have *private* scope; that is, each task (original or helper) will have its own private copy of these variables. The `PRIVATE` clause identifies those variables that are not shared between parallel processes. One variable cannot be declared both `PRIVATE` and `SHARED`. The loop control variable of the `DO ALL` loop cannot be specified as `SHARED`; it must be specified as `PRIVATE`. Variables cannot be subobjects (that is, array elements or components of derived types).

Table 5. Autotasking directive *parameters* (continued)

Parameter	Description
SAVELAST	Specifies that the values of private variables, from the final iteration of a DO ALL directive, will continue in the original task after execution of the iterations of the DO ALL. By default, private variables are not guaranteed to retain the last iteration values. SAVELAST can be used only with DO ALL, and if the full iteration set is not completed (for example, if the loop is exited early), the values of private variables are indeterminate.
SHARED(*var* [,*var*] ...)	Specifies that the variables listed will have *shared* scope; that is, they are accessible to both the original task and all helper tasks. The SHARED clause identifies those variables that are shared between parallel processes. One variable cannot be declared both PRIVATE and SHARED. The loop control variable of the DO ALL loop cannot be specified as SHARED; it must be specified as PRIVATE. Variables cannot be subobjects (that is, array elements or components of derived types).

Table 6. Autotasking directive *work_distribution*

Parameter	Description
`CHUNKSIZE(`*n*`)`	Specifies the number of iterations to distribute to an available processor . *n* is an integer expression. For best performance, *n* should be an integer constant. For example, given 100 iterations and `CHUNKSIZE(4)`, 4 iterations at a time are distributed to each available processor until the 100 iterations are complete.
`GUIDED [(`*vl*`)]`	Specifies the use of guided self-scheduling to distribute the iterations to available processors. This mechanism minimizes synchronization overhead while providing acceptable dynamic load balancing. The *vl* argument is the vector length. This option is for use on systems with a length greater than 64. *vl* must be of type integer and can be either a constant or a variable. On CRAY T90 systems and CRAY C90 systems, the default *vl* is 128. On all other Cray PVP systems and on SPARC systems, the default *vl* is 64.
`NCPUS_CHUNKS` (SPARC systems only)	Specifies that the iterations should be divided into the same number of chunks as the value of the `NCPUS` environment variable and distributed one each to the processors. That is, this parameter creates precisely as many chunks as there are processors available, and each processor works on one chunk. This distribution policy gives up load balancing, but has very low overhead and does not require any synchronization among the processors until the end of the parallel region. On architectures that use memory operations for synchronization, these advantages outweigh the lack of load balancing. This distribution policy is the default for SPARC systems.

Section 4

Table 6. Autotasking directive *work_distribution* (continued)

Parameter	Description
`NUMCHUNKS(`*m*`)`	Specifies that the iterations are divided into *m* chunks of equal size (with a possible smaller residual chunk) and distribute these chunks to available processors. The *m* argument must be an integer constant. For example, given 100 iterations and `NUMCHUNKS(4)`, 25 iterations at a time are distributed to each available processor until the 100 iterations are complete.
`SINGLE`	Specifies that iterations should be distributed one at a time to available processors. This is the default distribution policy for Cray PVP systems.
`VECTOR`	Distributes the maximum vector length (either 64 or 128 iterations at a time) to each processor. If this is used to stripmine an inner loop, unrolling is performed to automatically improve scheduling. On SPARC systems, 64 iterations is the maximum.

Section 4

4.2.4 Mark parallel loop: `DO PARALLEL` and `END DO`

The `!MIC$ DO PARALLEL` directive indicates that the `DO` loop beginning on the next line may be executed in parallel by multiple processors. No directive is needed to end a `DO PARALLEL` loop.

The `!MIC$ END DO` directive extends a control structure beyond the `DO` loop. Without a `!MIC$ END DO` directive, all of the CPUs will synchronize immediately after the loop, so that no processors can continue executing until all of the iterations are done. A `!MIC$ END DO` directive moves this point of synchronization from the end of the loop to the line of the `!MIC$ END DO` directive.

This lets the CF90 compiler use parallelism in loops containing some forms of reduction computations. These directives can be used only within a parallel region bounded by the `PARALLEL` and `END PARALLEL` directives.

Every variable in a parallel region must be declared as `PRIVATE` or `SHARED`.

The formats for these directives are as follows:

```
!MIC$ DO PARALLEL [work_distribution ]
!MIC$ END DO
```

The *work_distribution* parameters are described in Table 5, page 124. Only one *work_distribution* parameter can be used for a given `DO` loop.

In the following example, a parallel region is defined by `PARALLEL` and `END PARALLEL`. A reduction computation is implemented by a `DO PARALLEL/END DO` pair , which ensures that all contributions to `SUM` and `BIG` are included, and `GUARD/END GUARD`, which protects the updating of shared variables `SUM` and `BIG`.

```
      SUM = 0.0
      BIG = -1.0
!MIC$ PARALLEL PRIVATE(XSUM,XBIG,I)
!MIC$1         SHARED(SUM,BIG,AA,BB,CC)
      XSUM = 0.0
      XBIG = -1.0
!MIC$ DO PARALLEL
      DO I = 1, 2000
      :
      XSUM = XSUM + (AA(I)*(BB(I)-CC(AA(I))))
      XBIG = MAX(ABS(AA(I)*BB(I)), XBIG)
      :
      END DO
!MIC$ GUARD
      SUM = SUM + XSUM
      BIG = MAX(XBIG,BIG)
!MIC$ END GUARD
!MIC$ END DO
!MIC$ END PARALLEL
```

Critical region: `GUARD` and `END GUARD`
4.2.5

The `!MIC$ GUARD` and `END GUARD` directives delimit a critical region, providing the necessary synchronization to protect or guard the code inside the critical region. A *critical region* is a code block that is to be executed by only one processor at a time, although all processors that enter a parallel region will execute it.

The formats for these directives are as follows:

```
!MIC$ GUARD [n]
!MIC$ END GUARD [n]
```

n Mutual exclusion flag; two regions with the same flag cannot be active concurrently. *n* must be of type integer and can be a variable or an expression, from which the low-order 6 bits are used. For example, `GUARD 1` and `GUARD 2` can be active concurrently, but two `GUARD 7` directives cannot.

For optimal performance, no flag should be specified. Otherwise, *n* should be an integer constant; a general expression can be used for the unusual case that the critical region number must be passed to a lower-level routine. When *n* is not provided, the critical region blocks only other instances of itself, but no other critical regions. Critical regions may appear anywhere in a program; that is, they are not limited to parallel regions.

Section 4

Allocate CPUs: `MAXCPUS`
4.2.6

The `!MIC$ MAXCPUS` directive controls the number of CPUs allocated to a parallel region of code. After the region is finished, the number of CPUs returns to the previous value.

When the directive is used, the CF90 compiler adds the `MAXCPUS` parameter with the specified value to each `!MIC DO ALL` or `!MIC PARALLEL` directive that marks a parallel region. Without this directive, CPUs are allocated based on workload and the `NCPUS` environment variable.

The format for this directive is as follows:

```
!MIC$ MAXCPUS (ncpus )
```

ncpus Specifies the maximum number of CPUs to allocate. *ncpus* must be of type integer and can be a constant, variable, or expression.

4.2.7 Specify maximum number of CPUS for a parallel region: NUMCPUS

The !MIC$ NUMCPUS directive allows you to control the number of CPUs allocated to all subsequent parallel regions. The number of CPUs stays constant until another NUMCPUS directive is encountered.

The format for this directive is as follows:

```
!MIC$ NUMCPUS (ncpus )
```

ncpus Specifies the maximum number of CPUs to allocate. *ncpus* must be of type integer and can be a constant, variable, or expression.

Without the directive, the code will have CPUs allocated based on workload and the NCPUS environment variable. When the directive is used, the system limits the number of CPUs to the specified value for all parallel regions following the NUMCPUS directive in a file. The value is reset when a new NUMCPUS directive is encountered. If the value of *ncpus* is a variable, the system checks the value at run time to see if it has been set to a value greater than 64. If the value is greater than 64, the CF90 compiler sets the value to 64.

The NUMCPUS directive overrides the NCPUS environment variable unless you are requesting more CPUs than were originally obtained at the start of execution with NCPUS. For example, if NUMCPUS is set to 9 in a program and NCPUS is set to 2, only 2 CPUs will ever be used on the program.

4.2.8 Mark parallel region: **PARALLEL** and **END PARALLEL**

The `!MIC$ PARALLEL` and `!MIC$ END PARALLEL` directives mark, respectively, the beginning and end of a parallel region. Parallel regions are combinations of redundant code blocks and partitioned code blocks. The formats for these directives are as follows:

```
!MIC$ PARALLEL [parameters ]
!MIC$ END PARALLEL
```

The *parameters* are described in Table 5, page 124.

The `PARALLEL` directive indicates where multiple processors enter execution. The portion of code that all processors execute until reaching a `DO PARALLEL` directive is called a *redundant code block*. Because the iterations of the `DO` loop within a `DO PARALLEL` directive are distributed across available processors, this portion of code is called the *partitioned code block*. The scope of a variable in a parallel region is either shared or private. Shared variables are used by all processors; private variables are unique to a processor.

4.2.9 Declare an array with no repeated values: **PERMUTATION**

The `!MIC$ PERMUTATION` directive declares that an integer array has no repeated values. This is useful when the integer array is used as a subscript for another array (vector-valued subscript). The format for this directive is as follows:

```
!MIC$ PERMUTATION (ia [,ia ] ...)
```

ia Integer array that has no repeated values for the entire routine.

When an array with a vector-valued subscript appears on both sides of the equal sign in a loop, many-to-one assignment is possible even when the subscript is identical. Many-to-one assignment occurs if any repeated elements exist in the subscripting array. If it is known that the integer array is used merely to permute the elements of the subscripted array, it can often be determined that many-to-one assignment does not exist with that array reference.

Sometimes a vector-valued subscript is used as a means of indirect addressing because the elements of interest in an array are sparsely distributed; in this case, an integer array is used to select only the desired elements, and no repeated elements exist in the integer array, as in the following example:

```
!MIC$ PERMUTATION(IPNT)   ! IPNT has no repeated values
      ...
      DO I = 1, N
         A(IPNT(I)) = A(IPNT(I)) + B(I)
      END DO
```

Autoscoping rules

4.3

When the CF90 compiler generates code to handle a `!MIC$ DO ALL` or a `!MIC$ PARALLEL` directive, one of the following must be true:

- All the variables and arrays in the region must be defined in a `SHARED` or `PRIVATE` parameter.
- The `AUTOSCOPE` parameter must be specified.

A *shared variable* or *array* is one that all the processors use. A *private variable* or *array* is one for which each of the processors has it own copy. If the `AUTOSCOPE` parameter is specified, the compiler uses the following rules to determine shared or private status:

- The following entities are shared:
  - Variables or arrays in a `SHARED` parameter.
  - Variables or arrays that are read-only. For more information, see subsection 4.3.2.1, page 135.

  – Arrays indexed by the loop index. For more information, see subsection 4.3.2.2, page 135.

  – Variables or arrays that are read-then-write. For more information, see subsection 4.3.2.3, page 136.

- The following entities are private:

  – Variables or arrays in a `PRIVATE` parameter.

  – Variables or arrays that are write-then-read. For more information, see subsection 4.3.2.4, page 136.

In both cases, `SHARED` and `PRIVATE` parameters on a `!MIC$ DO ALL` or a `!MIC$ PARALLEL` statement override the default determination.

For more information on the `AUTOSCOPE`, `SHARED`, and `PRIVATE` parameters, see Table 5, page 124.

4.3.1 User-added scope required

If you want the scope of arguments to such subroutine calls to be private, you must explicitly declare them as private. To be certain of correct scope, all variables or arrays that occur in a function or subroutine call should be specified as shared or private.

The following example shows use of a subroutine call:

```
!MIC$ DO ALL AUTOSCOPE
      DO I = N1, N2
        CALL MMP(A(I), B, C)
      END DO
```

It is indeterminate whether `A(I)`, `B`, or `C` is read or written. The CF90 compiler assigns `A` as shared because it is indexed by the control variable. The compiler assumes that `B` and `C` are read; therefore, it designates them as shared variables. The compiler prints a message stating that such variables as `A`, `B`, and `C` require a `PRIVATE` or `SHARED` declaration, but the compiler treats them as shared.

Section 4

Examples
4.3.2

The following examples show shared and private variables and arrays.

Read-only variables
4.3.2.1

The following examples show read-only variables:

```
!MIC$ DO ALL PRIVATE(I) SHARED(N1,N2,A)
      DO I = N1, N2
      ...= A
      END DO
```

A is a shared variable because it is a read-only variable. All processors share the same location for A.

```
!MIC$ DO ALL SHARED(N1,N2,M1,M2,V) PRIVATE(I,J)
      DO 10 I = N1, N2
      DO 10 J = M1, M2
      ... = V(J)
   10 CONTINUE
```

V is shared because it is a read-only array. N1, N2, M1, and M2 are also shared because they are read-only variables. I and J are written and then read, so they are private variables.

Array indexed by loop index
4.3.2.2

The following example shows an array indexed by the loop index:

```
!MIC$ DO ALL SHARED(N1,N2,V,U,J) PRIVATE(I,T)
      DO I = N1, N2
        T = V(I)
        U(I,J) = T
      END DO
```

U and V are shared arrays because they are indexed by the loop index. All processors share the same location for V and U. T is written and then read, so it is a private variable. J is shared because it is a read-only variable.

Read-then-write variables
4.3.2.3

The following example shows read-then-write variables:

```
       SUM = 0.0
!MIC$ DO ALL SHARED(N1,N2,V,SUM) PRIVATE(I,T)
       DO I = N1, N2
       T = V(I)
!MIC$ GUARD
       SUM = SUM + T
!MIC$ END GUARD
       END DO
```

`SUM` is a shared variable because it is read before it is written. Special care is needed in writing into a shared variable.

Write-then-read variables and arrays
4.3.2.4

The following example shows write-then-read variables and arrays:

```
!MIC$ DO ALL SHARED(N1,N2,M1,M2) PRIVATE(I,J,V)
       DO 10 I = N1, N2
       DO 10 J = M1, M2
       V(J) = ...
       ... = V(J)
    10 CONTINUE
```

`V` is written to and then read. It must be a private array.

Restrictions
4.4

Subprograms that contain an assigned `GOTO` statement cannot contain Autotasking directives.

Branches out of a parallel region are not permitted and can produce incorrect results.

CF90 Programming Environment [5]

The CF90 programming environment includes many tools, utilities, and libraries. This section provides a software overview. It is written for users running the CF90 compiler on Cray PVP systems and CRAY T3E systems.

If you are running the CF90 compiler on a SPARC system, see *Introducing the CF90 Programming Environment*, publication IN–2155, for a software overview .

> **Note:** On Cray PVP systems, the products described as being part of the CF90 programming environment are not necessarily released with the CF90 compiler. The operating system, CrayTools, CrayLibs, and other releases contain software that can be used in conjunction with the CF90 compiler .
>
> Certain Cray Research software tools do not accept Fortran 90 source code. They are as follows: `atchop`(1), `fmgen`(1), and `fsplit`(1).

The CF90 compiler is one of many software products that form the CF90 programming environment. This environment allows you to develop, debug, and run Fortran codes on your computer system. It includes the following products:

- Loaders. `segldr`(1), `mppldr`(1), and `ld`(1) are available, depending on your platform.
- A lister, `cflist`(1).
- The `cflint`(1) utility, which checks CF90 programs for possible errors.
- A browser, `xbrowse`(1). For more information on the browser, see *Introducing the Program Browser*, publication IN–2140.
- The compiler information file (CIF) tools, which include the `cifconv`(1) command and the libraries. For more information on these see the *Compiler Information File (CIF) Reference Manual*, publication SR–2401.

- The libraries, which include functions optimized for use on Cray Research systems. Information on the individual library routines can be found in the online man pages for each routine. In addition to online man pages, the following manuals contain printed copies of the library routine man pages and other library information:

  – *Application Programmer's Library Reference Manual*, publication SR–2165

  – *Scientific Libraries Reference Manual*, publication SR–2081

  – *Math Library Reference Manual*, publication SR–2138

  The CF90 intrinsic procedures are implemented within the math library (`libm`), within `libfi`, and within the compiler itself. The *CF90 Fortran Reference Manual, Volume 2*, publication SR–3903, lists all the intrinsic procedures. In addition, the *CF90 Fortran Reference Manual, Volume 2*, publication SR–3903, contains copies of the online man pages for the procedures that are implemented within `libfi` and within the compiler. The intrinsic procedures that are implemented as `libm` library routines are available online and are published in the *Math Library Reference Manual* .

- The performance tools, which can help you analyze program performance. The performance tools include `apprentice`(1), `atexpert`(1), and `flowview`(1). Information on the individual tools can be found in the online man pages for each tool. The *Guide to Parallel Vector Applications*, publication SG–2182, describes several tools, and more information on the MPP Apprentice tool can be found in *Introducing the MPP Apprentice Tool*, publication IN–2511.

- Online documentation utilities. The `man`(1) command allows you to retrieve online man pages. The `cdoc`(1) command allows you to retrieve reference manuals in online form from your workstation.

- Debuggers. Both TotalView and CDBX are available. These debuggers include standard debugging capabilities, such as allowing you to step through code and set breakpoints. They can be used when you invoke the CF90 compiler and direct it to produce symbol tables. The `-g` and `-G` *debug* options to the `f90`(1) command line generate symbol tables.

The `-G 0` option generates symbol tables identical to those produced by specifying `-g`; no optimization, tasking, or vectorization is performed, and breakpoints may be set at each line. The `-G 1` option allows you to perform block-by-block debugging, except in the innermost loops. You can set breakpoints at referenced user labels and at the beginning and end of block constructs (such as `IF/THEN/ELSE` blocks, `DO/END DO` blocks, and `SELECT CASE/END SELECT` blocks).

For more information on TotalView, see *Introducing the Cray TotalView Debugger*, publication IN–2502. Detailed information on CDBX can be found in the *UNICOS CDBX Symbolic Debugger Reference Manual*, publication SR–2091, and in the *UNICOS CDBX Debugger User's Guide*, publication SG–2094.

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

M

N

O

P

R

S

T

U

V

W

X

Reader's Comment Form

CF90 Commands and Directives Reference Manual SR–3901 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 Commands and Directives Reference Manual SR–3901 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