

U.S. DEPARTMENT OF COMMERCE
National Oceanic and Atmospheric Administration
NATIONAL WEATHER SERVICE
Silver Spring, Md. 20910
June 9, 1992 W/OSD2:HRG

MEMORANDUM FOR: All TDL Programmers and Supervisors

FROM: W/OSD2 - Bob Glahn

SUBJECT: TDL Software Standards

Attached is the new IITDL Software Development and Documentation Guidelines," an update to TDL Office Note 79-13. Justification for this document is contained in its Introduction. Please read the document carefully and discuss any questions you may have with your co-workers, supervisor, or directly with me.

A draft version was provided to each Branch, and some of you were asked to review it. I seriously considered all comments; some I incorporated, some I didn't. I tried to make a reasonable and workable compromise between the standards used previously (and therefore much code exists which generally follows those standards) and what we might adopt if we were starting fresh. If the use in the future of new equipment requires changes/additions, we will make them then.

I cannot overemphasize the importance I attach to the use of software development and documentation standards. Each of you has an item in your GWPAS or Merit Pay Plan which addresses conformity to the guidelines. Following the guidelines for programming/documentation should get the same degree of attention and results as with other of our activities, such as writing memos or papers for publication. We strive to be "correct," but usually a few improvements can still be made no matter how hard we try. So, perfection, if there were such a thing in this partly judgmental area, is not expected, but good work and honest effort are.

Attachment

June 1, 1992

TECHNIQUES DEVELOPMENT LABORATORY SOFTWARE DEVELOPMENT AND DOCUMENTATION GUIDELINES

Harry R. Glahn

1. INTRODUCTION

The software development and documentation guidelines contained in this document are intended for use within the Techniques Development Laboratory and supersede those in TDL Office Note 79-13. However, the guidelines contained in TDL Office Note 79-13 remain 'essentially unchanged. Code conforming to TDL Office Note 79-13 will, in general, also follow the guidelines here. Enhanced capabilities of the FORTRAN language and compilers now existing have been considered.

Perhaps it is just as important to say what this document is not, as to say what it is. It is not intended to describe a complete software development method, complete with design documents, walk-throughs, etc. Those are important concepts and worthy of being implemented to varying degrees depending on the scope of the project and software. It is likely TDL will formally incorporate a more structured method in some of its development in the future. It is hoped that what is contained herein will fit into this formal structure with little change.

The critical importance of developing well documented and well structured code has become more obvious with time. Except for, possibly, some small programs/subroutines written exclusively to test an idea or structure that will soon be discarded, Government developed software will be inherited and maintained by others. "Tricky" coding in the name of efficiency is to be avoided (although the definition of tricky will vary with individual).

TDL is actually writing code for a number of hardware platforms, including the NOAA main frame NAS 9000 replacements; AFOS DG Eclipses; IBM 286, 386, and 486 micros (including clones); and the VAX-based DAR³E system.¹ Some of this code is being furnished to others for their use (e.g., AFOS profiler applications, SAO decoder, verification collection and collation, etc.) and this will become much truer in the future, as we develop "Category I" code for the AWIPS contractor. It is imperative that we follow good coding and documentation rules in the development of all code, and in particular code that is to be handed off for use outside of TDL. Reasons include:

- Most projects today are more than one-person efforts. With several persons involved in a project, it is important that guidelines be followed so that all can easily "read" another person's program.

June 1, 1992

- Usually, it will fall to someone other than the originator to modify or maintain a program at some time in the future. Again, if a program has been written and documented according to prescribed rules, revisions and maintenance are much easier. This applies to external documentation as well as the code itself.
- Some of our code will be developed for the express purpose of handing off to a contractor for implementation and integration into a much larger system. If all such code follows the same guidelines, understanding and dealing with it will be much easier, and we will be able to answer questions more readily than otherwise. Documentation will, of course, be mandatory.
- Standardization will reduce errors in coding and keystroking. The eye and mind become accustomed to "patterns," and a break in pattern may be an error. If there are no established patterns in the code, or if the patterns are considerably different from those to which the reader is accustomed, this human error detection feature cannot operate effectively.
- Converting a body of software from one computer system to another is easier if it is all written and documented to the same standards.
- Persons writing code and having it keystroked by others need not explain a preferred format; it will already be defined. Documentation may be assigned to a person other than the one writing the code; an established procedure makes individual coordination on a documentation format unnecessary.
- New employees with little or no programming experience can be more easily trained in good procedures if those procedures are written down and everyone in the Lab follows them.
- Some simple optimization procedures, if followed, can reduce execution time considerably. However, the primary purpose for TDL guidelines is not central processor optimization. Also, what is optimum for one system may not be for another.

¹No endorsement of a particular vendor or product is implied by anything contained in this document.

June 1, 1992

In summary, the objectives of the TDL guidelines are to enhance clarity, testability, maintainability, and person-to-person and computer-to-computer transferability of software throughout its life cycle.

Any system of software guidelines or standards is somewhat arbitrary. Different organizations have different standards, and textbooks do not agree. It is not so important exactly what the guidelines are, as it is that there be guidelines (assuming some semblance of reasonableness, of course).

This document is divided into two main sections: Coding Guidelines and Documentation Standards, the latter referring to external (to the code) documentation. It is recognized that the different platforms and the associated compilers will have somewhat different characteristics; however, this document is written in general terms with mention of a particular platform where necessary.

2. CODING GUIDELINES

The programming language to be used is the version of FORTRAN appropriate to the platform for which the code is intended. FORTRAN 77 (or its successor when available) shall be used whenever available. Other languages will be used only in exceptional cases and only after the formal approval of the programmer's supervisor, who shall make it known to the senior staff of TDL before coding starts. Vendor-specific extensions to the FORTRAN 77 standard will be used only in exceptional cases.

Appendix I provide an example to which the reader should refer when reading the following guidelines.

Documentation Block - Every program and subroutine must start with a documentation block following the outline in Appendix I. Starting column convention is imposed to promote readability. Generally, in the absence of specific guidelines, standard typing rules should be used in preparing the documentation. If the system being used supports lower case characters as well as upper case, then it is optional which is used for the documentation block. Lower case for documentation does distinguish that material from executable code (which shall be upper case) but does add a degree of complexity and non-uniformity among programs/programmers.

Program Name - The first line should be the subroutine name starting in Col. 7. If it is a main program, and the compiler doesn't permit a program name, substitute a Comment Card with the program name. For main programs in the MOS system, use the naming convention in TDL Office Note 74-14, Chapter III. For main programs in the LAMP system, use the naming convention established in the existing program documentation.

June 1, 1992

Date, Programmer, Organization, Computer - Maintaining the exact date is not important; it is not used, for example, as the date the routine was added to the library. The month and year are sufficient. Starting in Col. 10, the date, the programmer's name, TDL, and the computer system the program was written for are each put on the third line, after a blank comment line, separated by three spaces. Extra lines should be used here to indicate modification dates, etc., as appropriate. Spacing may be adjusted to "line up" names, etc.

Purpose - Following another blank line, the next line should contain the word `PURPOSE` starting in Col. 10. Following that will be a short paragraph explaining the purpose of the routine. This need not be extensive, as details can be placed in the program writeup (external documentation). However, it should be complete enough to be useful to the user. If the routine was written specifically for a calling routine, the comment `CALLED BY XXX` is useful. Start all lines in this paragraph in Col. 14.

Data Set Use - After the paragraph on "purpose" and a blank line, the next line should contain `DATA SET USE` starting in Col. 10. Listed below this line will be data set names followed by a brief explanation of them (see Appendix I). The explanation should state whether the data sets are input, output, or internal. If no data sets are used by this routine, put `NONE` on the line following `DATA SET USE`.

Variables - The statement following those explaining data set use should contain the word `VARIABLES` starting in Col. 10. Following that, most, if not all, variables used in executable statements in the program should be defined in the format shown in Appendix I. The equal sign should be in Col. 23 followed and preceded by one space. All lines except the one defining the variable start in Col. 25 unless some further indention seems appropriate, such as in lists. (Standardization here will allow copying from one routine to another when the variable is used in more than one routine. However, many times the explanation will have to be changed slightly for it to pertain to a particular routine.) Variables appearing only in `COMMON` need not be defined, but when a variable is used in `COMMON` and in other places in the routine, it must be defined. Variables used only to pass on to another subroutine should be defined, but the definition could be something like `"PASSED ON TO ANOTHER SUBROUTINE."` The cross reference list of the compiled source will identify where the variable is passed on.

List all variables in the subroutine call sequence, if any, first and in order. No other ordering is mandatory, but some logical sequence should be used and the best one to use may depend on the routine. The ordering might be alphabetical, especially if there are many variables. The order could be the approximate order the variables are first used in the program, especially the input variables; having the definitions of the input variables from an external source all in one place, and in order, has proven to be very useful. For each variable that is in the call sequence,

place at the end of the comment either (INPUT), (OUTPUT), (INPUT- OUTPUT) or (WORK AREA) to indicate its use in the subroutine. (This is not appropriate for a main program.) This should also be done for variables actually used that are in COMMON. As a useful option, variables not in the call sequence or in COMMON can have (INTERNAL) placed at the end of the comment. If, and only if, the type of variable is other than INTEGER*4 or REAL*4, place the type in parentheses at the very end of the comment, e.g., (CHARACTER*5).

Another option for grouping variables is to have sections headed INPUT, OUTPUT, etc. (starting in Col. 14) and to put the appropriate variables under these-headings.

Non-System Subroutines Used - The non-system subroutines used in the program are listed, separated by a comma and space and indented to Col. 14, following the section heading NON-SYSTEM SUBROUTINES USED, starting in Col. 10.

Declarative and Data Statements - Such statements, if any, should immediately follow the documentation block. An order such as PARAMETER, COMMON, TYPE, DIMENSION, EQUIVALENCE, and DATA is appropriate. Always use PARAMETER first, and DATA last.

PARAMETER - PARAMETER statements shall define a variable only where a DATA statement will not suffice, namely, in the definition of variable array dimensions or, rarely, when a computation is desired within the definition to retain the computed formula. The cross-reference lists provided by some Compilers do not treat variables defined with PARAMETER statements the same way as other variables, and some ignore them altogether; this makes checkout more difficult. This convention will let the user know that any variables defined in PARAMETER statements are variable dimensions.

COMMON Blocks - COMMON blocks should be used sparingly, if at all. Generally, code is easier to follow when the variables needed are passed through the call sequence rather than in COMMON, especially when some of the variables in the COMMON are used and some are not. Having variables in COMMON can also make it difficult to modify a program that has many subroutines. In any case, all COMMON should be labeled. The name of the block should be rather unique to keep to a minimum conflicts that might arise when a routine is used by others. For instance, XXXONE might be a good name for a program named XXX; this would be better than BLOCK1.

Type Statements - Type statements should not be used unless the type is "unusual." The CHARACTER type is unusual in this sense and is needed for character variables. Do not use type statements for REAL*4 or INTEGER*4 variables. (See Variable Naming below.)

June 1, 1992

Variable Naming - The FORTRAN predefined specification of integer and real variables shall be followed--INTEGER(I-N), REAL(A-H,O-Z). By using this convention, it is much easier to catch integer/real conversion errors than if the reader has to remember the type of all variables in a specific routine. With the advent of FORTRAN 77, reserving the letter 'C' for CHARACTER variables in new code is recommended. Do not use the IMPLICIT statement. FORTRAN 77 limits the name to 6 characters; we should honor that limit, even though many compilers allow more. Variables used for only one purpose (e.g., to hold values of dew point temperature) should be given easily recognizable names (e.g., DEWP). (Using an array for multiple purposes may make this difficult, if not impossible, but equivalencing should not be used to overcome the difficulty.) Generally, the use of single characters, such as "I" and "J," should be reserved for DO loop indices. In two-dimensional grid indexing, the use of "IX" for left to right and "JY" for bottom to top is a good practice, and the convention of using the first index to refer to the "IX" direction is mandatory.

EQUIVALENCE Statements - Equivalencing variables tends to make code harder to follow, and encourages mistakes. It may also hinder optimization in some compilers. Only in special cases or where much memory can be saved should equivalencing be used.

DATA Statements - When values are specified in DATA statements, try to arrange them so that they can be easily read. This is especially important for multiply dimensioned arrays. Put on separate lines whenever practicable values pertaining to different dimensions.

In-Line Documentation - In-line documentation should be provided at appropriate points in the program. Somewhere between 10% and 50% of the total lines should be devoted to documentation (besides the documentation block). The comments are used to explain the code and should be subordinate to it. Therefore, with code that has executable statements starting in Col. 7, start all comments in Col. 10. One should expect to "read" the code with explanation by the comments, rather than vice versa. (Indentation for IF THEN ELSE structures with accompanying comments will be treated later.) A block of code can be explained before the block by comments separated above and below by a blank line ("C" only on the line). A single line of code can be explained by a single comment following (or preceding) the executable line with no blank line. A comment should be used to explain the purpose of a called subroutine. Comments can be either upper or lower case, but the usage shall be consistent within a routine. Clarity is many times enhanced by inserting a blank line after a branch-type instruction.

Length of Programs - Program (subroutine) length (number of lines of executable code) should be governed by the function of the routine, and not by some arbitrary rule such as "all programs will be between 10 and 100 lines of code." A specific maximum size is not

as important as convenient program structure. Modularity is important when meaningful, and it usually is.

June 1, 1992

Top Down Coding - Program flow should be from the top down. With the IF THEN ELSE type of structures of FORTRAN 77, this is always possible with enough nesting. It is usually possible to do this even when the GO TO construction is used. Some slight duplication of code may be preferable to branching. In all cases, it is the clarity of the code that is important. It may be confusing to have nests more than, say, 6 deep. On the other hand, if a program essentially repeats itself when input data so indicate, a branch from somewhere (usually near the end) back to (near) an input statement should not be confusing, and may be more "natural" than trying to accommodate this option with an IF THEN ELSE construction.

Statement Labels - Statement labels should always start in Col. 2, no matter how many digits they contain. Number only those statements to which reference is made (i.e., only those it is necessary to number). Most cross reference lists will indicate any statement numbers that can be removed.

Some logical numbering sequence must be followed. Some possibilities are:

The numbers range from I through 9999 and be in sequence.

The numbers always contain 4 digits and are in sequence.

The primary numbering system start at 100 or above and end at 999, but, upon revision, when it is necessary to insert more numbers than space has been provided for, a fourth digit is added. Since all numbers start in Col. 2, they 'appear' to be in order even though 1115 comes between 111 and 112 (this may be slightly inconvenient in some compiler's cross reference listings, as all 3-digit numbers may precede 4-digit numbers). Although this method may seem at first glance to be more complicated, it is really very simple and workable; most MOS and LAMP programs use this system.

Statement Format - For programs that are-basically not in the IF THEN ELSE structure, start all statements (except comments) in Col. 7. Continuation statements should be indented by at least 5 spaces unless there is a reason to do otherwise (a FORMAT statement can usually be split between lines with no problem--even a string of characters can be stopped and restarted on another line). Limit the line length to the FORTRAN 77 standard, 72 characters.

Statements should not include blanks unless they are necessary to improve readability. Establish a pattern and stick with it. Examples as used in MOS and LAMP programs are:

```
SUBROUTINE INTR(P,BY,BX,BB)
```

```
DIMENSION SAVE(2,2),P(61,81)
```

June 1, 1992

```
EQUIVALENCE (P(1,1),NPK(I)),(X,Y)
```

```
COMMON/M400/VRBL1(IO),VRBL2(10)VRBL3(100),  
1      VRBL4(1000)
```

```
CALL RDMOSH(N,NWDS,NROWS,NCOLS,JDATE,NERR)
```

```
.WRITE(KFIL12,130)KDATE(MT),JDATE
```

```
130 FORMAT(' THERE IS A PROBLEM WITH THE INPUT DATA NEEDED,  
KDATE -II8,'. 1,      DATE FOUND IS -'I8)
```

```
X-IB(J)+IA(K)+3*(K+IC(J)**4)+M/N
```

```
CHARACTER*3 CWSFO,CNODE,CTIME(10)
```

```
DATA NCRIT/2,1,1,1,1/
```

```
PARAMETER (ND2-41,  
1      ND3-39)
```

```
STOP 115
```

Continuation Lines - Continuation lines can be denoted by the sequence of numbers 1 through 9, then alphabetically starting with A. occasionally, it may be desirable to start the sequence with 2 rather than I in DATA statements. As an option, the same character can be used for all continuation lines.

Spaces Versus Tabs - When spacing over to where a statement, statement label, or comment is to start, use the space bar, not the tab.

CONTINUE Statements - Continue statements should be used only where necessary, except a CONTINUE is always used at the end of a DO loop. End each DO loop with a separate CONTINUE statement even though this is not logically necessary. This serves the purpose of notifying the "reader" that this is the end of a DO loop, and may aid in optimization for some compilers. Each nest of nested DO loops will have its own CONTINUE (unless, perhaps, the loops are extremely short and all end at the same place).

DO Loops - A blank comment should immediately precede a DO statement and follow the DO loop's CONTINUE statement. For very short, multiple nests, a separate blank for each loop is not needed.

June 1, 1992

FORMAT Statements - Format statements should be used in the code where they are referenced, and should be numbered in sequence along with other numbered lines. A FORMAT statement should immediately follow the first I/O statement which refers to it. For ease of possible later modification, it may be best to duplicate a FORMAT statement, except for its number, so that it can be with the statement that refers to it. If multiple statements refer to the same FORMAT, later modification may remove (or renumber) the FORMAT, even though it is referred to elsewhere in the program, and a compile error will occur. When looking at the printed output and the code that produced it, it is much easier to match the output to the FORMAT statement when the FORMAT and the I/O statement are together.

Indentation - Several rules for indentation of statements are given above in connection with other topics. In general, when the GO TO structure predominates, start executable statements in Col. 7 and comments in Col. 10. For IF THEN ELSE structures, some indentation shall be used. One option is to indent each "nest" another 3 spaces. Comments could be indented 3 more spaces, or it may be more convenient to continue to start them all in Col. 10. Whatever convention is adopted for a routine, it must be used consistently within the routine.

I/O Device Reference - Device reference by FORTRAN number should be with an INTEGER variable, not a constant. For main programs, this variable should be given a value in a DATA statement. For subroutines, this variable should be passed through the argument list, after being defined in the main program. In some cases, it may be more convenient to read the variable name from a control file. (The convention in LAMP programs is KFILI for Unit No. 1, etc.)

Variable Dimensions - Whenever there is a chance that the dimensions of a variable will be changed, and always when the dimensions are referred to in other statements (for example, to keep from overflowing the array), the dimensions should be declared by defining a variable in a PARAMETER statement. The actual number should never be referred to in the code, but rather referred to by the variable name used in the PARAMETER statement. Usually, variables and their dimensions should be carried to subroutines through the argument list.

Subroutine Call Sequence - No matter what rules are established, exceptions will occur. Common sense must prevail. However, to the extent practicable, the order should be as follows:

If data set reference numbers are provided, put them first.

Other input to the routine should follow.

Variables used for both input and Output or work area should then follow.

June 1, 1992

Output variables, ending with an error (return) code (if any) and finally the alternate return symbol(s) (return to a statement number---FORTRAN 77 uses an * for this purpose in the SUBROUTINE statement) should come last. Alternate returns should be used sparingly (if at all), as following the program logic is usually more difficult than using an error (return) code and checking it for desired branches.

Variable dimensions for an array or arrays should follow the last array name in which they are used. Multiple dimensions passed for an array should occur in the same sequence as they occur in the DIMENSION statement. For extensive call sequences, the dimensions could all be put together near the end.

Subroutine Entry Points - Each subroutine should have only one entry point. Do not use the ENTRY statement.

End of File and Error Checks - Error checks for input should be used. Errors can be indicated by an error code returned to the calling routine (preferably with a print of the diagnostic in the routine itself), or exit can be made to an error handling routine. In case the error is fatal, it may be all right to stop in the routine itself with an appropriate diagnostic (see Program Termination below).

Error Codes - Some operating systems and associated routines may have a convention that should be followed. (For instance, the Data General Eclipse expects a "I" for the no error condition.) Whenever possible, the no error condition should be "O." Use these as INTEGER variables, not as, for example, LOGICAL.