OGIP Memo OGIP/92-012

OGIP Fortran Programming Standard

Koji Mukai

Code 668,
NASA/GSFC,
Greenbelt,
MD 20771

Version 1.0 (first release): 1992 Nov 19

SUMMARY

This memo contains various standards and recommendations regarding OGIP Fortran programs. One standard is the extensions to Fortran 77 that can be used, considering the features of existing compilers and the Fortran 90 standards. Another is the prolog comments that explain what the subroutine is, how to use it etc. This is followed by a general recommendations on programming style.

1 INTRODUCTION

Despite the considerable software development that has been going on at OGIP, many of which are distributed to the community, we have not had a group-wide policy on programming practices. Recently, we have had discussions on how rigidly we should conform to the ANSI Fortran 77 standard, the results of which are presented in §2. In §3, the major differences between two major brand of computers are discussed. We also deal with wider issues of programming style, possibilities within Fortran 77 that are not good, areas where the language standard is not specific enough, etc. etc.

1.1 Portability

In cases where OGIP has made a commitment to provide a set of software to be released to the community, the programmers[1] involved must ensure that the software products are portable. This must be done by conforming to the generally recognized standard and isolating any system-dependent operations into a small set of subroutines; these topics are covered in §2 and §3, respectively. All portable software must conform to the ANSI Fortran 77 except for OGIP-approved extensions, and follow the rules marked with a "¶" in §5.

It is possible to relax the portability requirement if we limit the machine types to, e.g., UNIX workstations and VAXes. In this example, IBM mainframes, PCs and Crays (among others) are excluded, but this will probably not impact many OGIP users. For this reason, there are two sets of OGIP-approved extensions; one for general portability, the other for a more restricted portability. Which portability criterion should apply to a given piece of code is a policy matter; even though this document does make suggestions, those should not be taken to be the final decisions.

1.2 Fortran 90

Fortran 90 is the new international standard of the language, containing many advanced features. All the widely used extensions of the current standard are in the new standard. It is backward compatible although certain features of the language are marked "obsolescent"; these may be deleted in the next standard after Fortran 90, an added incentive for programmers not to use them (only those features widely regarded as bad are in this list).

Fortran 90 includes:

1. Array operations.

[1] This includes, but not limited to professional programmers, who are likely to have known all the things in this document anyway. This memo is aimed more towards scientists writing software part-time.

2. Pointers.
3. User-defined derived data types composed of arbitrary data structures and operations upon those structures.
4. Facilities for defining collections called 'modules', useful for global data definitions and for procedure libraries.
5. A new source form, more appropriate to use at a terminal than with punched cards.
6. New control structures such as SELECT CASE construct.
7. The ability to write internal procedures and recursive procedures, and to call procedures with optional and keyword arguments.
8. Dynamic storage allocation.
9. Improvements to the input-output facilities, including handling partial records and a standardized NAMELIST facility.

Obsolescent features are:

1. Arithmetic `IF`
2. `DO` loops with real indices
3. Assigned `GO TO` and assigned formats
4. Branching to an `END IF` statement
5. Alternate `RETURN`
6. `PAUSE` statement
7. `H` edit descriptor

There are already some Fortran 90 "compilers," which translates Fortran 90 codes to Fortran 77 or C; these are adequate for prototyping and testing purposes (in the case of NAG Fortran 90 compiler, benchmark tests suggest noticeable (25–50%) slowness compared with the same algorithm compiled with a native compiler). DEC has declared they are developing a Fortran 90 compiler, although they would not say when it might be available. A general availability of Fortran 90 compilers probably is a few years off. Nevertheless, we should keep Fortran 90 standard in mind and switch to it as soon as it becomes widely available.

In the mean time, those obsolescent features should not be used in OGIP programs.

1.3 Other considerations

On other issues, such as making the software foolproof, easy to use and easy to maintain, we will not enforce rigorous rules; such rules tend to be definite for the sake of definiteness and often annoys the independent-minded, well-disciplined programmers. Therefore all the rules in §5 that are not marked with a "¶" should be taken as advice; there is no harm in establishing your own personal style as long as you understand the reason why these rules exist.

The explanations for the rules in §5, as well as §3, are put in this document so that you can make intelligent decisions regarding your programming style.

2 ANSI STANDARD FORTRAN 77 AND OGIP EXTENSIONS

VMS Fortran has the `/STANDARD` option; Sun F77 has the `-ansi` option. Both are used to turn off all the compiler-specific extensions to ANSI Standard Fortran 77. This options, as well as utilities such as `SPAG` can be used to check how closely your code conforms to the Fortran 77 standard.

2.1 For general portability

In OGIP Fortran programs to be released to the community for use on a wide variety of machines, only the ANSI standard Fortran 77 and the extensions given below are allowed. The FITSIO package already conforms to this strict standard; this level of strict conformance may be required of general-purpose subroutine packages.

1. Both upper and lower case letters can be used. However, the programmer must assume that the compiler is *not* case sensitive, and enforce this with a compiler switch if necessary.
   - The standard states that only upper case letters are allowed. However, this is rarely enforced by compilers and leads to hard-to-read code.
2. `END DO`s are allowed.
   - This is recognized by all major compilers and is in Fortran 90; with a proper use of indentations, `END DO`s make better-structured codes.
3. `DO WHILE` loops are allowed.
   - Again, this is recognized by major compilers and is in Fortran 90.
4. `INCLUDE` statements are allowed.

- This, again, seems to be recognized by all major compilers. However, `INCLUDE`, by definition, involves file name specification. The use of `INCLUDE` should be kept to within well-defined packages, preferably within one directory.

5. Data type `INTEGER*2` is allowed when they are essential to deal with data.

2.2 For portability among UNIX and VMS workstations

The following extensions are features common to VMS Fortran, Sun Fortran, MIPS Fortran and "f2c[2]."

ASTRO-D software, for example, may require only this level of portability.

1. Variable, module and common block names up to 31 character long, including underscores, are allowed.

   - Most UNIX and VMS workstations support such extensions, although this may make programs unportable to certain machines. This also is the Fortran 90 standard.

2. `IMPLICIT NONE` is allowed.

   - `IMPLICIT NONE` is now accepted by all major compilers and is in the Fortran 90 standard. In fact, for this level of portability, the use of `IMPLICIT NONE` is highly recommended

2.3 Examples of disallowed extensions

The following commonly-used extensions to ANSI standard Fortran 77 *will not* be allowed in any *portable* OGIP programs.

1. Lines longer than 72 characters will not be allowed.

   - Many compilers do have options to extend lines, but not all; it is unwise to rely on the use of compiler switches. Longer lines can lead to cryptic error messages and unexpected results.

2. Use of <tab>s in places of <space>s will not be allowed, except as a part of a character string.

   - Most compilers do accept <tab>s, but this practice can lead to problems, particularly in combination with the 72-character rule of the Fortran standard. There are utilities that converts <tab>s to <space>s, such as SPAG.

[2]F2c is a public domain software that produces ANSI standard C programs from input Fortran 77 units that may contain certain extensions

3. Avoid structures as much as possible; stick to standard Fortran data types, if at all possible.

    - Existing programs for which the banning of structures would cause a major headache (e.g., xspec) are exempt from this rule. Also, some low-level routines may have to utilize non-standard data types.

4. Avoid string concatnation that requires dynamic allocation of memory.

    - `strng3=strng1//string2`, when one of the two strings on the right hand side is a dummy argument declared as `CHARACTER*(*)`, is legal. However, it is illegal to use `string1//string2` as argument in a further subroutine call or in a print statement is illegal. The MIPS Fortran compiler does not support this extension (although many others do).

5. Do not mix character strings and numeric variables in a common block.

    - This usually works, but not in the standard and could cause word-length related problems.

6. Do not use free-format read in internal I/O

    - This reasonable-sounding extension is not supported by some compilers, while the Sun compiler does not process the error conditions properly for free-format internal read.

7. Do not use inline comments.

    - Comments preceded by "`!`" are useful, particularly in explaining what variables are — but this feature is not supported by many compilers (this is a Fortran 90 feature).

3 VAX/VMS/DCL vs. Sun/UNIX/c-shell

In addition to the difference among Fortran compilers, features of the operating systems and command languages play a major role in determining which program is portable and which is not. This section summarizes the major differences between VAX/VMS/DCL and Sun/UNIX/c-shell that Fortran programmers should be aware of.

3.1 File systems — general

- UNIX file names are case-sensitive; VMS ones are not.
- VMS has the concept of file version numbers; UNIX does not.
- UNIX has few rules about file names; under VMS, file names are of the form "<name>.<ext>;<n>" where <ext> is the extension name that indicates the file type and <n> is the file version number. UNIX programmers often use file names with multiple fullstops "." in them,

which is forbidden under VMS. The semi-colon ";", used under VMS to separate extension and version number, is used under UNIX to separate multiple commands in a single input line; this, and other special characters, can be a part of a file name under UNIX, but this makes specifying the file a laborious process (e.g., use a backslash "\").

- Device and directory specification syntax is rather different; UNIX uses a tree structure separated by slashes ("/") while VMS uses the form "<device_name>:[<dir>.<subdir>]<filename>."

3.2 File systems — Fortran IO

- Under VMS, `OPEN` with "`STATUS='NEW'`" will create a new version if file with the given name already exists; it leads to a crash under UNIX.
- Under VMS, `OPEN` with "`STATUS='OLD'`" will result in an error if the named file does not exist; under UNIX, an empty file will be created by this command, although any subsequent "`READ`" command is likely to result in an error.
- VMS has a vigorous system of file attributes; you cannot use direct access on sequential access files or vice versa. There is no such restrictions on UNIX systems.
- Fortran unformatted files are implemented in a certain way under UNIX, by adding 4-byte integers before and after a record representing the number of bytes in it. Thus, even though you can mix direct and sequential access under UNIX, that could lead to errors (or very UNIX dependent code).
- Under VMS, the easiest (the only?) way to allow simultaneous access to a file by multiple processes is to open it with a "`READONLY`" keyword (otherwise, the first process that opens the file will lock it). This is a non-standard keyword. Under UNIX, this is not the case but the onus is on the programmer to avoid nasty results due to one process reading a file while it is being modified by another process.
- Fortran standard does not specify whether the pointer should be when an existing file is opened. In many systems including VAXes and Sun workstations, it is at the beginning of the file. The non-standard keyword `ACCESS='APPEND'` works on both these systems to put the pointer at the end of the file, but there is no standard-conforming way to achieve this. The "`REWIND`" statement is in the standard and will move the pointer to the beginning of file[3].
- Fortran carriage control is not implemented on UNIX systems, although a subset of this feature can be used by opening the file (or even the terminal) with a "`FORM='PRINT'`" qualifier. On VMS, Fortran carriage control is the default, which can be overridden with a "`CARRIAGECONTROL='LIST'`" qualifier. Both these qualifiers are non-standard.
- Fortran standard specifies that you can open 64 files from a program at any given time; UNIX includes the standard input, output and error in this calculation thus 61 other files can be opened by default. On the Sun workstations, it is possible to extend this to 256

[3]In Fortran 90, there is a keyword "`POSITION`" which can take values "`APPEND`," "`REWIND`" or "`ASIS`."

total files, however. On VMS, there is a separate limit imposed by the process quota; this can be changed only with a SYSTEM privilege.

- Under VMS, file names can be specified using logical names; no comparable method exist under UNIX/c-shell, although environment variables can be expanded using a system call.
- "Using tape files on a SunOS or UNIX systems is awkward because, historically, UNIX development was oriented toward small data sets residing on fast disks. Magnetic tape was used by early UNIX systems for archival storage storage and moving data between different machines. Unfortunately, many FORTRAN programs are intended to use large data sets from magnetic tape." — from Sun Fortran Users's Manual. Sun provides a Fortran tape I/O package, which, by their own admission, offers only a partial solution and the manual actually discourages its use. It would also be non-standard.

3.3 Data Representations

- Byte order: On VAXes and 80x86 chips, the least significant byte comes first. On SPARC, 680x0 and IBM mainframes, the most significant byte comes first. MIPS chips can be configured either way.
- IEEE real numbers: Sun implements the IEEE arithmetic in such a way that programs do not crash when certain severe error conditions are encounters, e.g. dividing by 0. This is because IEEE real numbers include Inf (infinity), NaN (not a number) and other oddities. This seems to be a violation of the Fortran standard (the results must be mathematically defined), and contrary to what many Fortran programmers wish (division by 0 is usually a sign of coding error). You can trap these error conditions, but that makes the code Sun-specific.

3.4 Other issues

- Under UNIX, a backslash ("\") has a special meaning, which even applies to character strings within a Fortran program. Use of backslashes should be avoided, if at all possible.
- Under VMS, each process has a set of "quotas," limits on various system resources that the process can use; try "SHOW PROCESS/QUOTAS" for details. This can cause incompatibility problems amongst VMS machines that are configured differently. Under UNIX, these limits are typically hard-wired are non-existent; excessive use of system resources under UNIX typically hangs up the machine rather than stopping the program with an error condition.
- Under VMS, there is a limit on the size of a line that can be written out free-format. This is 128 characters by default.

4 PROLOG COMMENTS

The following is an example of prolog comments in a style recommended by Allyn Tennant and Ian George. The comments are sandwitched between *++ and *-, which allows automatic extraction of prolog comments.

```
*+SX_FITLIN
SUBROUTINE SX_FITLIN(X,Y,SIGMAY,NPTS,MODE,A,SIGMAA,B,SIGMAB,R)
INTEGER NPTS, MODE
REAL X(NPTS), Y(NPTS), SIGMAY(NPTS)
REAL A, SIGMAA, B, SIGMAB, R
C
C Description:
C  Fits straight line Y = A + B*X by least squares.
C
C Arguments:
C  NPTS         (i) : No. of data pts in X,Y,SIGMAY, must be >2.
C  X(NPTS)      (i) : Array of independent variables
C  Y(NPTS)      (i) : Array of dependent variables
C  SIGMAY(NPTS) (i) : Array of Std Errors on Y (only used if MODE=1)
C  MODE         (i) : Data weighting mode.....
C        -1  for WEIGHT = 1/ABS(Y)    i.e. Poission errors
C 0  for WEIGHT = 0.0     i.e. all points same weight.
C 1  for WEIGHT = 1/SIGMAY**2 i.e. user-supplied errors.
C  A         (io): Intercept on X=0 line
C  SIGMAA        (o): Std error in A assuming Gaussian error distribution.
C  B          (o): Slope of line
C  SIGMAB  (o): Std error in B assuming Gaussian error distribution.
C  R          (o): Correlation coeff between X and Y
C
C Origin:
C  Algorithm LINFIT by  Bevington, but adapted to ... blah,
C .... blah blah
C Authors/Modification History:
C  Ian M George    (1989 April 28), original version
C  Allyn F Tennant (1992 Aug 01), minor bugs corrected
C  Nick White      (1992 Sept 03), MODE=-1 added to handle ... blah blah
C blah blah blah
C  Allyn Tennant   (1992 Sep 14), improved comments.
*- Version 2.3.2
```

5 THE RULES

Here are the set of rules, with brief explanations. Those with "¶" will be strictly enforced for software to be released to the community.

5.1 Language

Do Not Use Unapproved Extension to Fortran 77 ¶ 1

Use the ANSI standard plus OGIP extensions (§2) only, if you are mandated to provide a portable software tool to the community.

Do Not Use Obsolescent Features ¶ 2

Although these features will remain in Fortran 90, it is wise to avoid them. They are marked obsolescent for good reasons.

5.2 Control structures

Use indenting to show structure ¶ 3

No explanation necessary.

Do not overuse `STOP`, `RETURN` and `ENTRY` 4

The idea behind this recommendation is to make it easy to see where the execution of programs and subroutine starts and ends. The first two are usually unnecessary since `END` statements have the desired effect (NB. HP Fortran requires `RETURN`, which is a violation of the standard).

Avoid `GO TO` 5

It is often possible to use `DO` loops and block `IF`s instead of `GO TO`s; this usually helps in making the program structure clear. However, in standard Fortran 77, a total avoidance of `GO TO`s can actually result in a poorly structured code so this rule should not be taken to extremes.

Use the computed `GO TO` sparingly 6

Computed `GO TO` can be a reasonable substitute for a "case" construct, where value of a single variable (e.g., command number in a command-driven program) determines what is to be executed next. However, there is an alternative (by using `IF` blocks) within the current language; Fortran 90 will provide an explicit `SELECT CASE` construct with all the features you would want in such a construct.

5.3 User interface issues

User input must not be case-sensitive ¶ 7

Only exception to this rule is the file names on case-sensitive systems.

Validate inputs ¶ 8

Users must be allowed to make mistakes without crashing the program. Reading with `err=<label>` is always a good idea; in addition, case-by-case validation (e.g., if the program requires a positive number, check for it) should be performed before using the input data.

Do not overprint with "+" 9

This does not always work; if it is desirable to overprint, wrap it up in a device-dependent subroutine.

Be device-independent 10

Avoid device (e.g., terminal) dependent code (e.g., VT100 escape sequences) as much as possible, or isolate it in a subroutine.

5.4 Variables

Declare every variable ¶ 11

Do not use `DIMENSION` for arrays.

Define sizes parametrically 12

Use the parameter statements so that you only have to change it to change the array size.

Use CHARACTER∗(∗) 13

Dummy argument should in general avoid a hard-wired length for generality.

Use standard data types 14

Fortran standard recognizes `INTEGER`, `REAL`, `DOUBLE PRECISION`, `CHARACTER` and `LOGICAL`. Type declaration of the form `REAL*n` is often recognized but non-standard, and should be avoided as much as possible, although there are situation where `INTEGER*2` is essential. `BYTE` is also highly non-standard.

Don't access uninitialized variables 15

Usually uninitialized numerical variables are 0, but this is not guaranteed.

Save everything you have to 16

Local variables within a subroutine usually retain the values at the end of the previous call. This practice is not guaranteed, however, unless there is a `SAVE` statement.

Logical and Integer are different 17

Thus, do not treat an integer as logical or vice versa.

Don't use EQUIVALENCE 18

Using `EQUIVALENCE` to "convert" among different file types *is* a useful trick; however, note this often makes the code machine-dependent. Using `EQUIVALENCE` on anything in a common block, or using `EQUIVALENCE` to extend an array, are both extremely dangerous.

5.5 Subroutines/functions

Use distinctive names 19

We actually recommend to break the Fortran 77 restriction of 6-characters for subroutines and functions, for a package of subroutines to be used widely for different application programs. This is to minimize the danger of clashing subroutine names among different packages (e.g., a new version of PGPLOT may have a new subroutine with the name you have used for your

subroutine). The British STARLINK recommends the following standard form: < *pcg* >_<s/f-name>, where <pcg> is a 3-character package name and <s/f-name> is a 5-character subroutine (function) name. This is an excellent suggestion and we propose to adopt this as an OGIP standard.

Data type must not change across a CALL 20

Changing data type across a call is very machine-dependent. It should be done only with extreme care.

FUNCTIONs must not have side effects 21

Make it into a subroutine, if there are more than one output variables.

Subroutines should report errors using a flag 22

This will result in a code more flexible than if the subroutine printed out error messages or stops the program.

NOTE: Should we have an OGIP standard for error handling? Sounds like a good idea, but how?

5.6 Common Blocks

Use distinct names 23

See the rule on subroutine/function names above.

Minimize use of COMMON 24

Overuse can result in a code that is hard to read. A subroutine package should collect all its internal common blocks in a include file. Passing arguments from the host program to a general-purpose subroutine library using a common block is discouraged.

Follow the standard order 25

Each common blocks should contain doubles, single precision reals and integers (and if non-portable, integer*2s, bytes/characters) in that order. On many machines, this minimizes the operations when extracting individual variables out of the common block.

5.7 Readability

Use statement labels only on Format and Continue 26

Actual number crunching should be separated from decisions and loops; also, different loops and decisions should have their own end-points. By the same token, avoid nesting multiple `DO` loops with the same label.

Clean up garbage 27

Unused statement labels, declared but unused variables and unused format statements all make the code less readable.

5.8 File IO

Use `XANLIB` 28

`XANLIB` contains a set of subroutines that perform system dependent tasks. OGIP programmers should use these whenever possible to avoid re-inventing the wheel.

Don't use literal I/O unit numbers 29

Hardwired unit numbers make it hard to avoid conflict.

5.9 Numerical accuracy

Be aware of floating-point limitations 30

There are many ways in which floating number arithmetic can trip up a programmer.

- On any machine, there is a floating number `X` and `Y` such that (mathematically computed) `X+Y`, `X-Y`, `X*Y` and/or `X/Y` are not a valid floating number.
- Adding 0.001 1000 times does not make 1.0. This is mostly due to inaccuracy inherent in adding a number to a much larger one, and in much smaller part due to conversion from decimal to binary.
- `A+B+C` and `A+C+B` are not necessarily the same.
- Compilers have the choice of executing `A/(B*C)` when the user specified `A/B/C` to speed it up. The results may be different from `(A/B)/C`.
- In all the above and much more, running the same code on different machines will probably result in a slight difference.

Thus, do not compare real numbers for equality; use brackets to specify the best order of execution; use multiplication rather than recursive addition, if that is possible.

Do not use mixed mode arithmetic 31

Converting from a real to an integer, for example, should be done explicitly; these are major events in terms of computing speed, and should be treated as such.

5.10 Comments

Prolog comments¶ 32

Explain what the module does, what input and output variables are, list author(s), dates and history, as specified in §4.

Begin modules properly 33

Comments preceding the first statement of a module (`PROGRAM, SUBROUTINE, FUNCTION` or `BLOCK DATA`), or those following the `END` statements, are discouraged to minimize confusion.

Explain variables 34

Particularly if the variable names are cryptic, which often is the case when the programmer is strictly obeying the 6-letter variable name rule.

Make comments stand out from code 35

Use blank lines and spaces, and use other visual aids.

Write meaningful comments 36

This is a warning against trivial comments (`I=I+1`; "increment counter"). Explain why you are doing this, rather than what you are doing.

5.11 Others

Avoid clever but obscure code 37

At least comment why it's clever, preferably with a reference.

Use VMS compatible file names ¶ 38

In cases where program has to know the file name (scratch file, help file), the file name should be a legitimate one on VMS, even if the program is written on a UNIX machine.

Do not overuse `INCLUDE` 39

`INCLUDE` statements, by their very nature, cannot avoid involvement with the operating system dependent nature of file names. Using `INCLUDE` on files in the same directory as the code is relatively harmless; `INCLUDE` across directory boundaries are necessarily system-dependent and should be minimized.

Do not rely on compiler switches 40

Compiler switches usually enable you to extend Fortran standard; it is hard to enforce the use of the necessary switches.

Acknowledgements

I thank several OGIP members and Allyn Tennant for their comments on an earlier version of this document. I also relied heavily on U.K. Starlink document by P. T. Wallace.

References

"Starlink Application Programming Standard" by Wallace, P. T., 1992, Starlink General Paper SGP 16.10.
"Fortran 90 Explained" by Metcalf, M. & Reid, J. 1990, Oxford University Press.
"The XANADU Programmer's Guide" by K. Arnaud *et al.*, 1992.