3.3 FORTRAN 77 Release Notes

Document Version 1.0

Document Number 007-3359-010

Technical Publications:

Marcia Allen
Scott Fisher
Melissa Heinrich

Engineering:

Calvin Vu

3.3 FORTRAN 77 Release Notes
Document Version 1.0
Document Number 007-3359-010

Silicon Graphics, Inc.
Mountain View, California

Contents

1. Introduction

The FORTRAN compiler option provides an *f77* compiler, a tutorial for using the symbolic debugger *edge*, interfaces for the IRIS Graphics Library, and many UNIX™ system and library calls for IRIS-4D Series workstations. This release contains enhancements for VAX VMS™ compatibility and many bug fixes.

Note: FORTRAN 77 is referred to as FORTRAN throughout these release notes.

This document contains the following chapters:

1. Introduction
2. Configuration Information
3. Major Enhancements
4. Changes
5. Bug Fixes
6. Known Problems and Workarounds

Note: Packaged with these release notes is a separate sheet that contains the Software License Agreement. This software is provided to you solely under the terms and conditions of the Software License Agreement. Please take a few moments to review the Agreement. Note in particular paragraphs 1 and 2 under the heading "Terms and Conditions of Software License." This software may be used only by you on *one* workstation, computer, or server at one time, and may be copied only as necessary for backup or archival purposes.

1.1 Release Identification Information

Following is the release identification information for 3.3 FORTRAN 77 software:

Software Option Product	FORTRAN Compiler Version 2.0
Version	3.3
Product Code	S4-FTN-3.3
System Software Requirements	4D1-3.3
Dependencies	Development System S5-DV01-3.3 (Personal IRIS only)

1.2 FORTRAN Option Subsystems and Disk Usage

This section lists the subsystems (and their sizes) on your product option tape.

Those marked ''default'' are the default subsystems. If you are installing this option for the first time, the default subsystems are installed when you choose the ''default'' or ''automatic'' menu items during the installation procedure.

If you are updating from an older version of software, and you select ''default'' or ''automatic'', the system installs only the default subsystems that already are installed.

Subsystem	IRIS-4D Series, IRIS POWER Series (512 byte blocks)	Personal IRIS (512 byte blocks)
ftn.sw.ftn (default)	3406	3406
ftn.sw.G0libraries	2346	2346
ftn.sw.fedgetut	20	20
ftn.man.ftn (default)	1731	1731
ftn.man.relnotes (default)	46	46

1.3 Software Installation

As of system software release 4D1-3.2, software product option tapes no longer contain the software installation tools. Therefore, these release notes do not document the installation procedure. The installation software, as well as the installation instructions, can be found with the standard products you received from Silicon Graphics, Inc.

The following table shows you where to look for the installation software and documentation.

	Software	Document
Personal IRIS	Execution-only Environment (1)	*3.3 Standard System Release and Installation Notes*
Personal IRIS	Development Environment	*4D1-3.3 Development Release and Installation Notes* or *IRIS-4D System Administrator's Guide*
All other IRIS-4D Series workstations and servers	Execution-only Environment (1)	*4D1-3.3 Development Release and Installation Notes* or *IRIS-4D System Administrator's Guide*

1.4 On-Line Release Notes

When you install the on-line documentation for a product, you can view the release notes on your screen as you would an on-line manual page.

The command:

```
relnotes
```

displays a list of products that have on-line release notes. The *relnotes* command accepts the following arguments:

-h describes how to use *relnotes*

<product> displays the table of contents for a product's on-line release notes, in this case *sna3270*

<product> <chapter>
displays a specific chapter of a product's on-line release notes

-t *<product> <chapter>*
prints a specific chapter of a product's on-line release notes

To page through the chapter, press **<space>** and to quit, press **<del>** or **<ctrl-c>**. See *relnotes*(1) for more information.

1.5 Product Support

Silicon Graphics, Inc., provides a comprehensive product support maintenance program for IRIS-4D Series products. For further information, please contact your service organization.

2. Configuration Information

The FORTRAN option package takes about 3.8Mb of hard disk space and consists of the binary FORTRAN compiler front end and libraries and the on-line man pages. The binary package contains the following files:

- */usr/lib/fcom* FORTRAN compiler front end
- */usr/lib/crt1.o* Runtime startup
- */usr/lib/crtn.o* Runtime startup
- */usr/lib/libF77.a* FORTRAN intrinsic function library
- */usr/lib/libI77.a* FORTRAN I/O library
- */usr/lib/libU77.a* FORTRAN UNIX interface library
- */usr/lib/libisam.a* Indexed sequential access method library
- */usr/bin/ratfor* Rational FORTRAN dialect preprocessor
- */usr/bin/asa* FORTRAN carriage control interpreter
- */usr/bin/extcentry* Extract FORTRAN-callable entries from C sources
- */usr/bin/f77* FORTRAN driver link
- */usr/bin/fsplit* Split FORTRAN or RATFOR files
- */usr/bin/mkf2c* Generate FORTRAN-C interface routines
- */usr/bin/uconv* Convert FORTRAN unformatted files from IRIS 3000 series
- */usr/lib/align/libI77.a* FORTRAN I/O library with no alignment restrictions

- */usr/lib/fgldat.o* Graphic library support routine
- */usr/lib/fixade.o* Allow misaligned data
- */usr/lib/getargSGI.o* Provide IRIS 3000-compatible getarg() and iargc()
- */usr/lib/libfgl.a* FORTRAN graphic library
- */usr/tutor/edge/fortran/** *edge* tutorial files

3. Enhancements

This chapter covers all the enhancements that have been made to the compiler since the 4D1-3.2 release. Most of these enhancements were implemented to improve the compatibility between the IRIS-4D Series workstations and other major computer systems.

- **Floating-point exception trapping.**
  In this release, a comprehensive floating-point exception handling mechanism has been implemented. The user can set up his own exception handler or use the standard handler to print out the stack strace, count the exception trap, and/or coredump. The FORTRAN floating point exception is documented in the man page for *fsigfpe(2F)*.

- **Endfile records (SCR 6524)**
  When compiled with the **-vms_endfile** option, the run-time I/O library writes an endfile record instead of a physical end-of-file to the file every time an ENDFILE statement is executed. This enables you to create files that contain several endfile records as on the Cray™ and VMS™ systems.

- **Multiple EOF from stdin (SCR 6150)**
  When compiled with the new option **-vms_stdin**, the run-time I/O library allows rereading from *stdin* after entering an EOF character (`<ctrl-d>`). This provides user interface compatibility with VMS™ and IRIS Series 3000 workstations.

- **Binary files (SCR 4998)**
  Besides formatted and unformatted files, two extensions have been added to allow two more types of files:

  The first type allows you to use formatted read/write to access a file containing binary data. This happens when you use A-format descriptor to read/write numeric data. This type of file is defined by opening the file with FORM='BINARY'.

  For example:

```
      open(unit=2,form='binary',status='unknown')
      work1 = 1.0
      write(2,1000) work1
      rewind(2)
      read(2,1000) work2
1000  format(A4)
      end
```

  The second type allows the use of an ordinary system files i.e. a binary files without any extraneous bytes added to mark the record boundaries. In this type of file each byte is individually addressable. A READ or WRITE request on these files consumes bytes until the I/O list is satisfied, rather than restricting itself to a single record. They are equivalent to files opened with FORM='BINARY' on the IRIS 3000 series.

  These files are opened as direct unformatted files with a record length of one byte. Note that in order to do this the program has to be compiled with the -old_rl option, since without that option the specified record length will be interpreted as the number of words and not the number of bytes.

- **POINTER type.**
  Variable type POINTER is now supported. Memory allocation and deallocation are executed via calls to MALLOC and FREE functions. Currently, POINTER is not supported by the *dbx* debugger.

- **SYS$INPUT, SYS$OUTPUT, and SYS$ERROR.**
  VMS™ predefined system logical names SYS$INPUT, SYS$OUTPUT, and SYS$ERROR are supported. This allows an OPEN statement to associate a unit number to *stdin*, *stdout*, and *stderr* respectively. A I/O unit opened in this manner can be redirected with the standard UNIX I/O redirection on the command line without having to change every READ and WRITE statement to use the predefined unit numbers 5, 6, and 0. This will also help people porting code from the IRIS 3000 series where unit 0 is connected to *stdin* and *stdout* and unit 1 is connected to *stderr*.

- **Omitted arguments.**
  Arguments in a function or subroutine call can now be omitted by specifying nothing between the argument delimiters, e.g. CALL SUB(1,,3). The omitted argument will be translated into a %VAL(0) and passed to the function or subroutine.

- **Maximum number of continuation lines.**
  The default number of continuation lines has been increased to 99. The user can increase this number by using the -NC option in the *f77* command line.

- **Variable number of arguments.**
  Variable number of arguments is allowed in the VMS™ system calls that are prefixed with the letters LIB$, OTS$, or SMG$. This is done by prepending the number of actual arguments to the argument list when calling those functions. To remain backward compatible, this extra argument is added only when the program is compiled with the *-vms_library* option. This compilation option is required when compiling programs that use the Accelr8™ runtime library package.

- **Using REAL*8 as default type (SCR 7316)**
  When compiled with the *-r8* option all REAL variables and constants will have the default type of REAL*8. However, if the type declaration statement explicitly states REAL*4 then the variable will remain as 32-bit floating point.

- **Assigning integer values to LOGICAL*1 (SCR 4995)**
  In many programs written for older generation computers, LOGICAL*1 was used in store 8-bit integer values since variable types BYTE and INTEGER*1 did not exist. To facilitate porting those programs to Silicon Graphics 4D series LOGICAL*1 type is now treated in the same way as INTEGER*1 and assigning an integer value to a LOGICAL*1 variable does not result in the conversion of the integer value into a logical TRUE/FALSE value.

- **Increased I/O performance (SCR 8434)**
  The runtime I/O library has been revised to make it execute faster. In some I/O extensive benchmarks, the speed has been increased as much as four times.

- **Changing constants passed as subroutine arguments (SCR 4775)**
  The old compiler allows a subroutine to change the value of its argument even when a constant is used as the actual argument. This causes the constant to change value and is very difficult to debug. In the new compiler this user error will result in a coredump at runtime so the user can figure out where his mistake is.

- **Variable declaration and initialization.**
  Variables can be declared in a type declaration statement and be assigned an initial value on the same line.

- **Using functions as subroutines (SCR 6590)**
  A function returning numeric values can now be used as a subroutine in the same program unit where it is also used as a function.

- **Multiprocessing Enhancements**

- **Support for REDUCTION**
  Min, max, sum and product reductions can now be handled by the compiler. The compiler does some simple tests to check the validity of the reduction. It is the responsibility of the user to assure that the reduction is legal. Note that reductions can cause roundoff errors to accumulate in a different order, causing the final results to be slightly different.

An example of reductions follows:

```
      x(j) = 0.0
      y = 1.0
      z = c(1)
C$docacross local(i), share(a,b,c), reduction (x(j),y,z)
      do i = 1, n
      x(j) = x(j) + a(i)!sum reduction, array reference
      y = y * b(i)!product reduction
      z = max (z, c(i))!max reduction
      enddo
```

- **New Scheduling Options**

  The previous release of Multiprocessed Fortran provided a single method of dividing loop iterations and assigning them to processes. That method is currently the default, and it is called **SIMPLE** scheduling. The current release provides 5 scheduling methods:

  SIMPLE: The iterations of the loop are divided into contiguous blocks, and each process is given one block. This method has the lowest overhead. It can, however, cause load balancing problems.

  INTERLEAVE: The interations are broken up into pieces of size **CHUNK**. These pieces are then assigned to the processed in an interleaved way.

  DYNAMIC: The iterations are broken up into pieces of size **CHUNK**. When a process finishes its current piece, it enters a "critical section" to get the next piece.

  GSS: "Guided Self Scheduling" is like **DYNAMIC**, but instead of using fixed sized pieces, the piece size is varied. **GSS** provides big pieces to start with, and small pieces towards the end.

  RUNTIME: With **RUNTIME** scheduling, the user provides the scheduling type at runtime through the environment variables "MP_SCHEDTYPE" and "CHUNK". For example "setenv MP_SCHEDTYPE GSS" directs any loop with the **RUNTIME** schedule type to use **GSS** scheduling.

Scheduling options can be designated in a variety of ways, with the most local designation taking precedence. These scheduling designations are listed in order of precedence:

(1) The clauses **MP_SCHEDTYPE=name** and/or **CHUNK=num** can be added to the **C$DOACROSS** directive. This defines the schedule type and/or chunk size for a single loop.

(2) The source directives **C$MP_SCHEDTYPE=name** and/or **C$CHUNK=num** can be added to the source code. This defines the default schedule type and/or chunk size for all multiprocessed loops ocurring afterwards.

(3) The compiler driver switches "-mp_schedtype=name" and/or "-chunk=integer" can be added to the compile line. This defines the default schedule type and/or chunk size for all multiprocessed loops that have not otherwise been designated.

(4) The environment variables "MP_SCHEDTYPE" and "CHUNK" define the schedule type for all multiprocessed loops with schedule types of **RUNTIME**.

Defaults:
if **MP_SCHED_TYPE** is not set, and **CHUNK** is not set, the default is **SIMPLE**.
if **MP_SCHED_TYPE** is not set, and **CHUNK** is set, the default is **DYNAMIC**.
if **MP_SCHED_TYPE** is set, and **CHUNK** is not set, the default is **CHUNK=1**.

- **Conditional Parallelism**

The **C$DOACROSS** statement may contain an **IF** clause that is evaluated at runtime, just before the loop is executed. If the clause is **TRUE**, then the loop executes in parallel. If the clause is **FALSE**, then the loop executes serially. This allows the user to check if there is enough work in the loop to offset the parallel loop overhead. To break even you currently need about 400 clocks of work, which normally works out to about 100 floating point operations. These IF clauses are automatically produced by PFA. For example:

```
C$DOACROSS IF(n-k .gt. 50), share(a,b,c), local(i)
           do i = n,k
              a(i) = a(i) + b(i)*c(i)
           enddo
```

- **New Locking and Barrier Functions**
  The zero argument functions mp_setlock(), mp_unsetlock() and mp_barrier() provide convenient access to the locking and barrier functions. These barrier functions automatically initialize a single lock and barrier through usconfig, usinit and usnewlock. For a great many programs a single lock and barrier is enough. If you need more, you should use the ussetlock family of routines directly.

- **Arrays may now be LASTLOCAL**

- **New Environment Variables**
  The following new environment variables have been provide the user with runtime control of the program.

  MP_SET_NUMTHREADS specifies the number of threads to use in the job, regardless of the number of cpus on the machine. For compatibility with the previous release, NUM_THREADS is supported as a synonym for MP_SET_NUMTHREADS.

  MP_BLOCKTIME is set to an integer. It is equivalent to calling mp_blocktime(3F) during program startup.

  MP_SCHEDTYPE and CHUNK are examined to determine the scheduling type for the **C$DOACROSS** loop if the scheduling type is set to **RUNTIME**. MP_SCHEDTYPE may be set to SIMPLE, INTERLEAVE, DYNAMIC, or GSS, and CHUNK may be set to a positive integer.

  MP_PROFILE may be set to get better profiling information on the synchronization routines. Normally, the synchronization routines do their job as quickly as possible, without regard for profiling information. Setting this variable will cause the synchronization routines to use a slightly slower synchronization method where each step is done by a separate routine with a long descriptive name. This can occasionally be useful if you suspect that the syncronization routines are the bottleneck. Note that MP_PROFILE does not have to be set in order to get profiling information. MP_PROFILE simply provides a more detailed profiling of the syncronization routines at the cost of slower execution.

- **Local COMMON Blocks**
  Named **COMMON** blocks may now be declared local to each task. Local **COMMON** blocks provide each process in the parallel job with its own private copy of the **COMMON** block.
  To define a **COMMON** block as local, use the link time compiler switch **-Xlocaldata <list>**, where **<list>** is a list of the external name of the **COMMON** block(s). The external name known to the loader is in lower case, with a trailing underscore, and no surrounding slashes. Thus:

```
$ROOT/usr/bin/f77 -mp a.o -L -L$ROOT/usr/lib -Xlocaldata foo_,bar_
```

  will make the common blocks **/FOO/** and **/BAR/** local.

  The local **COMMON** blocks may not be initialized with **DATA** statements. Blank **COMMON** may not be made local.

  Data is not automatically transferred between the master thread's version of the **COMMON** block, and the slave thread's version. However, it is possible to copy values from the master's version into the slave's version via the special source directive **C$COPYIN <list>**. Here, the **<list>**, must be items from local **COMMON** blocks. The item(s) may be scalars, arrays, individual array elements, or entire **COMMON** block names, including the slashes. For example:

```
                real x, y(100)
C$COPYIN x, y, /foo/, a(i)
```

  will propagate the values of **x**, the array **y**, the entire contents of **COMMON** block **/foo/**, and the "ith" element of array **a**. All of these items must be members of local **COMMON** blocks or whole **COMMON** blocks.

  Note that this directive gets translated into executable code; all subscripts appearing on array elements will be evaluated at runtime.

- **Gang Scheduling**
  In prior releases, the processes in the parallel job were scheduled as ordinary processes. This could cause extremely poor runtime performance when other jobs were running at the same time. In this release, the processes are scheduled as a "gang" as default; either all of the processes run, or none of them do. Gang scheduling allows a parallel job to run in a multi-user environment in a timely fashion. For example: two parallel jobs can run simultaneously, and it will only take twice as long as one job, rather than ten or twenty times as long.

  Gang scheduling has the disadvantage that if a job wants all the cpus, it will get kicked out whenever someone else wants to do anything, even though several of the cpus may be idle. To run a parallel job an a shared machine in a timely fashion, use one less process than the available cpus. This allows the job to run most of the time, leaving one cpu free for other users. The scheduler will still kick out the job when several other jobs run at the same time.

  To turn off the default gang scheduling, see the man page schedctl(2).

- **Unrolling in the optimizer, and PFA**
  The standard optimizer (uopt) is now capable of doing a limited form of unrolling; PFA is a little bit smarter about unrolling than uopt. As a result, most programs will see no difference whether they use PFA's unrolling or not, since the optimizer will wind up doing it if PFA didn't.

4. Changes

This chapter describes the differences between this release and the previous releases that may result in improvements, incompatibilities, or simply a change in the compiling/linking/debugging process.

- **Trigonometric functions.**
  The single-precision trigonometric functions have been modified to give faster execution and generally better results. Most of the time, the result would be identical to the old version but there may be some very small differences due to rounding in a few cases (about 20% of the time). When there is a difference, the new result is generally more accurate then the old one.

- **LOGICAL*1 variables.**
  To facilitate the porting of programs written for older generation computers where there was no INTEGER*1 type, LOGICAL*1 is now treated in the same way as INTEGER*1 and assigning an integer value to a LOGICAL*1 variable does not result in the conversion of the integer value into a logical TRUE/FALSE value. This change will not effect the execution of existing programs unless the program depends on specific bit patterns of the TRUE/FALSE values to work correctly.

- **Changing constants passed as subroutine arguments.**
  The old compiler allows a subroutine to change the value of its argument even when a constant is used as the actual argument. This causes the constant to have a wrong value and the problem is very difficult to debug. In the new compiler, this user error will result in a coredump at runtime. If you experience a coredump after compiling with the new compiler, it is a good idea to check for this user error first before calling the Hot Line.

- **Intrinsic subroutine time() vs. library function time()**
  Since the 1.31 version of the compiler, a VMS-compatible intrinsic subroutine time() has been added. This causes the calls to the old time() library function to result in an error message regarding bad number of arguments. Any programs that want to use the time() library function must now add:

  ```
  EXTERNAL TIME
  ```

  in the program units where time() is used.

- **Gang Scheduling**

  In prior releases, the processes in the parallel job were scheduled as ordinary processes. This could cause extremely poor runtime performance when other jobs were running at the same time. In this release, the processes are scheduled as a "gang" by default. Either all of the processes run or none of them do. Gang scheduling allows a parallel job to run in a multi-user environment in a timely fashion. For example, two parallel jobs can run simultaneously, and it will take only twice as long as one job, rather than ten or twenty times as long.

  The disadvantage of gang scheduling is that if a job wants all the cpus, it will get kicked out whenever someone else wants to do anything, even though several of the cpus may be idle. To run a parallel job on a shared machine in a timely fashion, use one less process than the number of available cpus. This allows the job to run most of the time, leaving one cpu free for other users. The scheduler will still kick out the job when several other jobs run at the same time.

  To run off the default Gang Scheduling, see the schedctl(2) man page.

5. Bug Fixes

This chapter briefly describes the bugs that have been fixed in the 4D1-3.3 *f77* version. Some of the descriptions are followed by a Silicon Graphics, Inc., bug report number in the form (SCR xxxx).

- The compiler will now give a warning for multiple unnamed blockdata since it is against ANSI usage (SCR 3156).
- Some instances where the compiler dumps core after giving an error message have been fixed to provide better error recovery (SCR 4082, 4893, 5590).
- Initiliazing character variables with very long character strings can cause an error in *ugen*. This bug has been fixed (SCR 4539).
- Fixed a compiler bug to allow typeless constants to be compared with numeric variables in a conditional expression.
- Several minor bugs regarding PARAMETER statements have been fixed (SCR 4639, 5124, 6259).
- Fortran I/O statements which return the status of the file (e.g. INQUIRE) have been enhanced to take variables of size other than 4 bytes long (SCR 5212).
- All outstanding optimizer bugs have been fixed (SCR 6245, 6433, 7287, A16391).
- A bug in ugen which causes incorrect runtime result when a function entry is used inside a complex floating-point expression has been fixed (SCR 6590, 7303).
- Structure element can now be used as argument to call an intrinsic routine (SCR 6806).
- An array element of a structure can now be used as an argument in a function call without having to specify an array index (SCR 6809).

- Initializing a character variable with a character constant of length 0 no longer results in a *ugen* error (SCR 6906).
- Fixed a bug to allow inline code generation for the len() function when it is used in an I/O list (SCR 7078).
- Fixed a bug which caused the decimal digit not to be printed when the output format for the floating-point number specifies only one decimal digit (SCR 7149).
- A namelist record following a slash-terminated record is not recognized and result in a runtime I/O error. This has been fixed (SCR 7169).
- A bug in ISAM file operation where using a character variable as the record key may result in error 134, 'bad key description'. This bug has been fixed (SCR 7185).
- The compiler now gives a compile-time error message when the user tries to use an integer key of sizes other than 4 bytes instead of allowing the program to compile and produce a mysterious runtime error later (SCR 7211).
- The ibits() function has been fixed so it will give the correct value without sign-extending it (SCR 8561).
- Opening an ISAM file after closing another file can cause an I/O error. This has been fixed (SCR 7277).
- Using END='label' while reading a namelist record does not work. This bug has been fixed (SCR 8463).
- Several bugs where using untyped constants (either hollerith constants or character constants used in a numeric context) results in a compiler error have been fixed (SCR 8603, 8604).
- If a program unit contains DATA statements and the first executable statement in it has certain character sequences, e.g. two consecutive backslashes or aspostrophes, the compiler may give an invalid error. This has been fixed (SCR 8610).
- If the first executable statement(s) in a program unit starts with the word DATA, e.g. DATAM = 0.0, no debugging line number information will be generated for it and you cannot use the debugger to stop at that line. This error has been fixed (SCR 8741).

- Reading a namelist which contains an array of character data results in inrecognizable input. This has been fixed (SCR 9066).
- The FORTRAN frontend allows arbitrary functions and variables to be used to define a constant in a PARAMETER statement. This has been fixed.
- Clash between common block name and routine name no longer causes *fcom* to coredump.
- Hexadecimal constants in the form 'nnn'X can now have blanks.
- FORTRAN now allows concatenated expression involving the intrinsic function char().
- The compiler now generates an error message when an IMPLICIT character is redefined to have a different type.
- The compiler will no longer coredump when an ENDIF is missing inside a DO loop.
- A bug regarding formatted direct-access I/O has been fixed.
- FIxed a bug which gave wrong result when raising a complex variable to an integer variable power.
- Namelist read fails to read partial logical arrays. This has been fixed.
- Fixed a bug in namelist read of real constants in the form .nnnn.
- A function returning result of type complex must be asigned the result value just before the return statement otherwise it may give the wrong answer. This bug has been fixed.

The following bug fixes apply to Multiprocessed Fortran:

- **Improved Process Termination**
  Multiprocessed Fortran creates many processes to do the work contained in a single Fortran program. In the prior release, if one of these processes died, the rest of the processes in the group would often wait forever for the dead process to complete. Now when a process dies, all of the members of its process group are sent a signal, allowing the members in the process group to exit.

- **Delayed Creation of Processes (SCR 6970)**
  Programs that call 'fork', in particular, background graphics programs, now work with Multiprocessed Fortran. To fix this, the creation of the new processes was taken out of Fortran initialization. The runtime routines create the processes at the first **C$DOACROSS** loop.

  If a benchmark times only a portion of a run, the user may not want to include the thread creation as part of the benchmark time. Thread creation can be forced via mp_create(num), or mp_setup().

- **Support for up to 16 processes**
  The multiprocessed runtime libraries now support up to 16 processes.

6. Known Problems and Workarounds

This chapter describes the known problems with the current release of the FORTRAN product. It also lists workarounds.

- **Compiling old FORTRAN code (SCR 3526)**
  FORTRAN, by default, allocates local variables on the stack for faster execution speed. These local automatic variables are uninitialized, as opposed to being initialized to zero, which occurs using the static allocation of older FORTRAN systems. Also, the value of an automatic variable is not retained between successive calls to a subroutine as for a static variable. Therefore, if you have an old program, especially one ported from the VAX, that behaves strangely, recompile it with **-static** and check the results. If using **-static** works correctly, the problem is due to automatic allocation of undeclared static variables. If execution speed is not an issue, the program can be compiled with **-static** without having to be modified. Otherwise, you need to track down all variables that expect an initial value (either a zero value when starting a program or the value from the previous subroutine invocation in a subroutine call) and declare those variables as static using the STATIC statement.

- **Conditional GO TO offset limit (SCR 3917)**
  There is a restriction on the length of a conditional GO TO branch. The branch target must be less than ±32K bytes of machine code from the conditional GO TO statement. In FORTRAN, a conditional GO TO statement has the form:

```
IF (condition) GO TO label
IF (condition) THEN
```

  or a DO loop construction.

  However, an unconditional GO TO statement uses 3 bytes for its offset and allows jumps to addresses as far as 8 megabytes away. Therefore, the restriction on a conditional GO TO can be circumvented by using an unconditional GO TO. For example, instead of using:

```
IF (condition) GO TO label1
```

  where *label1* is more than 32K from the IF statement, change it to:

```
      IF (.NOT. condition) GO TO label2
      IDUMMY = 0   ! dummy statement to trick the optimizer
      GO TO label1
label2 CONTINUE
```

- **Intrinsic subroutine time() vs. library function time()**
  In the 1.31 version of the compiler, a VMS-compatible intrinsic subroutine time() has been added. This causes the old time() library function to result in an error message regarding bad number of arguments. Any programs that used time() in previous releases of the compiler must now add:

```
EXTERNAL TIME
```

  in the program units where the library function time() is used.

- **Nested DO loops sharing a termination label statement (SCR 4972)**
  When there are two or more DO loops sharing the same termination statement, a GO TO statement in an outside LOOP that jumps to the terminal label does not generate an error as it should. Instead, it may silently result in the execution of the inside loops and give the unpredictable result. This problem can be avoided by using separate loop control labels for nested DO loops that contain a GO TO statement that jumps to the end of an inner loop.

- **DATA statements at end of program units (SCR 5245)**
  If you have DATA statements at the end of the program unit, they may cause the compiler to loop forever. The solution is to move those DATA statements before the first executable statement.

- **NAMELIST group names are not automatically recognized in I/O statement (SCR 4214).**
  When using NAMELIST in an I/O statement, the namelist group name must be preceded by the keyword 'NML='.

- **Using the FLOAT instrinsic in a PARAMETER statement produces the wrong result (SCR 6763)**
  Defining a parameter by using an expression involving the intrinsic function FLOAT might give the wrong result. Avoid using the FLOAT instrinsic function to define parameters.

- **Using include files may confuse *dbx* (SCR 6796)**
  There are three ways to include a header file into a Fortran program: by using #include, by using $include, and by using the INCLUDE statement. The files included by using #include are preprocessed by *cpp* and can have C-style comments in them, the others are not. Sometimes the wrong line and file numbers are generated when include files are preceded by comments or blank lines. This makes *dbx* display the wrong line/file. Many line number bugs associated with $include have been fixed and right now there are no outstanding bugs for it. However, the most reliable way to include header files seems to be #include.

- **Header file contents**
  There is a limitation that header file can only contain declaration statemetns. When executable statements of the same program units, especially the first ones and the last ones, are split across several files the compiler may issue some internal error.

- **LOGICAL*1 output. (6822)**
  Printing out a LOGICAL*1 variable results in 0/1 being printed out instead of the logical .TRUE./.FALSE. values

- **READ statement requires an input list (SCR 7121)**
  If a READ statement does not have anything in the input list nothing will be read and the current record is not skipped.

- **-O3 and multiprocessed Fortran**
  Multiprocessed Fortran code optimized -O3 may run slower than multiprocessed Fortran code that has been optimized at level -O2. One important optimization, "no parameter aliasing" has been turned off for -O3 optimization of Multiprocessed Fortran code. "No parameter aliasing" combined with -O3 optimization has the potential to generate bad code for programs containing source from different languages. If a program contains only Fortran code, or if the non-Fortran code does not do any parameter aliasing, it is beneficial and legal to use both -O3 and "no parameter aliasing". Safe non-Fortran code also includes routines with less than 2 arguments, and interfaces to system calls. To turn on "no parameter Aliasing" during -O3 optimization, use the compiler switch **`-Wo,-noPalias`**. "No parameter Aliasing" is safe with all lower levels of optimization, and is turned on automatically in those cases.

- **Multi-processed I/O restrictions (SCR 8218)**
  The Fortran I/O library is not re-entrant. As a result, multiprocessed Fortran I/O is not supported. For example, Fortran does not allow two different processes to write to seperate files.

  The user can, however, write Fortran callable 'C' routines that use safe routines to do multiprocessed I/O. Safe routines include the system calls **read, write, open, close**, and the routines listed in the usconfig(3P) man page under the CONF_STHREADIOOFF command. For all routines listed in the man page, the user must first call usconfig(CONF_STHREADIOON), and must call them from 'C'. Some of the routines have Fortran callable twins that use the Fortran I/O library: these will fail when multiprocessed.