SGP/16.10

Starlink Project
Starlink General Paper 16.10

P. T. Wallace

23rd March 1992

Starlink Application Programming Standard

Contents

1 PREFACE

This Programming Standard should be read by all those who implement programs on Starlink computers. The techniques described are requirements for Starlink-supported applications software but are equally appropriate for private programming. The standard applies mainly to the implementation of freestanding VAX/VMS and Unix applications, but also includes some material specific to developing programs for the Starlink ADAM Software Environment.

Most of the recommendations made here have been available since the appearance of the first edition in June 1981. Apart from a few additions in August 1984, Starlink's advice on general FORTRAN coding standards has not changed. Recent versions of this document have contained additional material on use of graphics and on writing ADAM applications.

2 INTRODUCTION

This note offers advice on how to write applications programs for Starlink, so as to:

- maximize readability and comprehensibility
- encourage uniformity
- simplify maintenance
- eliminate ego
- assist portability
- promote reliability and fault tolerance

Programming standards like this one are apt to be treated with contempt by battle-hardened user-programmers, at least those who have only ever used one sort of computer and have never been faced with having to look after someone else's code. Even experienced professional programmers are sometimes reluctant to accept advice, in the confident belief that their own style constitutes the ideal balance between discipline and pragmatism. However, the lamentable standards of programming generally and Starlink's objective of software sharing make some restrictions necessary and desirable. The requirement that a program should work is merely the beginning; the effort expended in distributing and supporting even the best software is well known to be much greater than that required to write it in the first place.

Several existing standards were consulted before the Starlink standard was drawn up. Compared with most standards, the Starlink one is rather liberal, omitting many of the rules in these others while adding comparatively few of its own.

The 'rules' in this standard are of variable importance, and range from mere stylistic suggestions to firm requirements. The relative importance of each rule is indicated by the following coding:

.	Suggestion
!	Strong recommendation
!!	Rule which can sometimes be waived
!!!	Firm rule

Any program submitted to Starlink (whether written by a Starlink programmer or a generous user) may be vetted for conformity with this standard, and any violations taken into account when deciding whether Starlink is to undertake distribution and support. Major violations will only be acceptable if there are good reasons. For example (a) the program may be of an interim nature and not require long-term support, (b) it may be so urgently required that this has to override all other factors, (c) while departing from the standard in detail, it is so disciplined and consistent that it can be accepted for support on its own terms.

Of course, this document does not pretend to be an exhaustive specification for writing Starlink application programs. More information on, respectively, the Fortran 77 language and general program design and coding, can be found in the following two books:

- PROGRAMMING IN STANDARD FORTRAN 77 by A.Balfour and D.H.Marwick, published by Heinemann. This is a good reference for the language, and also contains some sound programming advice. Note that certain extensions to the published US standard are permitted in the Starlink programming standard; these are not included in Balfour and Marwick.
- THE ELEMENTS OF PROGRAMMING STYLE by B.W.Kernighan and P.J.Plauger, published by McGraw-Hill. Every Starlink programmer should read this classic work and follow its recommendations except in those rare instances where the Starlink standard differs.

3 GENERAL CODING STANDARD

The rules and guidelines in this section apply to all applications programming irrespective of what 'software environment' they run under. They fall into four groups – language, design, quality and presentation.

3.1 Language

Fortran only !!! **1**

For the time being, writers of Starlink application programs are urged to use Fortran exclusively. Increasing amounts of Starlink systems code are being written in ANSI C (SGP/4 is the present document's counterpart for the C language), but use of C is discouraged for applications code. A few pieces of Starlink system software use various dialects of assembler and Pascal, but these are being eliminated. There may conceivably be future interest in ADA.

The only permitted dialect of Fortran is ANSI X3.9-1978 (known as Fortran 77) plus certain US Department of Defense extensions and – in a few cases only – some extensions provided in DEC and several other proprietary Fortrans.

Discriminating Fortran users were disappointed with the 1978 standard, which failed to contain many expected improvements. Experience with preprocessors (RATFOR for example) had shown the feasibility of turning Fortran into a much better language with a minimum of disruption; unfortunately, many features which could easily have been added to the language weren't. DEC Fortran, however, does have a few of these features, and it was decided that certain of these should be permitted within the Starlink standard. The rationale was (i) it would not be hard to eliminate these extensions manually if necessary, (ii) they have become almost universal, and (iii) similar facilities will be part of later Fortran standards. Note that anyone wishing to adhere more rigidly to the Fortran 77 standard is permitted to do so. The latest version of Fortran, called Fortran 90, contains many improvements over Fortran 77 and almost eliminates the need to use **any** platform-specific features. Extra facilities not available in existing Fortran 77 implementations (for example the processing of whole arrays as primitive data items) are also available. The Fortran 90 standard identifies certain features as 'deprecated' or 'obsolescent', candidates for deletion in future standards; almost all of these are already prohibited by the Starlink standard. Fortran 90 compilers are not presently provided on Starlink but will be in due course.

The following extensions to the Fortran 77 standard are **permitted** within Starlink:

IMPLICIT NONE	encouraged
END DO	permitted
DO WHILE	permitted but worth avoiding
INCLUDE	encouraged

IMPLICIT NONE, END DO, DO WHILE and INCLUDE are all in Fortran 90. See Section 5, *Writing Portable Programs*, for detailed advice on INCLUDE statements.

Here are some examples of features which are part of neither the Fortran 77 standard nor the Starlink standard, and are to be **avoided**:

INTEGER*4, REAL*4, REAL*8, LOGICAL*1 etc. (see note 1, below)
Names $>$ 6 characters (note 2)
More than 19 continuation lines
Data initialization in a type declaration
Departures from Fortran 77 permitted statement order
Overlapping character substrings on both sides of an assignment
Data structures
Bit handling functions (note 3)
VMS specific OPEN keywords (note 4)
$, O, Q, or Z edit descriptors in FORMAT statements (note 5)
%VAL, %REF, %DESCR (note 6)

Certain other non-standard features are dealt with later sections. The DEC Fortran manual has all extensions to the ANSI standard printed in blue, and these features should be avoided unless expressly sanctioned in the Starlink standard. If an extension has to be used for any reason (because the required result simply cannot be obtained any other way) this should be highlighted with comments. If you are unsure about whether your program conforms to the Fortran 77 standard, try compiling it with the /STANDARD qualifier (on VMS systems).

Notes:

(1) INTEGER*2 and BYTE data types may be used where they are essential to deal with input data sets containing 8 or 16 bit values.

(2) Many programmers refuse point-blank to stick to 6 character names, and some even like using whole sentences with underscores between the words. Opinions vary on whether long names help readability all that much, but what is certain is that several computer systems of considerable potential importance to UK astronomers do **not** support long names (for example some mainframes, certain attached processors, some workstations). Some computers accept names longer than 6 characters but only a little longer; others accept long names but ignore all but the first n characters.

Though names internal to a Starlink program must be no more than 6 characters long, the use of names of more than 6 characters is permissible (and indeed strongly encouraged) in the case of program unit and labelled COMMON block names, in order to reduce the chances of name clashes between different facilities (especially the C run-time library). Such names should be constructed by prefixing the name proper, which must be no more than 6 characters long, with a facility name of the form 'FAC_'. An example is SGS_OPEN: the facility name is SGS and the name of the routine OPEN. For the utmost portability, it is wise to limit the name proper to 5 characters rather than the full 6; thus SGS_ZSIZE can readily be preprocessed to SZSIZE (for example) if merely dropping the SGS_ gives name clashes. Within a given package, the name proper should be unique even if different facility names are being used. A further Starlink convention is that routines called only internally within a package are given names prefixed with 'FAC1_' rather than simply 'FAC_'. Application programmers should never use existing facility names (see Appendix A) unless contributing solicited and debugged software for inclusion in the facility concerned.

Fortran 90 allows names of up to 32 characters.

(3) Use of bit manipulation routines will be unavoidable in instrument specific processing but is prohibited for general applications. (Note that Fortran 90 contains bit-manipulation facilities.)

(4) Certain VAX extensions to the OPEN and INQUIRE statements overcome serious omissions from the Fortran 77 standard and may be used if necessary:

- READONLY
- CARRIAGECONTROL='LIST'
- ACCESS='APPEND' (worth avoiding)

Highlight these VAX-specific items with comments, and use them only in short routines which can be re-coded for other platforms.

(5) In FORMAT statements, O (octal) and Z (hexadecimal) edit descriptors may be required for the early stages of instrument specific processing, but must not be used otherwise. Q (number of characters input) and $ (suppress carriage return) should be avoided. Sometimes they have to be used, typically at just one place in an application, and when this happens they should be prominently commented as VAX-specific and preferably isolated inside a short routine which can be re-coded for other platforms.

(6) The DEC Fortran %VAL etc. may be required to interface to 'software environment' routines. (ADAM is one such software environment – see section 4.) They should not be used for any other purpose.

Stick to the Fortran 77 character set !! 2

The Fortran 77 standard permits use of only the following characters (except in comments and character strings):

```
0-9, A-Z, space, currency symbol, +-*/(),'.:
```

Though a few other characters are sufficiently common for trouble to be unlikely it is best to avoid them even in comments as they may appear different on different terminals and printers – hash and pounds sign are particular examples of this. Backslash is best avoided because it has a special meaning in almost all Unix Fortrans. Applications should not depend on the presence of nonstandard characters – stick to the Fortran 77 set.

An exception is made for lowercase letters a-z, which are (a) permitted in program source code, and (b) encouraged in output messages. The recommended style is to use upper- and lowercase freely in comments but to use uppercase in the Fortran proper. If you must use lowercase characters in Fortran statements, it is important to be consistent, not just to make the code easier to read, but also because the Fortran compilers on some Unix systems regard upper- and lowercase characters as different (so `N` and `n` would be different entities for example).

The character set allowed by Fortran 90 includes `_ ! " % & ; < > ? [ ]`.

Take care with collating sequence. The letters A-Z appear in the expected *order*, but you cannot assume that they are *contiguous* in the collating sequence. Similar remarks apply to 0-9. Do not assume that the numbers appear before the letters. Assume nothing about the collating sequence of punctuation characters, except that blank comes before both the letters and the numbers.

Input must not be case sensitive !!! **3**

Applications must not distinguish between upper- and lowercase in data they receive.

Avoid end of line comments ! **4**

End of line comments (preceded by '!') are not in the Fortran 77 standard, and are best avoided, though they are in Fortran 90 and are supported by many current compilers. There is some justification for using them with data declarations. If used at all they must be neatly aligned by means of spaces, **not** TAB characters (see the next rule).

Don't use TABs !!! **5**

The use of TAB characters is prohibited, both in source code and as required data input to a program. Though their use in source code may give an *illusion* of good layout with a minimum of effort, tab settings on different terminals and printers will in general be different and will not reproduce what appeared on the screen when the program was being written. With most editors it is easy to program rarely used keys or special function keys to produce multiple spaces – for instance 3 and 6 – which will reduce the number of keystrokes when entering Fortran.

Arithmetic IF banned !!! **6**

The use of the arithmetic IF:

```
IF (X) 10,10,20
```

is prohibited. It offers even more opportunity for unstructured programming than the discredited GO TO and, moreover, doesn't read naturally in English. (The arithmetic IF is a fossil of one of the machine instructions on the computer for which Fortran was first devised, the IBM 704.)

Use the computed GO TO sparingly ! **7**

Don't use the computed GO TO unless you have to, and then only to implement a properly laid out 'case' construct. In such instances, a good plan is to use comments to indicate the case for each destination. For example:

```
*  Switch according to command
      GO TO ( 1000, 2000, 3000, 9000 )
*               Go  Hold  Stop  Abort
```

ASSIGN banned !!! **8**

The ASSIGN statement is prohibited. It offers endless opportunities for incomprehensible code, and is a peculiar historical feature not found in other languages. Though it can be used in conjunction with the assigned GO TO (also banned) to implement a sort of internal subroutine call, the temptation to do clever things with saved or multiple return pointers etc. will prove too great for some practitioners – so the feature is outlawed by Starlink. (See also rule 29.)

Don't use PAUSE !!! **9**

PAUSE is prohibited. It produces different results on different sorts of computers and is sure to interfere with whatever software environment you are using.

STOP, RETURN, ENTRY banned !!! **10**

STOP, RETURN and ENTRY are all prohibited.

STOP produces a message that is usually uninformative and is incompatible with software environments. It is also redundant – the main program's END statement causes program termination.

RETURN is redundant – a subprogram's END statement returns to the caller.

The key reason for prohibiting STOP and RETURN is to force the programmer to have one exit point only per routine, at the end, to aid debugging. This complements the ban on multiple entry points (and hence on alternate RETURN) in satisfying structured programming requirements.

Note that the ban on ENTRY and RETURN means that multiple ENTRYs and RETURNs are similarly prohibited.

Don't use DIMENSION !!! **11**

The DIMENSION statement should not be used; arrays must be given explicit type declarations. For example:

```
REAL A(512,512)
```

must be used, rather than:

```
DIMENSION A(512,512)
```

Don't try to overprint !!! **12**

In FORMAT statements, the use of the printer control character '+' is prohibited. Assume that overprinting is not possible; this is almost always the case.

Don't devise your own character handling mechanisms !!! **13**

Characters must be handled only by the character string mechanisms of Fortran 77. Private mechanisms (involving for example integer arrays) are banned.

Remember to use SAVE !!! **14**

The SAVE statement must be used in all cases where subprograms assume that their local variables retain their values between successive invocations. If you don't do this, **your programs**

will not run correctly on some computers even though they happen to work on VAX/VMS and other compilers that allocate static storage by default. Note that SAVE is also required when using labelled COMMON, unless the COMMON block is declared in at least one other program unit in the calling chain (the main program for instance); in this case the SAVE specifies the COMMON block itself rather than the individual variables. Avoid using the form of the SAVE statement where the variable and COMMON block names are omitted.

The rules in the ANSI standard concerning SAVE are rather complex and no further attempt will be made to summarize them here.

Statement labels on FORMAT and CONTINUE only !!! 15

Statement labels are permitted on FORMAT and CONTINUE statements only. They should be less than 10000, increase monotonically through the routine, occupy columns 2-5 only and be consistently justified, left or right.

Declare everything explicitly !!! 16

All variables, parametric constants and functions (except the Fortran 77 generic functions) must be given explicit type declarations. To force this, it is strongly recommended that the IMPLICIT NONE statement be used, with the additional advantage that undeclared arrays, functions, etc., and many typing errors are exposed during debugging.

The use of a consistent naming scheme to indicate type, though unfashionable, is worth considering. The standard Fortran convention of an initial I-N can be used to highlight integers, and where REAL and DOUBLE PRECISION variables are being mixed it can be helpful to reserve initial D for the latter. CHARACTER, LOGICAL and COMPLEX entities are more rarely confused and there is usually less need for naming conventions.

Arrays appearing in COMMON must have their type and dimensions declared in separate statements which precede the COMMON declaration. For example:

```
REAL ARRAY(512)
COMMON /fac_BLOCK/ ARRAY
```

The types of subprogram arguments must be specified explicitly.

Use CHARACTER∗(∗) !! 17

Dummy arguments of type character, and character functions, should be declared CHARACTER*(*) in order to pass the length implicitly.

Keep I/O out of mainline code !! 18

Portability considerations make it desirable that all Fortran I/O (except to internal files) be encapsulated within suitable primitive subroutines.

Avoid list-directed I/O, which may give different results on different machines. There are library routines for input and output conversions (see sla_DFLTIN in SLALIB for example); use these.

In many 'software environments' (ADAM for example – see section 4), Fortran I/O to the terminal is not permitted, and any Fortran I/O to files should be done in accordance with any special rules of the software environment concerned.

Don't use literal I/O unit numbers !!! 19

The external unit identifiers (logical unit numbers) used in OPEN, READ, WRITE, etc. must not be hardwired – the number itself should appear at a maximum of just one place in the program, in a PARAMETER statement. Keep to the range 0-255.

The various 'software environments' usually have ways of supplying such logical unit numbers from a pool, and this is recommended.

Don't use mixed mode arithmetic !! 20

Avoid mixed mode arithmetic or change of type across an assignment statement. Use explicit type conversions; for example:

```
X = REAL(N-1)
```

rather than:

```
X = N-1
```

Changing from one number format to another is a very significant event in an algorithm and should be clearly expressed. Great circumspection should be employed when deciding where such conversions should occur.

Data type must not change across a CALL !!! 21

Data types must not be mixed across a function reference or subroutine call, even in cases where it appears to work on a VAX (between REAL and DOUBLE PRECISION for example).

LOGICAL and INTEGER are different !!! 22

LOGICAL and INTEGER usage must not be mixed. For example:

```
INTEGER JFLAG
   :
   :
IF (.NOT.JFLAG) ...
```

is not permitted.

Use only INTEGER DO-variables !! 23

Don't use REAL or DOUBLE PRECISION DO-variables, just integers. The way the iteration count is determined and the measures taken to avoid cumulative rounding errors are important parts of the algorithm and should be spelt out.

Don't use EQUIVALENCE !! **24**

EQUIVALENCE should only be used if there is a very good reason, and even then only in a straightforward way. In particular:

- Do not use EQUIVALENCE to 'extend' an array.
- Do not EQUIVALENCE anything in COMMON.
- Do not EQUIVALENCE entities of different data types.

3.2 Design

Use Structured Programming !! **25**

The principles of *structured programming* should be followed except in cases where this obscures the program logic.

Programs written using structured programming techniques are built out of three basic elements: (a) processing sequences, (b) decisions, and (c) loops. Each of these three elements has only one entry point (at the top) and only one exit (at the bottom), and its relationship with the data it uses is clearly defined. The whole program consists of a hierarchy of these elements. In Fortran 77, (a) is just a series of statements, (b) is an IF THEN ELSE construct and (c) is one of the DO constructs. The most conspicuous feature of structured programs is the absence of (or, depending on the suitability of the programming language being used, the relative absence of) GO TO statements. Such programs are incomparably easier to follow than ones containing tangled GO TO logic, but may be harder to write, especially if you have not completely grasped what you are trying to do before you start coding.

To conform to the Starlink standard, a program must *not* show signs of having been written 'bottom-up', or having 'grown like Topsy'. It can be a good plan to begin by writing the program in a 'structured English' pseudo-language; this can then become, as comments, part of the program.

The SPAG utility (SUN/63) can re-arrange GOTO- and arithmetic-IF-infested code into block-structured form, and this is a good first step when working up old code into a maintainable form.

Keep program units small ! **26**

Modules should be small wherever possible, ideally a page or less (not including prologue comments). Programs which, despite being longer than this, are nevertheless not difficult to follow – because they consist of a simple top-to-bottom flow or contain a single main loop with a simple flow inside it — may be acceptable.

Avoid the GO TO !! **27**

The GO TO must not be used unnecessarily. Use the DO and IF structures unless they make the program difficult to follow. There are cases where use of GO TO is justified by the need to jump downwards in a program as the result of some exceptional condition, and use of IF ... ELSE IF together with indenting of code would give a less satisfactory appearance. If you have to form structures using the GO TO, clarify what is going on by means of commenting and indenting.

If you are prepared to use the non-ANSI-standard DO WHILE, you can write a loop which includes tests for exit and repeat conditions, without using the GO TO:

```
LOGICAL LOOP
      :
LOOP = .TRUE.
DO WHILE (LOOP)
      :
  IF (‘exit’ condition) THEN
    LOOP = .FALSE.
      :
  ELSE IF (‘next’ condition) THEN
      :
  END IF
END DO
```

Don’t loop using GO TO !! **28**

Do not implement loops by jumping backwards using GO TO.

Don’t use GO TO to drive internal subroutines !!! **29**

Internal subroutines driven by GO TO statements are prohibited, whether using the assigned GO TO or not. (See also rules 8 and 28.)

FUNCTIONs must not have side effects !!! **30**

FUNCTION subprograms must only be used when a single argument is returned and there are no side-effects. All other procedures should be SUBROUTINE subprograms.

Minimize use of COMMON !!! **31**

No unnecessary use of COMMON must be made. The use of COMMON can lead to code which is difficult to maintain. It should not be used simply as a lazy way of passing arguments to subprograms; when it is used for passing arguments there must be a good reason and every item referenced must be mentioned in the prologue comments just as for formal arguments.

The names of entities in COMMON must not change from one program unit to another; the use of dummies is discouraged.

All COMMON block source should be stored in separate files from the rest of the source, and be inserted using the INCLUDE statement.

Blank common must not be used – only labelled common (with a fac_ prefix to the name, as described in rule 1 note 2).

Note that the Fortran 77 standard does not permit initialization of common blocks via DATA statements in main programs or normal subprograms; BLOCK DATA must be used. (Avoid potential linking difficulties when using BLOCK DATA by referring to the name of the BLOCK DATA module in an EXTERNAL statement in at least one of the subprograms that makes use of the relevant COMMON block.)

No garbage !!! **32**

The following are prohibited:

unused statement labels
unused declarations
unused FORMAT statements
unreachable code

FORCHECK (see SUN/73) will detect all of these conditions.

Be device-independent !!! **33**

Do not make unwarranted assumptions about the hardware being used, especially the properties of terminals or printers. If your application requires use of special features, either include mechanisms external to the program which can be configured to match the hardware available, or have the user make explicit assertions. A common crime, for example, is to assume that the terminal is ANSI standard (VT100 etc.) and to output escape sequences in order to control scrolling or to output large characters etc.; these will have unpredictable effects on other types of terminal. A solution is to include in your program's repertoire of commands one which allows the user to announce that he is on an ANSI terminal and only to output ANSI escape sequences if that command has been invoked.

Be environment independent !!! **34**

Be careful when specifying filenames or device names in application programs. There are, unfortunately, no platform-independent ways of handling such names, but difficulties will in practice be minimized if filenames (i) contain no uppercase letters, (ii) are no longer than eight characters or eight plus a period and a further three, and (iii) contain only letters, numbers and a maximum of one period.

When soliciting a file or device name from the user, make no assumptions about its format and pass on the string received (to an OPEN statement typically) without altering it or trying to deduce things from it.

Be aware of floating-point limitations !!! **35**

Programs must not rely on more than the following ranges and accuracies:

type	range	accuracy
`REAL`	$\pm 10^{\pm 38}$ & zero	6 dp
`DOUBLE PRECISION`	"	14 dp

Don't access uninitialized variables !!! **36**

Never access an uninitialized variable. Relying on variables being initially zero (usually the case for VAX Fortran) is **not** permitted.

Don't use VAX system service or RTL calls !! **37**

Any programs which call system services (QIO etc.) or Run-Time Library routines will clearly be non-portable, and these techniques should not in general be used. This may not always be possible, and where a VAX-specific call is required it should be kept separate from the application proper by encapsulating it in a subprogram of its own. Occasional use in mainline code is excusable if removing the call happens still to result in a runnable program (albeit of reduced capability), and if prominent commenting is used to highlight the VAX-specific code.

3.3 Quality

Use meaningful names !! **38**

Use sensible names which (within the 6-character limit) offer some indication of the meaning of the entity concerned. The use of I, N, W, X, etc. for purely local and temporary use is permissible; daft or misleading names are banned.

Don't re-use variables !! **39**

Use a variable for a specific purpose; use a different one if the meaning has changed.

Define sizes parametrically ! **40**

In general, the sizes of tables, queues, buffers and work arrays should be defined parametrically. For example:

```
*  Reference star positions
      PARAMETER (NREFS=1000)
      REAL REFS(2,NREFS)
```

If a size is required in more than one program unit, it should be declared in an INCLUDE file.

Minimize rounding errors !! 41

In cases where control over execution order within a statement is important in order to minimize rounding, this can be achieved by means of otherwise redundant parentheses. For example, if DELTA1 and DELTA2 are small compared with B, their sum could be computed without avoidable loss of precision as follows:

```
A = B+(DELTA1+DELTA2)
```

Validate inputs !!! 42

Everything coming into the program from outside must be validated.

There are only two exemptions permitted:

- Subprograms called exclusively by programs which can guarantee to present valid arguments.
- Subprograms which explicitly put the onus of validation onto the caller to achieve some real efficiency advantage.

Don't output error messages at too low a level ! 43

Subprograms will be more flexible if they do not output error messages themselves but instead leave this to the caller by returning a status. (Using PAUSE and STOP is even worse – see rules 9 & 10.)

Applications running under a 'software environment' (for example ADAM – see Section 4) should adopt the error reporting strategy provided by that environment. The ERR_ and MSG_ packages (SUN/104) provide error reporting that works both in ADAM-based and freestanding programs. These packages enable subprograms to report errors in detail while still allowing higher levels to decide whether or not messages will actually appear on the user's screen.

Don't terminate by count !!! 44

Terminate input by end-of-file or by a special end record, **not** by count.

Don't test REALs for equality !!! 45

REAL or DOUBLE PRECISION variables must never be tested for equality against non-zero numbers. Testing for zero is also frowned on by most experts but can be difficult to avoid.

(But see remarks about bad-pixel handling in ADAM applications – section 4.)

Use the standard order for arguments !! **46**

The order of arguments in subroutines should be as follows:

Given
Given and altered
Returned
Status return

Note that the Fortran 77 standard prohibits use of the same actual argument more than once when calling a subprogram which gives one of the arguments concerned a new value. Thus a subroutine P(A,B,R), which computes some function of A and B and finally returns it in R, must not be called with arguments (X,Y,X) even though this technique happens usually to work in VAX Fortran (for example).

Use generic names ! **47**

Use the generic names of intrinsic functions, for example MAX(A,B) rather than AMAX1(A,B).

Don't re-invent existing routines !! **48**

Starlink library routines should be used whenever possible, rather than writing routines which duplicate (or almost duplicate) the functions of existing Starlink routines. It is permissible to *adapt* Starlink code where the required changes are substantial and the code uses only published interfaces; the source must be seamlessly blended with the new application and be given a new name.

Do not include in a package copies or slight variants of Starlink routines in an attempt to make the package self-contained. Assume the availability of the required Starlink library, and if the package has to be run on an installation which does not have the Starlink software collection make proper arrangements with the Starlink Software Librarian to have the up-to-date libraries sent there.

Avoid using the Run-Time-Library routines available on VAX/VMS and other platforms. Many of the facilities included in these libraries are also provided by POSIX, an industry-standard Portable Operating System Interface. Fortran-callable versions of many of these routines are provided in the Starlink PSX library.

Many of the libraries which form part of the ADAM Software Environment (see Section 4) are available in two forms: an ADAM version and a free-standing version. Programmers are strongly recommended to use these libraries even where there is no immediate intention of running under ADAM.

Don't write clever code !! **49**

Programs must not be obscure in the name of efficiency. The first version of the program should be coded for clarity rather than efficiency (within reason). If there are found to be worthwhile (i.e. obvious to the user) improvements in efficiency possible, at the expense of clarity, then

changes can be made. The reduction in clarity must then be made good by extra comments – perhaps including the original code. (If the changes do not reduce the clarity of the program, then it was badly written in the first place.)

3.4 Presentation

Begin modules properly !! **50**

The first statement in a program unit must be one of the following: PROGRAM, FUNCTION, SUBROUTINE or BLOCK DATA. Preceding comments are discouraged, to avoid confusion (either to the reader or to software) over where each new module begins in a concatenated sequence of such modules. (Likewise, comments following an END statement are not allowed.)

All program units must have sensible and self explanatory names, as far as is possible given the limitations imposed by the 6 character or fac_ + 5 character rule (see rule 1 note 2).

Include prologue comments !!! **51**

There must be one or more blocks of comments at the beginning of every program unit, which together form a 'prologue'. The prologue must include the name of the program unit, a brief description of what it does, and full details of its interactions with the calling environment. The author, organization, and date should be given, expressed compactly.

To enable automatic recognition, each block of prologue comments must begin with a comment which starts $*+$ and ends with a comment which starts $*-$. Elsewhere in the program, do not have any statements with $+$ or $-$ in the second column.

Main programs must have a prologue which says what files will be read or written, and gives the I/O unit identifiers used (numbers or symbols).

Subprogram prologues must list all the arguments, clearly describing their function, type (unless obvious), units (where applicable) and any special properties (e.g. whether an array). The words 'given' and 'returned' are recommended for direction, rather than 'input', 'output', 'source', 'destination' and other possibly ambiguous terms. Access to COMMON blocks should be treated similarly, with every item referenced fully described.

It is recommended that the names of all subprograms called (except the Fortran 77 intrinsic functions) be given in the prologue. It is extremely important that all the information given in the prologue is accurate and up to date. Prologues can be automatically extracted from source held in text libraries by using the LIBPRE facility (see SUN/8).

Example:

```
      SUBROUTINE sla_NUT (DATE, RMATN)
*+
*     - - - -
*      N U T
*     - - - -
*
*  Form the matrix of nutation for a given date (IAU 1980 theory).
*
```

```
*  (double precision)
*
*  References:
*     Final report of the IAU Working Group on Nutation,
*                                  chairman P.K.Seidelmann, 1980.
*     Kaplan,G.H., 1981, USNO circular no. 163, pA3-6.
*
*  Given:
*     DATE   dp         TDB (loosely ET) as Modified Julian Date
*                                        (=JD-2400000.5)
*  Returned:
*     RMATN  dp(3,3)    nutation matrix
*
*  The matrix is in the sense   V(true)  =  RMATN * V(mean) .
*
*  Called:   sla_NUTC, sla_DEULER
*
*  P.T.Wallace   Starlink   10 May 1990
*-
```

Note, however, that much of the freedom implied by the above recommendations may not be available to programs which conform to the documentation standards of a particular 'software environment', especially if automatic documentation facilities are involved. For details of the prologue requirements of the ADAM software environment, see section 4.

Begin comments with ∗. **52**

The comment symbol '∗' should be used in preference to the old-fashioned 'C'.

Use blank lines to improve layout. **53**

Blank lines should be used freely to break up code.

Make comments stand out from code !! **54**

Comments must be clearly distinguished from code. The recommended style is begin the text some fixed amount (1-3 characters) to the left of the code, and to follow the same indenting scheme in comments and code alike. An alternative style, less likely to be spoiled by editors and reformatters which change the code indentation, is to begin the comments always in column 3 or 4. Comments should use lowercase freely, but the code should be in uppercase.

Comments beginning in column 7 are strongly discouraged, even if preceded by `C*****` and the like.

Be considerate !!! **55**

Every effort must be made to present the program in the clearest and most agreeable way **from the point of view of the support programmer** – who has a much more difficult job to do than

the original author. The layout must be neat, clean and consistent, and there must be liberal commenting, both at the start of each module and within the code.

The comments, which should in general precede the code they describe:

- must accurately reflect the behaviour of the program;
- must be detailed without merely stating the obvious;
- must not be cryptic – 'clues' are not enough;
- must be in English, without spelling or grammatical errors;
- must not contain attempts at humour, meretricious phraseology, catchwords, superfluous pleasantries, private jargon and clever abbreviations.

Modifications must blend in !!! **56**

When a program is changed, the modifications must be 'invisibly mended' into the coding; scars are not permitted. The original style and conventions of the program must be preserved – which will be much easier and more palatable if the program conformed to the Starlink standard in the first place.

There should be no need for elaborate and unsightly 'revision-flag' schemes except, perhaps, during debugging or where software that runs on more than one machine has machine specific inclusions.

Fix bad code !!! **57**

Bad code must be rewritten, not merely commented.

Use ':' as the continuation character ! **58**

Only one continuation character should be used, preferably ':'. (If you do choose something else, remember it should be in the Fortran 77 character set. The dollar sign is a popular choice because it has no syntactical function in ANSI Fortran outside a character string.

Using 1,2,3 ... for successive lines is specifically discouraged as it leads to annoying editing problems (and, after all, protection against shuffling is no more necessary than for any other region of the program).

Lines longer than 72 characters are not allowed !!! **59**

Program lines (including comments) must not be longer than the legal 72 character maximum (even though compiler options such as /EXTEND_SOURCE on VAX/VMS can override this maximum). A lengthy statement (that for some reason cannot be broken up into several shorter statements) should be split at natural breakpoints and the pieces neatly aligned. Statements must not be split between lines simply by exploiting the break at column 72. Take care with character constants, where this can easily happen inadvertently – especially inside format specifications.

Use spaces to improve readability. **60**

Spaces should be used freely within Fortran statements to make them easier to read. A space before and after the equals sign in an assignment statement is particularly recommended.

Use indenting to show structure !!! **61**

It is essential that the structure of a program be reflected in a consistent and pleasing indentation scheme.

A suggested scheme is as follows. The normal starting columns for comments and code should be 5 and 7 respectively. For each block between a DO and END DO, or between IF, ELSE IF, ELSE and END IF, both comments and code should be indented a further 3 columns. Blank lines should be used freely to improve presentation further.

This scheme is used in the following example. This is, of course, only one acceptable layout, and tastes vary.

```
*   Reset bad pixel count
      NBLEM = 0

*   Look at all pixels except edge
      DO IY = 2, NY-1
         DO IX = 2, NX-1

*         Reset blemish flag
            IBLEM = 0

*         Pick up pixel
            PXV = A(IX,IY)

*         Expected value = mean of surrounding eight
            S = 0.0
            DO J = IY-1, IY+1
               DO I = IX-1, IX+1
                  S = S+A(I,J)
               END DO
            END DO
            EV = (S-PXV)/8.0

*         'Blemish' criterion
            BLEM = SIGMAS*SQRT(ABS(EV))

*         Decide whether in star or not
            IF (EV.GT.STAR) THEN

*              In star
                IF (PXV.LT.EV/4.0-BLEM) IBLEM=1
                IF (PXV.GT.4.0*EV+BLEM) IBLEM=2
            ELSE

*              Not in star
```

```
                D = PXV-EV
                IF (D.LT.-BLEM) IBLEM=3
                IF (D.GT.BLEM) IBLEM=4
             END IF

*         Override if negative or small
             IF (PXV.LT.DIM) IBLEM=0
             IF (PXV.LT.0.0) IBLEM=5

*         If blemish, fix and report
             IF (IBLEM.NE.0) THEN
                B(IX,IY) = EV
                NBLEM = NBLEM+1
                WRITE (LINE,'(''Blemish at    ('','     //
     :                      'I4,'','',I4,'')'',''        //
     :                      'G16.4,''  changed to'','   //
     :                      'G16.4)') IX-1,IY-1,PXV,EV
                CALL WRUSER(LINE,JSTAT)
             END IF
          END DO
       END DO

*   Final report
       WRITE (LINE,'(I6,''  blemishes removed'')') NBLEM
```

Put FORMAT statements inline ! **62**

FORMAT statements should be inline (following the appropriate READs or WRITEs) rather than, for example, massed at the end of the program. When a given format specification is required once only it can be incorporated into the READ or WRITE itself as a character constant, unless the result is less clear (for example because of multiple apostrophes). When using this technique, be sure the constant doesn't continue through multiple lines (see rule 59, and the example in the previous rule).

4 GRAPHICS

Plot using an approved graphics package !!! **63**

All graphics operations must be carried out, ultimately, by the GKS (SUN/83) and IDI (SUN/65) libraries.

GKS: Direct use of raw GKS in applications is discouraged, in favour of using one of the Starlink-supported higher-level packages as an intermediary. These include SGS, the NCAR utilities (and the SNX routines), the NAG Graphical Supplement, and PGPLOT. (Though the Starlink version of the proprietary package MONGO uses GKS, use of callable MONGO in application programs is not encouraged in this standard as its functions are provided elsewhere. MONGO is supplied by Starlink principally for interactive use, and is in any case being phased out in favour of the Starlink PGPLOT-based PONGO package.)

SGS: Where only low-level facilities are required (lines, character strings, formatted numbers, simple shapes, but not complete axes or whole graphs) the SGS package (SUN/85) is recommended. SGS consists mainly of convenient packaging of GKS functions, including easy control of plotting zone and a flexible workstation naming scheme, and can be used in conjunction with direct GKS calls.

PGPLOT: In standalone applications where complete graphs are to be drawn and convenience is more important than flexibility, the Caltech PGPLOT package (SUN/15) may be used. PGPLOT was expressly designed to meet the requirements of astronomers, and is especially good at publication-quality laserprinter output. PGPLOT exists in two forms, one using GKS and the other with native low level and device driver layers, which gives it certain portability advantages for astronomers wishing to send their applications overseas. The two forms are of comparable performance except in greyscale output, where GKS is substantially faster. PGPLOT is moderately flexible, and gives good access to colour and multiple text fonts. However, direct access to GKS during use of PGPLOT is prohibited, limiting the package's capabilities to those explicitly provided by PGPLOT itself.

NAG: The NAG Graphical Supplement (currently available only at certain sites – see RAL LUN/45) also draws complete graphs, in a number of useful though spartan formats. It has portability advantages, however, especially for users of UK university computing facilities. Some direct access to GKS is possible.

NCAR & SNX: The NCAR package, (SUN/88) supplied by the National Center for Atmospheric Research, is widely available and runs on many different types of computer (n.b./ recent versions are proprietary). The Starlink SNX routines (SUN/90) allow the NCAR routines to be used in conjunction with SGS, and direct use of GKS facilities is also possible. The most important of the NCAR utilities is AUTOGRAPH, which draws complete plots of one variable against another. Great flexibility is available, and effects can be achieved which are impossible with other packages of comparable level. For example numeric labels can be intercepted and replaced; opposite axes can have different scales (including non-linear ones), lines can have embedded captions, and so on.

IDI: The IDI package (SUN/65) differs in its objectives from all those mentioned so far, which are mainly oriented towards line-drawing. IDI is a low-level interface for image display devices, and supports concepts absent in GKS, for instance re-configurable pixel memory. Compared

with the GKS-based packages, IDI's drawing facilities are rather primitive, but IDI comes into its own for highly interactive applications, involving special cursors, pan/zoom, blink etc.

AGI & GNS: Two supporting packages are the Graphics Database library AGI (SUN/48) and the Graphics Name Service library GNS (SUN/57). AGI remembers what has been plotted where and allows one applications to access the plotting of another. GNS is a unified naming scheme which is supported across all the Starlink graphics packages. Application programmers should be fully aware of what these packages do, should make full use of these facilities, and should not duplicate their functions in application code.

Mixtures: With care, several of the plotting packages can be used in conjunction with one another. The various packages interrelate with each other and with applications in ways which are unfortunately too complicated and varied to permit a simple diagram to be drawn or a simple set of rules to be laid down. The techniques available depend on the particular combination of packages involved, and guidance should be sought from the documents describing those packages. The SGS, SNX, NCAR and GKS packages are more miscible than some of the others. It is normally safe to use the packages sequentially; the dangers lie in calls to one package deranging the context of one of the others. The difficulties and complications of this area are a direct result of offering the utmost flexibility. Those who prefer safety and simplicity would be well advised to stick to a single package (for example PGPLOT) used in a straightforward way.

Don't interpret graphics device names or numbers !!! **64**

Do **not** examine in applications the GKS/SGS/GNS workstation type or name in order to control the behaviour of the program. For example, you are not allowed to say "the GKS workstation type is 3200, therefore this is an Ikon image display, therefore colour is available". This information must be obtained by calling the appropriate GKS, SGS, PGPLOT or GNS enquiry routines; if the property you are interested in is not available through such enquiries then devise some mechanism external to the program, or solicit information from the user.

5 THE ADAM SOFTWARE ENVIRONMENT

5.1 Introduction

Applications running under Starlink's recommended software environment ADAM should in most respects be programmed according to the rules given so far. However, ADAM has a number of special requirements which may mean that one of the general rules has to be reinterpreted – in some cases strengthened, in others relaxed. There are, in addition, several new rules which do not have to be obeyed when writing non-ADAM applications.

Programming standards for ADAM applications written outside Starlink may, of course, differ from those laid down by Starlink.

5.2 Rules for programming ADAM applications

Initialize variables !!! **65**

It is **essential** that variables are initialized. Even the VAX's initialization to 0 cannot be relied upon as the task may or may not be reloaded between invocations. DATA statements must only be used to initialize data which will not change. (Emphasizes rule 36.)

Use the ADAM standard prologue !! **66**

ADAM standard prologues differ in some respects from the Starlink standard, allowing less freedom but giving more opportunity for the automatic production of documentation and help files. For details, see SUN/105 and SUN/110. (Modification of rule 51.)

Output messages via the MSG_ routines !!! **67**

Message output must be done using the ADAM message system MSG_ subroutines. The ADAM message system is described in SUN/104. A stand-alone version of the MSG_ package exists. (Supplements rule 18.)

Don't use $, %, ^ in messages ! **68**

The non-Fortran 77 characters $, % and ^ are used as escape characters in the ADAM message system, and special methods have to be employed if they are to be included in messages (see SUN/104). In general, it is better to avoid using them. (Reinforces rule 2.)

Report errors via the ERR_ routines !!! **69**

Error reporting must be done using the ADAM error system ERR_ subroutines and the ADAM error strategy should be employed. The ADAM error system is described in SUN/104. A stand-alone version of the ERR_ package exists. (Reinforces rule 43.)

Set STATUS on failure !!! **70**

All applications which fail must return to the environment with an error status value set. This is to enable the environment to detect the failure so that users can write procedures which take appropriate action. (Supplements rule 43.)

When setting STATUS, generate a message !! **71**

If a subroutine is entered with STATUS=SAI__OK but, during execution, sets the STATUS (other than by calling another ADAM routine), an appropriate error message must be generated using ERR_REP (see SUN/104).

Some routines have >6 character names. **72**

Some of the ADAM environment package subroutines have names and prefixes greater than 6 characters. Where it is necessary to call these, rule 1 note 2 must be relaxed.

Get parameters with the PAR_ routines etc. !!! **73**

All program parameters must be obtained using the parameter system PAR_ or pkg_ASSOC subroutines. (Reinforces and supplements rule 18.)

Use symbols when testing for bad pixels !!! **74**

A REAL or DOUBLE PRECISION variable may be equated to its corresponding bad-pixel value, though explicit bad-pixel values, e.g. -32768, are banned. The parameters VAL__BADx (see SUN/39), where x corresponds to the data type, must be used. (Relaxation of rule 45 for this special case.)

Avoid Fortran input/output !!! **75**

Use the environment facility packages MAG, FIO etc. wherever possible. (Reinforces rule 18.)

If it is necessary to use Fortran I/O, obtain and release logical unit numbers using FIO_GUNIT and FIO_PUNIT. (Supplements rule 19.)

Use symbolic names !!! **76**

Status values and package constants are given symbolic names such as PAR__NULL and DAT__SZLOC by INCLUDE files for each package. These symbolic names should be used on every occasion that the constant is required. Follow these conventions when developing your own INCLUDE files, and use file names in the INCLUDE statements which conform to the convention *fac_err* for error codes and *fac_par* for other symbolic constants, where *fac* is the facility name. Further advice on INCLUDE statements can be found in Section 5, *Writing Portable Programs*.

RETURN is permissible when testing status. **77**

The RETURN statement is allowed in the form:

```
IF (STATUS.NE.SAI__OK) RETURN
```

as the first executable statement in a subroutine. This avoids an extra, unhelpful IF clause and indentation. Alternatively, use a GO TO n, where line n is a CONTINUE statement immediately preceeding the END statement. (Relaxation of rule 10.)

In generic routines use only the standard tokens !! **78**

The preprocessor for generic routines supports special tokens used by the ASTERIX package (SUN/98), as well as ones for general use. Use only the standard tokens.

PAR__ABORT status (!!) must abort the application !!! **79**

An application must terminate if the *abort* response (!!) is made when a parameter has been requested. Note that this rule does not mean that the application has to test for the abort status after every parameter is obtained; the inherited status will look after that. What matters is the appearance to the user of the application, who should:

- not be re-prompted for the parameter,
- not be prompted for further parameters, and
- not receive additional error messages merely because the status was not OK.

An abort does not absolve the programmer from ensuring that the application closes down in an orderly fashion.

6 WRITING PORTABLE PROGRAMS

One of the key reasons for having a Starlink programming standard is to promote *software portability*. What is meant by this term, and why is it important?

6.1 Meaning of Portability

Other things being equal, it is clearly desirable for applications to be usable on different computers rather than be limited to just one type. Equally clearly, there may be a tradeoff between the extra trouble of ensuring that application code is highly machine-independent and the work of modifying or rewriting programs from time to time. The Starlink Application Programming Standard recognizes this tradeoff and allows the programmer to choose what degree of portability is appropriate, taking into account:

- the type of software;
- its life expectancy;
- who will support it long-term; and
- the extent to which the programmer is prepared to rely on infrastructure software provided by others.

To put the recommendations of the Starlink Standard in context, consider four degrees of portability, called here *Absolute Portability*, *Portable Fortran*, *Adaptable Fortran* and *Laissez-Faire*.

ABSOLUTE PORTABILITY is where application source code compiles and runs on all types of computer without any alterations whatsoever. Once the programmer has completed work on an application, the code need never again be touched. To achieve this result, the programmer must be fully insulated from the facilities offered by the platform. Because the Fortran 77 standard does not include all those things which applications need to do, and in any case compilers vary in their compliance with and interpretation of the standard or have bugs, it is not possible to rely on pure standard Fortran. (Similar arguments apply to other languages.) The classic solution is to write applications in a private programming language, and to accommodate differences between computers, compilers and operating systems by providing different versions of the language interpreter software. This approach has the benefit that for any new platform, once a new version of the system software has been written, unlimited quantities of application code will run. However, having to use the systems's own programming language provokes scepticism among users, introduces extra training needs, produces code which can only run within the system, and reduces the convenience and effectiveness of online source code debuggers. These drawbacks led Starlink to reject this approach.

PORTABLE FORTRAN, Starlink's recommendation, is to write applications in an industry-standard language – Fortran 77 – with controlled use of certain platform-dependent features as sanctioned by Sections 2 and 4 of the present document. Significant departures from standard Fortran (for example the use of `%VAL`) should be present in only a small minority of modules, with most routines in *de facto* standard Fortran. These departures can, if and when necessary, easily be edited using simple preprocessors like `forconv` or even by hand. Furthermore, the programs can be understood and modified by non-specialists.

ADAPTABLE FORTRAN, also embraced by the Starlink Standard, differs from *Portable Fortran* in the degree to which departures from ANSI Fortran are tolerated. While gratuitous use of platform-specific features is frowned upon, it is accepted that some use of such features will be convenient and relatively harmless. Programs of this general level of portability are easy to write and to adapt manually for new platforms as required.

LAISSEZ-FAIRE programming is where programmers can use whatever the current machine's Fortran compiler accepts – the objective is simply to have a program that works. If a new computer is introduced, authors can decide whether to adapt, rewrite or scrap their applications. This style of programming lies outside the Starlink Standard and is deprecated for anything more than a casual one-off.

The *Absolute Portability* and *Portable Fortran* categories presuppose substantial quantities of *infrastructure software*, libraries and utilities which leave the programmer free to concentrate on the application itself rather than worrying about user interfaces, error handling, input/output and so on. At the lowest levels within the infrastructure there is a small platform-specific kernel, which has to be rewritten for each new machine. The *Adaptable Fortran* and *Laissez-Faire* categories allow programmers to provide their own infrastructure if they wish.

Starlink's recommendation is to base programs on the various standard tools and libraries, and to aim for the PORTABLE FORTRAN level in applications code.

6.2 Why Portability Matters

Despite the fact that for its first decade Starlink supported just one platform – VAX/VMS – the importance of avoiding platform-specific software has been stressed from the beginning. There are two main reasons for this. Firstly, there were and are collaborating astronomical institutions using non-Starlink types of computer – Data General, Fujitsu, Perkin-Elmer, CDC, Cray and various Unix platforms – and it is useful if they can run Starlink applications, and programs written by Starlink users. The second reason is to enable existing software to run on different sorts of Starlink equipment – currently Sun and DECstation as well as VAX/VMS. Quite apart from further Unix-based platforms, it is also possible that fast specialized processors will be added to the existing Starlink systems, and there is interest in using various types of Personal Computer. Attention to software portability – which means resisting the temptation to use Sun and DECstation features now just as much as avoiding VAX dependency in the past – means great benefits in the long term.

6.3 Achieving Portability

Programmers who have followed the recommendations given earlier in Section 2 are likely to encounter fewer difficulties in adapting their code to run on new and multiple platforms than programmers who have not. Of those recommendations, the key one is to work only with ANSI Standard Fortran 77 and only to use a VAX extension or an extension available on some other platform when it is essential, or safe, to do so. This advice should be borne in mind when reading the following notes, many of which refer to problems that afflict code where non-standard Fortran extensions have been used. The notes concentrate on the specific problem of adapting VAX/VMS applications to run on Unix platforms but also serve to illustrate more general portability issues.

- ANSI FORTRAN 77 STANDARD: There are a number of violations of the Fortran standard that are allowed by the VMS compiler that will cause problems on a UNIX system; some are rejected by the compilers but others cannot be detected at compile time and will cause programs to fail.
  - *Overlapping character substrings:* The character strings on either size of an assignment statement must not overlap. If not detected at compile time, such an overlap will produce incorrect results on Unix platforms.
  - *Illegal string concatenation:* The concatenation operator // cannot be used in circumstances that would require the allocation of arbitrary amounts of dynamic memory at run-time. For example, a CHARACTER*(*) dummy argument of a subroutine cannot be concatenated with another character string in the argument list of a subroutine or function call. (The ANSI standard puts it thus: a passed-length character dummy argument may only be the operand of a concatenation operator within an assignment statement.)
  - *Mixing character and numeric data in a COMMON block:* Separate COMMON blocks are required for character data on the one hand, and numeric and logical data on the other. (Similarly, it is illegal to EQUIVALENCE character data with anything else.)
- INPUT/OUTPUT: The Fortran I/O system is not tightly enough specified to avoid problems with different implementations:
  - Most compilers have their own set of non-standard I/O keywords, especially in OPEN statements. If use of such keywords is unavoidable they must only appear in explicitly platform-dependent routines, not in the middle of large programs.
  - Compilers vary in their tolerance of illegal combinations of keywords, which must be avoided.
  - *I/O unit numbers:* On Unix platforms the I/O unit numbers 0, 5 and 6 refer to the standard error, input and output channels respectively and cannot be used for anything else. Furthermore, only those unit numbers can usefully be used for reading from and writing to the terminal; other logical units are buffered in a way that is inappropriate for terminal I/O.
  - *Version numbers:* The Unix file system does not have file versions; opening a file with `STATUS='NEW'` when the file already exists will either destroy the contents of the file or fail depending on the system.
  - *READONLY:* On the DECstation, the non-standard keyword `READONLY` is required in order to open a file that you do not have write access to. The Sun compiler issues a warning message if `READONLY` is used.
  - *RECORDTYPE:* On Unix platforms, opening an existing file with the non-standard keyword `RECORDTYPE='FIXED'` requires that the RECL keyword is used as well because unlike on VMS the file system does not store the record length in the file header.
  - *Unformatted direct-access files:* In the OPEN statement for direct-access files the Fortran standard requires the record length to be specified, by means of the `RECL` keyword. In the case of a formatted file, the length is in characters; however, the Fortran standard does not specify the units of length for an unformatted file. For unformatted files the Sun uses bytes, whereas the VAX and DECstation use numeric storage units (the space required to store a REAL or INTEGER value).

  - *Printer control codes:* In the OPEN statement, `CARRIAGECONTROL='LIST'` is non-standard and is not supported by some platforms. There is no machine-independent way of specifying whether a text file contains Fortran printer control codes or not, and the effect of typing out text files produced by Fortran programs or of reading such files into a text editor cannot be predicted. This problem must be handled through per-platform code variations or by using per-platform utilities for processing the files (for example the `fpr` command on the DECstation and Sun).
  - *Prompt strings:* There is no portable way of suppressing CR/LF after a message has been output, though the VAX, DECstation and Sun all use the non-standard '$' edit descriptor. Provision must be made for per-platform variations at this point in an application.
  - *Status:* The I/O status values returned by OPEN, CLOSE, READ and WRITE are non-portable and application code should avoid using them in anything more than a general way (or should use the ERR_FIOERR routine – see SUN/104). Unfortunately, it is not even possible to map the numbers from the different platforms onto a single adopted set of values since the conditions that each platform reports as errors are different.
  - *End-of-File:* A READ or WRITE statement which includes the ERR= specifier behaves differently on different platforms when end-of-file is encountered. The condition is treated as an error on the VAX and DECstation but not on the Sun. To comply with the ANSI standard, all platforms return an IOSTAT value of -1.

- DATA STORAGE ALIGNMENT: The VAX is unusual in imposing no restriction on the addresses of data; many architectures generate a hardware error if, for example, a floating point operand has an odd rather than an even address. Both the Sun and the DECstation do this and although both operating systems handle the error successfully and allow the program to continue, it is at the expense of both a huge execution time overhead and a mysterious message being output. COMMON blocks should therefore be arranged such that the longer data types always appear before shorter data types.

- THE BACKSLASH CHARACTER: Unix compilers treat the backslash character as an escape character (so that for example `\t` is translated into a tab character) and to insert a true backslash character the source must have two backslash characters (i.e. \ on VMS must be converted to \\ on Unix). The `forconv` tool (SUN/111) can be used to insert the extra backslash character when converting source code.

  Some compilers have a switch that turns off the special meaning of the backslash character but using this is unwise – see the remarks on compiler switches later.

- THINGS THAT WORK BY ACCIDENT ON VAX: There are a number of bugs that can go unnoticed on VMS but will cause programs to fail on other systems:

  - *Argument mismatches across subroutine and function calls:* For example a DOUBLE PRECISION argument passed to a subroutine expecting a REAL happens to work on VMS but is a bug.
  - *Uninitialized variables:* On a VAX uninitialized variables will be set to zero; on Suns and DECstations they will not (see section 2, rule 36).
  - *Missing SAVE statements:* On VMS the values of variables local to a subroutine are retained between calls to the subroutine. On other systems they may not be; on the DECstation, for example, it depends on compiler switches. (See section 2, rule 14.)

- THINGS THAT WORK BY ACCIDENT ON UNIX: Similarly, there are bugs that go undetected on many Unix systems which will cause problems on VAX/VMS. For example, if a CHARACTER*n argument is not declared as such in a subprogram, it is often possible to get away with this on Unix systems but not on VMS.
- UNUSED VARIABLES: The Unix compilers always complain about declared but unused variables. (See section 2, rule 32.)
- INCLUDES: INCLUDE statements, by their very nature, cannot avoid involvement with the syntax of file names which makes writing source code that will run on many machines with absolutely no change of source code difficult if not impossible. However, the following scheme keeps the changes to a minimum and allows what changes that may be necessary to be automated.
  - On the VAX call your INCLUDE files `xxxx.for` where `xxxx` is some name of your own choosing.
  - Include them with statement like:

    ```
    INCLUDE 'xxxx'
    ```

  - Either compile your code in the same directory as the included files are stored or define `xxxx` as a logical name.
  - On Unix call the INCLUDE file `xxxx` (with no file extension) and compile your code in the same directory.

  The INCLUDE files that are used when calling Starlink subroutine libraries have logical names defined on the VAXs so that, for example, SAE_PAR.FOR is included with the statement:

  ```
  INCLUDE 'SAE_PAR'
  ```

  On Unix the corresponding file is called `sae_par` and is stored in `/star/include` along with all the other Starlink INCLUDE files so that the INCLUDE statement must be changed to:

  ```
  INCLUDE '/star/include/sae_par'
  ```

  The `forconv` program described in SUN/111 will accomplish the conversion from VMS to Unix. The reverse operation can be done with a simple edit script.

  It is also possible, though not at present the recommended technique, to set up a *soft link* file pointing to the required INCLUDE file and to specify the name of the soft link file in the INCLUDE statement.
- ONE MODULE PER SOURCE FILE: Code that is going to be inserted into a subroutine library (a Unix *archive*) must have just one routine per source file before it is compiled. This is because the Unix equivalent (`ar`) of the VMS librarian does not split object files into separate modules when it inserts them into a library. The Unix command `fsplit` will split a Fortran source file into separate files.
- COMPILER SWITCHES: It is unwise to do anything requiring use of special compiler switches. There are sure to be problems in the future when someone – not the original author – compiles the program, in good faith, without the switch. Examples are the

switches that allow code to extend beyond column 72 (see section 2, rule 59) and to disable the special meaning of the backslash character.

- WHEN ALL ELSE FAILS: Unavoidable per-machine variations can be handled either by using the `forconv` preprocessor (SUN/111) or by using separate files. Where the latter technique is used, a code identifying the platform should be appended to the name:

suffix	*platform*
`_vax`	VAX/VMS
`_sun4`	Sun SPARCstation etc.
`_mips`	DECstation etc.
`_pcm`	PC/Microsoft
`_ind`	platform-independent substitute

and the file extension should identify the language in the normal way:

extension	*language*
`.f`	FORTRAN
`.c`	C

Thus, different versions of a Fortran routine `fsub` for, respectively, VAX and Sun, would be `fsub_vax.f` and `fsub_sun4.f`. Care must be taken not to exceed 15 characters or `ar` will truncate the file name.

A Reserved Facility Names

The following is a list of packages used by existing Starlink software. Software written by others than those actually responsible for developing or maintaining these packages must **not** use existing package names, even if the new routines resemble or complement the official set. Programmers may wish to submit their own package names for inclusion in the following lists.

ADAM	ADAM interface file handling
ADAMCL	ADAM Command Language
ADC	Astronomical Data Catalogues
AG*	AGI + interfaces to graphics packages
AIF	Auxiliary ADAM interface routines
ANT	ADAM networking
ARGS	ARGSLIB
ARY	ARRAY-structure access
AST	Asterix
CHA	CHART part of SCAR
CHI	Catalogue Handling Interface
CHP	Character Handling “Plus”
CHR	Character handling
CHT	CHART
CLV	ADAM command language variable system
CMP	HDS component handling
CNF	C ’n’ Fortran
CNV	Figaro data-conversion
CON	CONVERT package routines
CTASK	ADAM Ctasks
CTM	Colour-Table Management
DAT	HDS and extensions to it
DAU	HDS internal routines
DCV	Data conversion
DIA	DIAGRAM
DIP	Internal system for DIAGRAM
DSA	Figaro data-structure access
DSK	Disc output for PGPLOT
DTA	Figaro data-structure access
DTASK	ADAM Dtasks
DYN	Figaro dynamic memory routines

EMS	Error Message Service
ENG	ADAM Engineering interface
EOS	Extendable Object System
ERR	Error reporting system
EXC	HDS internal routines
FIG	Figaro general purpose
FIO	File I/O
FIT	FITS processing (including Figaro FITS)
FTS	KAPPA FITS library
GEN	General utility routines
GKD	Graphics dialogue routines
GKS	Graphics Kernel System
GNS	Graphics Name System
GWM	Graphics Window Manager
HDS	Hierarchical Data System
HELPSYS	ADAM help system
HLP	Starlink portable help system
ICH	ICL real-to-character conversion
ICL	Interactive Command Language
IDI	Image display interface to ADAM
IMG	Simple Image Interface
IOC	Low-level C magtape routines
IRA–IRZ	IRAS software
KPE	KAPPA environment packaging
KPG	KAPPA general routines
KPS	KAPPA specific routines
LEX	ADAM command line parsing
LOCK	ADAM file sharing system
LOG	ADAM logging system
MAG	Magnetic tape handling high level
MCH	Machine-dependent constants etc.
MESSYS	ADAM message system implementation
MGO	MONGO routines
MIO	Magnetic tape handling low level
MON	ADAM monitor parameter system
MSG	Message reporting system
MSP	Message System Primitives
NBS	NoticeBoard System

NDF	NDF access
NUM	Primitive data arithmetic
PAR	ADAM parameter system interface
PARSECON	ADAM interface file parsing
PRM	PRIMDAT
PSX	POSIX interface
REC	HDS internal routines
REF	HDS reference handling
RIO	Random-access I/O
REPORT	ADAM reporting system
SLA	Subprograms mainly concerned with positional astronomy
SGS	Simple Graphics System
SMS	ADAM Screen Management System
SNX	NCAR/SGS integration
SST	Simple Software Tools
STRING	String handling
SUBPAR	ADAM parameter system implementation level
TAP	Theoretical astrophysics library
TASK	ADAM task control
TCV	ICL type conversion
TEL	UKIRT clock handling
TPT	TPOINT
TRA	Trace
TRN	TRANSFORM coordinate transformation facility
UFACE	ADAM User Interface
UNI	Unit conversion utilities
UTIL	ADAM/VMS utility
VAL	Primitive data arithmetic
VAR	Figaro user variable routines
VEC	Primitive vector arithmetic
VIO	VDU I/O (Used by SCAR)

In addition to the above, the following blocks of facility names are reserved by Starlink:

DSx	x = 0-9,A-Z
SLx	x = 0-9,A-Z

Though not strictly facility names, the following prefixes are used internally within ADAM facilities and should be avoided:

ACT	ADAM task activation errors
ADM	ADAM general errors
PARSE	PARSECON errors
SAI	Starlink Applications Interface errors
USER	Error values available for users
VAL	Special values