ALLIED ORDNANCE PUBLICATION

AOP-61 (Edition 1)

NATO INTERNATIONAL STAFF – DEFENCE INVESTMENT DIVISION

NATO TECHNICAL SHAREABLE SOFTWARE PRODUCT DEVELOPMENT STANDARDS AND GUIDELINES

AOP-61

OCTOBER 2011

AOP-61
(Edition 1)

NORTH ATLANTIC TREATY ORGANIZATION

NATO STANDARDIZATION AGENCY (NSA)

NATO LETTER OF PROMULGATION

24 October 2011

1. AOP-61, (Edition 1) - NATO TECHNICAL SHAREABLE SOFTWARE is a NATO/PFP UNCLASSIFIED publication. The agreement of nations to use this publication is recorded in STANAG 4683.

2. AOP-61, (Edition 1), is effective upon receipt.

3. AOP-61, (Edition 1), contains only practical information. Any corrections that nations may make to it are not subject to ratification. The Custodian Group is AC/225(LCG3-SG/2) and any correction or addition proposed for the AOP should be addressed to members of that Group. Such amendments, if accepted by the Custodian Group in plenary session, will be incorporated into the next version of AOP-61.

Cihangir AKSIT, TUR Civ
Director, NATO Standardization Agency

RESERVATIONS

NATION	SPECIFIC RESERVATIONS

RECORD OF CHANGES

Change date	Date entered	Effective date	By whom entered

Table of Contents

1. Introduction

1.1 Identification

This document describes the NATO Technical Shareable Software (NTSS) Product Standards (Part 1) and FORTRAN Coding and Style Guidelines (Part 2) for NAAG AC/225 Land Capability Group 3 (LCG 3) Sub-Group 2 (SG/2).

1.2 Scope

This AOP applies to all software that is part of the NATO Technical Shareable Software suite developed under STANAG 4683 (“NATO Technical Shareable Software”) unless an exception has been explicitly authorized by SG/2.

1.3 Rationale

The NTSS is a collection of software products designed for applications dealing with technical ballistics issues. These products will typically be used in a laboratory or desktop office environment. NTSS products will generally be developed by the member nations of SG/2 and shared according to STANAG 4683 (NTSS). To enable SG/2 to maintain the NTSS products and ensure the each product is usable and maintainable, a minimal set of standards and guidelines are required.

1.4 Reference Documents

(a) NATO STANAG 4683 (Edition 1) on a NATO Technical Shareable Software dated 29 January 2008

Part 1 – Product Development Standards

1. Product Standards

1.1 Supporting STANAGS

Each NTSS software product will have one or more related STANAGs and shall implement all or part of the STANAG.

1.2 Product Release

1.2.1 Contents

NTSS products shall contain at least the following artifacts:

- User Guide
- Source code
- Installation instructions
- Sample inputs and outputs
- Release Notes

1.2.2 Schedule

The product release schedule is determined by the nation that maintains the product.

1.2.3 Media

Products shall be distributed on CD or DVD

1.3 Classification Markings

All artifacts in a product release shall be marked with the appropriate level of classification: **UNCLASSIFIED** [**U], NATO/PP UNCLASSIFIED [NPU]** or **NATO RESTRICTED [NR**].

2. Product Development Standards

2.1 Requirements Management

- New Change Requests shall be submitted to SG/2 for review and approval
- Problem Reports shall be communicated to the national SG/2 representative of the nation that is maintaining the software
- The nation that is maintaining the product shall maintain a list of requirements and problem reports

2.2 Software Development Standards

- Coding and style guidelines shall be applied to the software when possible. See Part 2 for the FORTRAN Coding and Style Guidelines. A NTSS product developed using a different language should amend this AOP with coding and style guidelines for the language.
- The nation developing or maintaining the product shall conduct engineering level testing

2.3 Configuration Management

The nation maintaining the product shall keep the product under configuration control and maintain a change history for the product

Part 2 – FORTRAN CODING AND STYLE GUIDELINES

1. Purpose

These guidelines apply to FORTRAN77 and FORTRAN90 code maintained and developed as part AOP-61. It is not intended to address all issues in FORTRAN coding. In the event of that applicable coding standard guidance cannot be found in this document, refer to the ANSI and ISO standards listed in Section 1.1. In addition, it is not intended as a FORTRAN90 tutorial. It is assumed the reader is familiar with the language elements and capabilities.

1.1 Reference Documents

The following documents should be consulted when deciding which intrinsic functions and procedures are to be used in the code:

- ANSI X3.198-1992 (F90)
- FORTRAN International Standard, ISO/IEC 1539-1:1991 (F90)

Compiler supplied language extensions which are not specified in the above references shall **not** be permitted.

In addition, a partial list of FORTRAN90 programming reference manuals is given in Annex D.

1.2 Document Structure

The structure of this document is as follows:

2. Application

This document is intended to be strictly applied to new programs and associated subroutines, functions and modules. Existing code will not necessarily be reworked to meet all of the aspects of this coding guideline. In order to help decide how and when guidelines should be retrofitted to existing code, each guide item is annotated with one of the following terms:

Immediate	Do immediately.
Opportunity	Do when the compilation unit is edited the next time. Such changes are relatively small and should not affect, or be affected by existing code in a significant way.
Rework	Do when the unit is being reworked. Changes have impact on the code semantics and should be done as part of a rewrite of such a unit (usually when more than 40% of the unit will be changed).
New Code	This item applies only to new code. There is no applicability to existing code, possibly because existing code has no occurrences of the construct/issue in question or it is thought that it is not reasonable to retrofit the guide item.

2.1 Waiver Approvals

Every time a rule in this standard is broken, approval for the deviation must be documented. Depending upon the nature of the deviation, differing approvals are required. The approval level needed for a deviation from a rule is documented in the "Waiver" column on each table. The following Waiver authorizations are used.

Peer Review	The Peer Reviewer accepts the deviation from the standard and verifies that the documentation in the code explains the rationale for the deviation. No further levels of approval are necessary.
Authority	The Project Authority accepts the documented deviation, with the peer reviewer's concurrence, from the standard.

2.2 Background

[1]FORTRAN was invented by a team of programmers working for IBM in the early nineteen-fifties. This group, led by John Backus, produced the first compiler, for an IBM 704 computer, in 1957. They used the name FORTRAN because one of their principal aims was ``formula translation''. But FORTRAN was in fact one of the very first high-level language: it came complete with control structures and facilities for input/output. FORTRAN became popular quite rapidly and compilers were soon produced for other IBM machines. Before long other manufacturers were forced to design FORTRAN compilers for their own hardware. By 1963 all the major manufacturers had joined in and there were dozens of different FORTRAN compilers in existence, many of them rather more powerful than the original.

All this resulted in a chaos of incompatible dialects. Some order was restored in 1966 when an American national standard was defined for FORTRAN. This was the first time that a standard had ever been produced for a computer programming language. Although it was very valuable, it hardly checked the growth of the language. Quite deliberately the FORTRAN66 standard only specified a set of language features

[1] from Professional Programmer's Guide to FORTRAN77; Clive G. Page, University of Leicester, UK; 7th June 2005; Copyright © 1988 - 2005 Clive G. Page

which had to be present: it did not prevent other features being added. As time went on these extensions proliferated and the need for a further standardization exercise became apparent. This eventually resulted in the current version of the language: FORTRAN77.

FORTRAN77 was produced in 1977 by a committee of the American National Standards Institute (ANSI) and was subsequently adopted by the International Standards Organization (ISO). The definition was published as ANSI X3.9-1978 and ISO 1539-1980.

When the latest FORTRAN Standard was issued in 1977 there was fairly widespread disappointment that it did not go just a little further in eliminating some of the tiresome restrictions that had persisted since the early days. The US Department of Defense issued a short list of extensions which manufacturers were encouraged to add to their FORTRAN77 systems. The most important of these were the following:

- the END DO statement
- the DO WHILE loop
- the INCLUDE statement
- the IMPLICIT NONE facility
- intrinsic functions for bit-wise operations on integers.

One of the most irksome restrictions of FORTRAN77 is that symbolic names cannot be more than six characters long. This forces programmers to devise all manner of contractions, abbreviations, and acronyms in place of meaningful symbolic names.

[2]FORTRAN 90 (F90) is a recent update of the FORTRAN programming language. FORTRAN 90 introduces several new features over its predecessor, FORTRAN77 (F77), including allocatable arrays, modules, and new statements. For more information about FORTRAN 90 can see the following tutorials:

- FORTRAN90 for the FORTRAN77 Programmer
- FORTRAN90 Tutorial at NASA/Ames

F90 is totally backwards-compatible with F77. Older code (a.k.a. "legacy code") which adheres to the F77 standard should compile and run under F90 without problems.

In recent years there have also been further updates to FORTRAN, namely FORTRAN 95 (F95), and FORTRAN 2000 (F2K). These new FORTRAN versions include the new language features which are present in F90, but generally do not

[2] from the Harvard University Atmospheric Chemistry Modeling Group web site http://www-as.harvard.edu/chemistry/trop/geos/documentation/geos_chem_style.html, GEOS–CHEM Style Guide, dated May 27, 2004

include the old, obsolete features found in F77. Therefore, pure F77 code may not always compile in F95 and F2K.

3. Rationale for this Coding Standard

The goal of this standard is to provide the framework in which a multi-site development of code occurs. Readability, maintainability, correctness, testability and the use of common programming paradigms are the overriding goals of this standard. This is essential in providing consistent, maintainable code within an organization. The goals of readability and understandability take an even more stringent form when coupled with the fact that many of the developers and maintainers may not be native English speakers; and acronyms, short forms, and idioms are very challenging.

4. Rules and Conventions

4.1 General (Rationale)

ID	Rule	When Retrofitted	Waiver
4.1-1	Every time a rule is broken, the rationale for the deviation shall be documented.	N/A	Peer Review
4.1-2	The length of a line of source code shall not exceed 96 characters, including trailing comments.	Rework	Authority
4.1-3	All new programs shall conform to the FORTRAN 90 standards.	New Code	Authority
4.1-4	All new programs shall be written in free format with a minimum number of labels. Exceptions are labels for DO loops and IF constructs.	New Code	Authority
4.1-5	New subprograms written for legacy code shall be written in the same format as the legacy code (fixed or free). If legacy code is both fixed and free format, the new code shall be in free format.	New Code	Authority

4.2 Documentation (*Rationale*)

4.2.1 Program Headers (Rationale)

4.2.1-1	Every FORTRAN source code main program shall have a **completed** program header block of the format defined in Annex B.1 (a blank template is provided in Annex C.1).	Opportunity	Authority
4.2.1-2	The program header shall contain the name of the program, which shall **NOT** be MAIN.	Opportunity	Authority

4.2.2 Subroutine/Function/Module Headers (Rationale)

4.2.2-1	Every FORTRAN source code subroutine, function and module shall have a **completed** program header block of the format defined in Annex B.2 (a blank template is provided in Annex C.2).	Opportunity	Authority

4.2.3 Comments (Rationale)

4.2.3-1	The comment character "**!**" shall be used for comment lines and trailing comments (required for free format). One space shall be inserted between the comment character and the comment text.	Rework	Peer Review
4.2.3-2	Comments should provide useful information that clarifies or enhances the code. If a section of source code requires continual line by-line comments, and cannot be followed without these comments, the code itself should be re-evaluated to determine if its base level of clarity is sufficient.	Rework	Peer Review
4.2.3-3	Comments shall have beginning and ending demarcations consisting of: !---------- for minor blocks of code and: !********** for major blocks of code.	Rework	Peer Review
4.2.3-4	All comments shall be indented at the same level as the code to which they apply. Starting at one column less than the level separates the comment from the code and preserves the structure.	Rework	Peer Review
4.2.3-5	Use trailing comments when they provide additional clarity. Possible uses of trailing comments are to show the units of variables, or to comment the call to a subprogram. Attempt to align trailing comments at the end of lines where lines are close together.	Rework	Peer Review
4.2.3-6	Comments that refer to compilation bug fixes and/or workarounds shall include the vendor name and the release number(s) of the compiler.	Opportunity	Peer Review

4.3 Format Issues

4.3.1 Naming Conventions (Rationale)

4.3.1-1	All types, variables, subprograms, etc., shall be named using meaningful identifiers, such as full names/words, with a description that relates to the item so that a reader will recognize its use. Multi-word identifiers shall use an underscore to separate words (e.g. Wpn_System). Use approved abbreviations and acronyms.	Rework	Peer Review
4.3.1-2	FORTRAN keywords and intrinsic function names shall **NOT** be used as identifiers.	Immediate	Authority

4.3.2 Capitalization (Rationale)

4.3.2-1	All reserved words (e.g. in, out, call, read, write) shall appear in lower case.	Opportunity	Peer Review
4.3.2-2	All pre-defined FORTRAN identifiers (e.g. Integer*2, Real*8, Character) shall appear in mixed case.	Opportunity	Peer Review

4.3.2-3	All other names shall have the initial and first letter after each underscore capitalized, as in:
	`Character Fuze_Model`
	`Real*8 Fuze_Setting`
		Opportunity	Peer Review

4.3.2-4	Acronyms shall appear in upper case (e.g., ASCII_Filename).	Opportunity	Peer Review

4.3.3 Blank Lines, White Space and Alignment (Rationale)

4.3.3-1	Use blank lines, white spaces and code alignment to enhance the readability of the code	Opportunity	Peer Review
4.3.3-2	Three blank spaces shall be used as the basic unit of indentation for continuation lines.	Rework	Peer Review
4.3.3-3	Indent inside each control structure (e.g., **do**, **if-then-else**, **case**). An indentation level shall consist of three blank spaces.	Rework	Peer Review
4.3.3-4	Blank spaces shall always be used instead of tabs in source files.	Rework	Peer Review
4.3.3-5	Use spacing around operators (=, +, -, *, and /).	Rework	Peer Review
4.3.3-6	Use spacing between subroutine/function names and the opening parenthesis. This includes FORTRAN intrinsic subprograms.	Rework	Peer Review
4.3.3-7	DO NOT use spacing between array names and the opening parenthesis.	Rework	Peer Review

4.3.4 Order of Declarations (Rationale)

4.3.4-1	The order of information conforms to FORTRAN requirements : • Compiler options • Program/subroutine/function/module name • Header • "Use" statements in alphabetical order • "Include" statements in alphabetical order • Passed variable types • Local parameters in alphabetical order • Local types in alphabetical order • Variables grouped by type and in alphabetical order	Opportunity	Peer Review

4.4 Statements (*Rationale*)

4.4-1	Avoid the use of **go to** statements whenever possible. The **go to** statement shall only be used to transfer to: an error processing block; a program end; a subprogram end; or to make a conditional exit from a **do** loop or **if** block.	Rework	Peer Review
4.4-2	Use the **do**-**end do** form of the **do** loop. Use a label if the loop extends over a large number of lines.	Rework	Peer Review
4.4-3	Use "**do**" loops over a static range whenever possible. Use "**do**" loops over dynamic ranges as a second choice and, as a 3rd choice, use **do while** loops.	Rework	Peer Review
4.4-4	Terminate a **do** loop with a **continue** statement if it is not the **do**-**end do** format. Do not terminate a **do** loop with an executable statement	Opportunity	Peer Review
4.4-5	Do not use language features which are obsolete, e.g., ENCODE, DECODE, Hollerith (in any form), arithmetic IF, etc.	Rework	Authority
4.9-6	Use **&** as the continuation character in column 6 (for fixed format). This is the "official" F90 continuation character. It may also be used at the end of the code line in free format.	Rework	Peer Review
4.4-7	All end statements for programs, subroutines, functions and derived types shall include the routine type (program, subroutine, function)and/or defining name.	Opportunity	Peer Review

4.4-8	Minimize the number of return statements in a subprogram.	Rework	Peer Review
4.4-9	DO NOT use **assign** and assigned **go to** statements.	Rework	Peer Review
4.4-10	Use the **select case** construct to select from a list of choices, instead of the **if-then-else** construct**.**	Rework	Peer Review
4.4-11	Numeric labels shall be monotonically increasing	Rework	Authority
4.4-12	Multiple statements shall **NOT** be used on a line of code.	Rework	Authority
4.4-13	Composite keywords (**end do, end if...**) shall contain the blank character separator.	Rework	Authority

4.5 Types (*Rationale*)

4.5-1	The compiler directive **Implicit NONE** shall be used.	New Code	Authority
4.5-2	Use the initialization format for all variable type and constant definitions, e.g., `Logical :: End_Header = .FALSE.` and `Integer*2, parameter :: Aero_Cnt = 11, Array_Max = 20`	New Code	Peer Review
4.5-3	Multi-indexed arrays should be dimensioned the same way the compiler will store the values, i.e., the inner (leftmost) index varies the fastest. An example: if storing 100 locations as latitude, longitude, altitude; dimension Position(3, 100).	New Code	Peer Review
4.5-4	Use the D exponent when assigning a constant value to a REAL*8 number.	Opportunity	Peer Review
4.5-5	The underscore form for exponents shall not be used when assigning a constant value, e.g., -1.83_4. Use -1.83D4; -1.83D04; -1.83E4; -1.83E04	Opportunity	Authority

4.6 Complexity (*Rationale*)

4.6-1	Parentheses shall be used to clarify complex precedence and to enhance readability.	Rework	Peer Review

4.6-2	Constructs and names that rely on the use of negatives should not be used.
	Use:
	`if (Operator_Missing) then`
	rather than:
	`if (.not.Operator_Found) then`
		Rework	Peer Review

4.6-3	All subprogram passed parameters should have their intent (**in**, **inout**, **out**) declared.	Rework	Peer Review

4.6-4	All output parameters should be initialized as soon as possible and in any case always before an explicit return could be encountered.	Opportunity	Peer Review
4.6-5	Passed parameters shall not exceed 10 for any subprogram.	New Code	Peer Review
4.6-6	Input data required for or data output from a subprogram or module shall preferably be passed through the calling sequence instead of common blocks.	New Code	Peer Review

5. Rationale for Rules and Conventions

5.1 General

5.1-1 (4.1-1) This rule provides the flexibility to deviate from the coding standard when necessary but requires that the rationale for the deviation be documented. This deviation is then highly visible for peer review and long term maintenance.

The general philosophy is that the code itself must be self-documenting. Comments should augment code where necessary but should be created in the knowledge that comments and header comments are often not maintained to the same standard as the code and often get out-of-date, creating confusion for readers later in the maintenance cycle. Comments therefore should explain the intent of the code and leave it to well-written code to provide the details.

5.1-2 (4.1-2) This rule provides for increased readability of source code in both online and printed formats and reflects the minimum level of support which is easily available to users. In addition, there are human limitations in the width of the field of view for understanding at the level required for reading source code. Although other documents recommend 70-80 columns, the increased screen capability and tradeoff with representative naming suggests a larger width. The choice of 96 was decided as a reasonable value that could be displayed on all modern displays.

5.1-3 (4.1-3) This rule provides for conformity to a stricter standard than the FORTRAN 77 standard while providing modern programming practices. It also provides backward compatibility to FORTRAN 77 for legacy code.

5.1-4 (4.1-4) This rule provides more coding space to accommodate meaningful names and indentation practices.

5.1-5 (4.1-5) This rule provides compatibility with legacy code but allows the piecemeal upgrade to the current standard.

5.2 Documentation

5.2.1 Program Headers

5.2.1-1 (4.2.1-1) - (4.2.1-2) Program headers are used to quickly gather pertinent information about that program. The quick summary of the purpose, compilation modules, I/O interfaces and change history are important for giving the succinct overview that is often needed.

5.2.2 Subprogram Headers

5.2.2-1 (4.2.2-1) The subprogram headers are used to quickly gather pertinent information about that subprogram. The quick summary of the purpose, passed parameters, external modules used, processing and change history are important for giving the succinct overview that is often needed.

A.2.3 Comments

5.2.3-1 (4.2.3-1) - (4.2.3-2), (4.2.3-5) Comments should be reserved for expressing needed information which cannot be expressed in code and highlighting cases where there are overriding reasons to violate one of the coding standard rules. The structure and function of well-written code is clear without comments. Obscured or badly structured code is hard to understand, maintain, or reuse regardless of comments. *Bad code should be improved, not explained.*

An additional factor is that code and documentation should be oriented towards maintainers who potentially may be less comfortable with the domain than the original author. In addition, maintainers and reviewers may not be completely comfortable working in English and may not be familiar with acronyms, idioms and contractions. Needed information must be easy to understand.

5.2.3-2 (4.2.3-3) - (4.2.3-4) The formatting rules for comments are designed to make comments visually distinct from the code. Standardizing this formatting provides increased readability.

5.2.3-3 (4.2.3-6) This rule documents information that often proves to be extremely useful in future maintenance releases of the software.

5.3 Format Issues

5.3.1 Naming Conventions

5.3.1-1 (4.3.1-1) Standardization of naming with clear identifiable names is critical to readability and maintainability.

5.3.2 Capitalization

5.3.2-1 (4.3.2-1) - (4.3.2-4) Standardized capitalization increases readability.

5.3.3 Blank Lines, White Space and Alignment

5.3.3-1 (4.3.3-1), (4.3.3-3) and (4.3.3-5) - (4.3.3-7) Blank lines help to group logically related lines of text (AQ&S Section 2.1.6). White space between operators and punctuation improves readability. Alignment of operators, declarations, parameter modes, and parenthesis facilitates readability and understandability.

5.3.3-2 (4.3.3-2) Consistent spacing improves the readability of the code because it gives a visual indicator of the program structure.

5.3.3-3 (4.3.3-4) Indenting with spaces is more portable than indenting with tabs because tab characters are displayed differently by different terminals, editors, and printers.

5.3.4 Order of Declarations

5.3.4-1 (4.3.4-1) These rules provides for a consistent ordering of items within a program or subprogram. This makes declarative items easier to find, especially for those not familiar with a given compilation unit.

5.4 Statements

5.4-1 (4.4-1) The **go to** is an unstructured change in the control flow. Worse, the label does not require an indicator of where the corresponding **go to** destination(s) are (hence the requirement (4.4-14) that all labels be monotonically increasing). The exceptions are jumps to error blocks or the end of the program or subprogram, and exit from a **do** loop or **if** construct where the next logical statement is not appropriate. An example is a do **loop** search in which a drop through (the item was not found) constitutes an error requiring additional processing.

5.4-2 (4.4.2) Use of the **do**-**end do** form of the **do** loop reduces the necessity of line labels. Loop names for **do** loops help readers of the source code find the associated end for that loop.

5.4-3 (4.4-3) Do while loops are more difficult to analyze, to show termination, or to optimize. The invariance of the **do** loop variable inside the loop prevents a number of possible programming errors. Expressing the loop as a static range is preferable, expressing it as a dynamic range in a **do** loop is next, followed by do while loops with evaluations of simple expressions and while loops with complex expressions.

5.4-4 (4.4-4) This format produces a cleaner, more readable code by providing blocking.

5.4-5 (4.4-5) FORTRAN 90 indicates these FORTRAN 77 features will be obsolete in future versions of the language.

5.4-6 (4.4-6) The use of '**&**' as the continuation character is useful in it's consistency for fixed format and required in free format. The only exception is continuation lines which exceed two or three lines, in which case (for fixed format) a numeric character in column 6 is useful for sequencing the lines.

5.4-7 (4.4-7) The use of the routine type and name is especially useful for files containing multiple subprograms that span more than one page (or screen).

5.4-8 (4.4-8) Excessive use of returns can make the code confusing and unreadable. Too many returns from a subprogram may be an indicator of cluttered logic. However, do not avoid return statements if it detracts from natural structure and code readability.

5.4-9 (4.4-9) The **assigned** and assigned **go to** statements are less readable, more difficult to analyze and more prone to programming errors. In addition, FORTRAN 90 indicates this FORTRAN 77 feature is obsolete.

5.4-10 (4.4-10) This format is required for discrete variables that encompass all or many of the allowable values and typically have different code paths for each value.

5.4-11 (4.4-11) Monotonically increasing numeric labels allows the maintainer to more easily analyze and trace the code logic.

5.4-12 (4.4-12, 4.4-13) Enhances readability and reduces confusion.

5.5 Types

5.5-1 (4.5-1) **Implicit none** shall be used in all program units. This ensures that all variables must be explicitly declared, and hence documented. It also allows the compiler to detect typographical errors in variable names.

5.5-2 (4.5-2) This format keeps the variable and constant name and value in one declaration.

5.5-3 (4.5-3) Multi-indexed arrays with the fastest index as the innermost, allows compiler optimization and faster execution times. In addition, this format keeps common data together when displaying data using an interactive debugging tool.

5.5-4 (4.5-4) Use of the **D** exponent character insures that the accuracy of floating point constants or initializations will not fall victim to the whims of compiler optimization.

5.5-5 (4.5-5) Use of the underscore form for exponentiation is prohibited because it is easily mistyped as a minus or dash. (-) which can cause havoc in mathematical expressions.

5.6 Complexity

5.6-1 (4.6-1) Parenthesis can be helpful in clarifying complex sub-expressions for authors and subsequent readers, as well as ensuring that order of evaluation and the precedence of terms are precisely understood by users of the code.

5.6-2 (4.6-2) Relational expressions are more readable and understandable when stated in a positive form. As an aid in choosing the name, ensure that the most frequently used branch in a conditional construct is encountered first. There are cases in which the negative form is unavoidable. If the relational expression better reflects larger structures in the code (larger than the condition statement under consideration), then inverting the test to adhere to this guideline is not recommended , for example when the positive choice is null, as in

```
if (.not. Found) then
   write (5, "(' Data in file ', A, ' does not exist')") Filename
   go to 9999
end if
```

is still better than

```
if (Found) then
   continue
else
   write (5, "(' Data in file ', A, ' does not exist')") Filename
   go to 9999
end if
```

5.6-3 (4.6-3) The intent of parameters passed through a subprogram should be declared to avoid inadvertent changes to variables intended to be for input only, or use of output variable in internal processing.

5.6-4 (4.6-4) Although most FORTRAN compilers zero numeric variables and blank character variables, there is no guarantee any particular compiler will, or will in the future, do so. All output variables should be initialized at the beginning of the subprogram to insure the return value(s) are set whether the routine completed successfully or there was an error.

5.6-5 (4.6-5) Subprograms which require a large number of input parameters or generate a large number of output values should create record types which can be passed as a block and thus reduce the number and confusion in the calling sequence.

5.6-6 (4.6-6) Variables passed through common blocks are essentially invisible, especially if the common block is inserted as an "include", and as such, can be inadvertently changed affecting other routines.

6. Sample Headers

6.1 Sample Program Header

```
Program Conv_Aero
!*************************************************************************
! Name: Conv_Aero
! Purpose: To read the traditional aerodynamic database file and convert it to
!           an XML formatted file.
!
! Modules: Conv_Aero.f90, xml_parser.lib
! I/O Units: 1 - Aerodynamic input file [86 column] (ex: 155.aero)
!            2 - Projectile family model (ex: M107)
!            3 - Output file (ex: 155.XML)
!            5 - Standard System input (Keyboard)
!            6 - Standard System output (Monitor)
!     History:     History     See     the     end-of-file     for     the     change
history!*************************************************************************
**
```

Note that the header contains the following:

- A description of the program, the original date and most recent modified date, with the initials of who modified it last)
- A list of Logical Unit Numbers (LUN) and what they reference
- A list of module filenames used, including the main program filename
- History. Each time the file is modified the history should be updated.

-

6.2 Sample Subroutine/Function Header

```
Subroutine FTPR (ILun, Format_Str, Data_Str)
!*****************************************************************************
! Name: FTPR
! Purpose - Formats firing data output and prints it to unit ILun
! Parameters - ILun       (In) : unit to which output is written;
!              Format_Str (In) : format string containing the processing information;
!              Data_Str   (In) : string containing the encoded data.
!
! Sample Format - (A4,.10,A5,-L,+R,.S10)
!
! Process Commands - B: check for blanks in the field with the decimal point;
!                    A: no processing, just copy to the output;
!                    .: (decimal point) - eliminate the decimal point;
!                    -: use the next character ("L" in this case) as a minus;
!                    +: use the next character ("R" in this case) as a plus;
!                   _S: append the opposite sign of the data to the data output;
!                   .S: eliminate the decimal point and append the opposite
!                        sign of the data to the data output;
! Process Notes - Commas must separate formatting descriptors;
!                    An E or * (F format overflow indicator) will blank the field.
!                    The open parenthesis, "(", and close parenthesis ")" are the
!                    start/stop indicators and are required syntax.
!
! History: See the end-of-file for the change history
!*****************************************************************************
```

Note that the header contains the following:

- A description of the routine, the original date and most recent modified date, with the initials of who modified it last)
- A list of input and output arguments
- A list of references (if applicable)
- History. Each time the file is modified the history should be updated.

6.3 Sample Subroutine/Function Trailer

```
!-----------------------------------------------------------------------------
! History:
! Version   Date         Modification
! -------   ----         ------------
! 0.1      29 Jun 2006   Initial. jeh
! 0.2      20 Aug 2006   Added XSL style sheet reference. jeh
!*****************************************************************************
```

7. Header Templates

7.1 Program Header Template

```
!*****************************************************************************
! Name:
! Purpose:
!
! Modules:
!
! I/O Units: 1 -
!
! [Notes:]
!-----------------------------------------------------------------------------
! History: See the end-of-file for the change history
!*****************************************************************************
```

7.2 Subroutine/Function/Module Header Template

```
!*****************************************************************************
! Name:
! Purpose:
! Parameters: X       (In) :
!             Y       (In/Out) :
!             Z       (Out) :
!
! Processing:
!
! History: See the end-of-file for the change history
!*****************************************************************************
```

7.3 Subroutine/Function/Module Trailer Template

```
!-----------------------------------------------------------------------------
! History:
! Version   Date         Modification
! -------   ----         ------------
!
!
!*****************************************************************************
```

7.4 File header template

```
!=============================================================================
!
!                        ***** NATO RESTRICTED *****
!
!     NATO ARMY ARMAMENTS GROUP, LAND CAPABILITY GROUP 3, SUB-GROUP 2
!                     TECHNICAL INFORMATION AND DATA
!
! Use and distribution of the enclosed information must conform to
! NATO STANAG 4683 "NATO Technical Shareable Software (NTSS)".
! The agreement defined in NATO STANAG 4683 is implemented when any or all
! of the enclosed information is incorporated into any software.
! ----------------------------------------------------------------------------
```

8. FORTRAN 90 Reference Manuals

1. FORTRAN 90 - Meissner, PWS Kent, Boston, 1995, ISBN 0-534-93372-6.
2. FORTRAN 90 - Huddleston, Exchange Publ. Div., Buffalo, NY, 1996, ISBN 0-945261-07-1.
3. FORTRAN 90 and Engineering Computation - Schick and Silverman, John Wiley, 1994, ISBN 0-471-58512-2.
4. FORTRAN 90 Concise Reference - Wagener, Absoft, 1998, ISBN 0-9670066-0-0.
5. FORTRAN 90 for Engineers - Etter, Benjamin/Cummings, Redwood City, 1995, ISBN 0-201544-46-6.
6. FORTRAN 90 for Engineers and Scientists - Nyhoff and Leestma, Prentice Hall, 1996, ISBN 0-13-519729-5.
7. An “Introduction to....” also exists: 1996, ISBN 0-13-505215-7.
8. FORTRAN 90 for Scientists and Engineers - Brian D. Hahn, Edward Arnold, 1994, ISBN 0-340-60034-9.
9. FORTRAN 90 Programming - Ellis, Philips, Lahey, Addison Wesley, Wokingham, 1994, ISBN 0-201-54446-6.
10. Problem solving with FORTRAN 90: for scientists and engineers - Brooks, 1997, Springer, 0-387-98229-9.
11. Programmer’s Guide to FORTRAN 90, third edition - Brainerd, Goldberg and Adams, Springer, 1996, ISBN 0-387-94570-9.
12. Programming in FORTRAN 90 - I.M. Smith, Wiley, ISBN 0471-94185-9.
13. Upgrading to FORTRAN 90 - Redwine, Springer-Verlag, New York, 1995, ISBN 0-387-97995-6.