

Taming the Chaos: Standards for SAS® Programming

SSgt C. Michael Whitney, Air Force Combat Climatology Center

ABSTRACT

We've probably all seen it: poorly written code with cryptic variables, untested and undocumented monstrosities full of spaghetti code, and other nightmarish programs that should have been locked in a closet instead of being loosed upon the world. What can be done to exorcise these programming demons? A strong set of programming standards that all code must meet! This paper will give an overview of what should be in your standards and discuss documentation, program life cycles, and other topics that will help you tame the programming chaos.

INTRODUCTION

The purpose of this paper is to provide a framework for high-quality, structured, and easily maintainable SAS software. Uniformity of software development is essential in creating and maintaining computer systems that operate at peak efficiency.

The guidance presented here is used by the Air Force Combat Climatology Center (AFCCC - formerly the USAF Environmental Technical Applications Center), and is based on Air Force standards. It is, however, applicable to any organization, be it governmental or corporate in nature.

To be effective, programming standards should apply to all programs created in your organization (otherwise they wouldn't be standard, would they?). Programs transferred from other environments should be brought up to standards as maintenance is performed, if possible. (However, some code is so bad, it should be treated as "black box" and modified as little as possible).

SOFTWARE DEVELOPMENT LIFE CYCLE

The software development life cycle describes the events which take place from the determination that a software application will be required for customer support, to the cataloging of the application. At CCC, most projects start with a weather analyst receiving a request from the field, and then either developing the software him or herself, or tasking the applications programming division to develop the software.

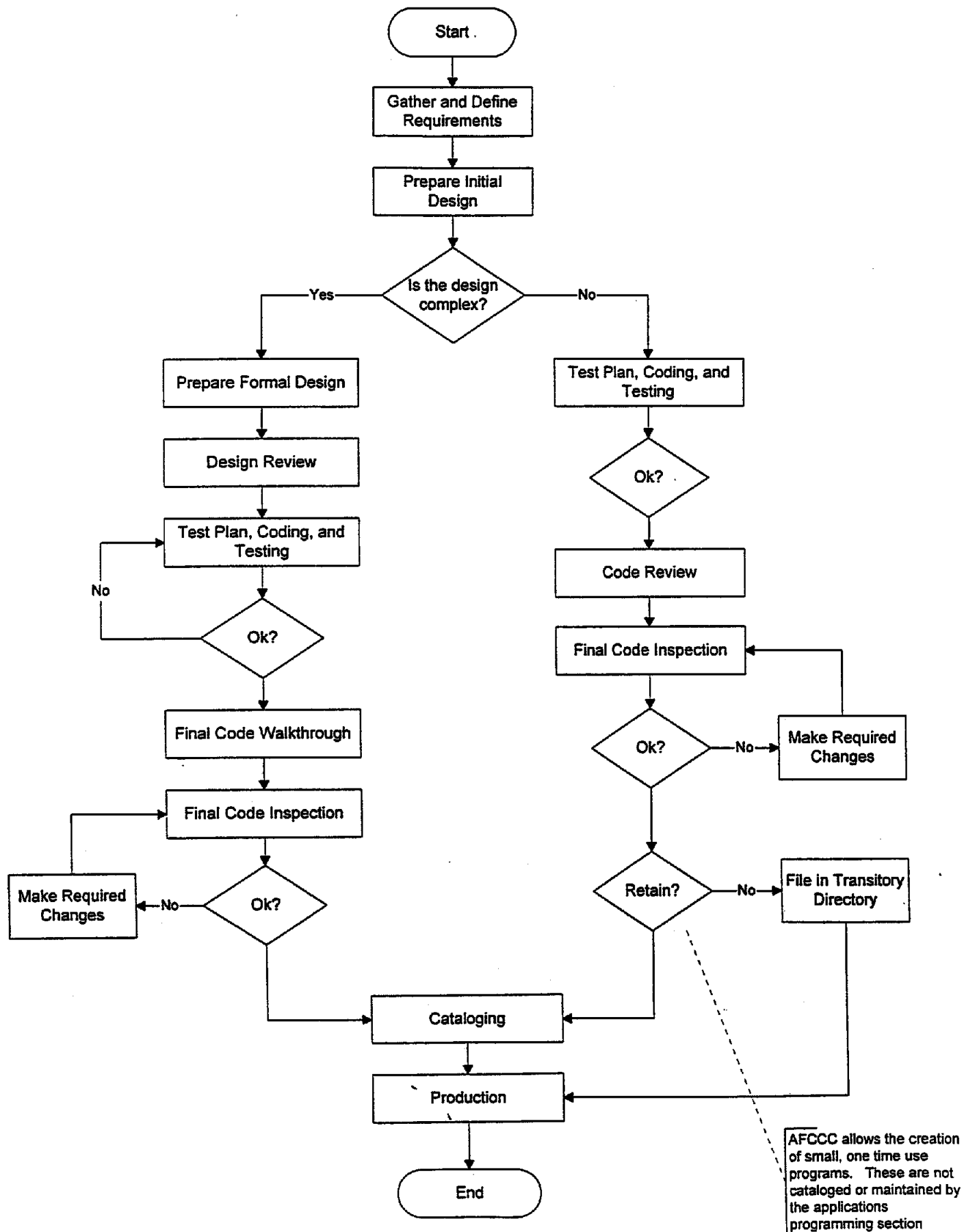
The development life cycle identifies important phases associated with the process of developing software. When completed, each phase should significantly reduce future maintenance costs, and minimize the chance for software errors to remain in the final product. The amount of detail required for each phase should be directly related to the size and complexity of the project. Large, complex projects require more extensive documentation than do small projects.

Overall Life Cycle

There are several major phases in the software development life cycle. Each phase should be completed and documented before proceeding to the next. Figure 1 provides a summary of the software development life cycle. The following is a brief description of the major phases:

- a. **Gather & Define Requirements.** The first task is to determine if existing software can be used to fully or partially meet the customer's requirements. Analysis and research are the foundation on which the rest of the project lays.
- b. **Prepare Initial Design.** Should software development or modification be necessary, the programmer will formulate the initial requirements and design documentation. The more detail that can be provided, the better and more easily understood the design will be. This design should be presented to the customer for approval, insuring that it meets the customer's requirements.

Software Life Cycle Flow Diagram at
Air Force Combat Climatology Center



- c. Formal design and design review. The detailed design is a joint effort between the programmer and the analyst to construct a concise, logical solution to the problem which the programmer can easily translate into code. The design will be reviewed before coding begins.
- d. Test plan, coding and testing. Appropriate coding standards will be followed for all production software. The development and execution of a test plan is essential to ensure the customer receives quality data. After testing is completed, the software will undergo a review or walk-through prior to production.
- e. Final Code Inspection. This is an intense, line-by-line review of the software prior to cataloging.
- f. Cataloging. This is the process of placing approved software into the Production Software Libraries. It is extremely advantageous to place production code in libraries where only the software librarian has write authority. This will prevent accidental corruption of the source code, and ensure that everyone is running the same version for production.

TEST PLAN AND TESTING

Based on the requirements and design documentation, an initial test plan should be created. This plan should be a list of the tests to be made to verify the correctness of the results. The test plan will identify the test data to be used. The test plan should be updated as necessary during testing to reflect the actual testing done. All software requires testing by both the programming team and the customer before it is run in production or cataloged. Testing ensures that customer requirements are met or exceeded as much as possible, coding and logic errors are discovered and corrected and that each routine does what it was designed to do.

Testing should follow the test plan. Update the test plan when new or additional tests are required, or if tests described on the test plan become unnecessary or too difficult or impractical to perform. As a minimum, software should be tested as follows:

- a. Test the program with both actual and test input data. Testing should include challenging the program with data both in and out of the testing parameters. This is done to ensure the program either stops or rejects the bad data.
- b. Test all modules to the maximum extent possible with valid data. If possible, make sure each decision is executed at least once. Do hand calculations, if necessary, to verify that each module is functioning properly. Blackbox modules need not be tested in isolation.
- c. It is normally helpful to have someone else test your program. Often another person can discover awkward or cumbersome procedures or manage to break the program with erroneous data.
- d. After modular testing, test the entire application ensuring all programs and/or subroutine calls are working. Be especially aware of the arguments and units. Often, different arguments (or magnitudes) are required for different subroutines. Make sure the correct arguments with the correct units are passed to each subroutine.
- e. An acceptance clause should be included at the end of the test plan. This should be signed by the customer and the programmer at the conclusion of testing.

REVIEWS

Proper use of both reviews and walk-throughs will result in better software products and reduce long-range maintenance costs. A review is a very informal check of a portion of a software program which can be conducted at any point in the development or maintenance processes. Very little documentation is needed and structure is of no concern. Two or three individuals are sufficient for a review.

Program reviews may be made at any point in the program development cycle. The best times to review a program are after the design is developed, before any formal walk-through or code inspection, and before cataloging

and production. Periodic reviews of all programs, either being developed or modified, are essential to ensure programming standards are adhered to, errors are detected (as undetected errors will haunt you later), and ensure documentation is correct. Make entries in the project log of each review. A properly kept log will help trace any problems that may arise later on in the project.

WALK-THROUGHS

A walk-through is a group evaluation of a product at various stages of its life cycle. Walk-throughs should be formal, properly structured and well-documented. Proper structuring will make the walk-through more beneficial. Walk-throughs allow you to produce reliable, error free code. They can reduce the average from 3 to 5 errors per 100 lines to as few as 3 to 5 errors per 10,000 lines. They can help you correct design flaws and improve program documentation, as well as cut production time by as much as 50 percent. They also help increase the quality of system software. There are several types of walk-throughs:

- a. Design walk-throughs focus on the solution to the problem. It is critical in that it sets the guidelines on HOW a project is to be completed.
- b. Periodic walk-throughs are conducted whenever thought necessary. Periodic walk-throughs will tell you WHERE a project is.
- c. Final walk-throughs are necessary prior to submitting a program for a final code inspection. This walk-through will help find any discrepancies previously missed.

A minimum of three and a maximum of seven individuals should be involved in any type of walk-through. The number of attendees should be determined by the size and scope of the project. One individual moderates the walk-through. A second individual records all pertinent information discussed during the walk-through, such as recommended changes to the material being presented. A third individual presents the material.

WALK-THROUGH STAGES

There are three stages of the walk-through process:

- a. Review stage. This 3 to 5 day period prior to a walk-through is used to acquaint each attendee with the product. Standards, checklists, material to be 'walked through', and any relevant document from prior reviews or walk-throughs will be looked over during this stage.
- b. Walk-through stage. In this stage, each detail of the product is reviewed. the presenter or moderator guides the meeting which should last from 1 to 2 hours, no more.
- c. Follow-up stage. It is here that changes are implemented. All involved parties are informed of the changes and should agree with them.

HELPFUL HINTS

Here are a few very simple guidelines to remember while conducting a walk-through:

- a. The author is not on trial.
- b. The product is guilty until proven innocent.
- c. Choose walk-through participants carefully. Avoid personality conflicts if at all possible.
- d. Keep walk-throughs within the predetermined time limits.
- e. Schedule well in advance and make it formal, but not too formal.
- f. Create and follow a checklist of possible problems.

There are some people-problems to watch out for in reviews and walk-throughs, however. Egos can play a factor in that people naturally do not like to be told they made mistakes. Another problem area is inexperience at giving and receiving criticism. (Misdirecting comments at the creator rather than the code can turn the meeting into a defensive war). The final problem is apathy - not trying hard to find errors. (Don't assume that others will find the same errors that you find).

The team only has three decisions to choose from at the conclusion of a walk-through: accepting the product as-is; accepting the product with revisions, trusting the creator to make the fixes; and moving that another walk-through be conducted when the errors have been corrected and the comments for improving the product have been implemented.

TOP-DOWN STRUCTURED DESIGN

Software should be developed in a top-down structured manner. Top-down design begins with formulating the solution in terms of generalized statements. Once the general algorithm has been developed, it can be refined by adding the details that are necessary to perform the general actions. For a complicated problem, this refinement process may be repeated several times, with each version containing more detail than the last. The design proceeds from the top (most general) to the bottom (most detailed), with the resulting design reflecting the nature of the problem. Top-down structure makes the program highly readable and easier to follow.

Structured software consists of separate functional modules. Modules may call other modules, and in many cases several modules may be required to complete a single function. Modules are designed with control flowing from the top to the bottom.

Top-down structured software should adhere to the following characteristics:

- a. Cohesion. All modules should perform single functions or small, related functions that require common data and pass information from one step to another.
- b. Coupling. Connections between modules must be obvious and should be minimized as much as possible for simplicity. Data will be passed as arguments between modules, when applicable.
- c. Limited data exchange. A minimum amount of should is to be passed between modules.
- d. Exit and entry points. Modules should have one entry point and one exit point.
- e. Span of control. Never let a module (except the program driver) directly control or call more than seven subordinates.
- f. Scope of effect. Subordinate modules are modules that make decisions so that other modules can complete a function. The scope of effect of any module includes all subordinate modules that are necessary for the completion of the modules function.
- g. Module size. Modules should be no more than four pages (200 lines) of executable code, minus comments. Modules should not perform more than one unrelated function, but may handle more than one related function for the program. If a module grow larger than 200 lines, check to see if the module is accomplishing more than a single function. If so, attempt to break it down into single-function modules. However, do not cut a single-function routine into multiple pieces simply for the sake of module size.
- h. Independance. The execution of the module is completely independent and does not depend on anything that happened in previous invocations.

A module is a sub-portion of a program and is composed of a bounded group of instructions with a single identifier. A module may be a subroutine, function, or driver. In SAS terms, a module could be a LINKed subprogram within a large dataset, or a macro. (Macros can provide great modularity!)

Top-down structured software directly results from the use of structured techniques and tools such as data flow diagrams, flow charts, pseudo code, structure charts, data dictionaries, and organized documentation. It requires forethought and work on everyone's part.

Top-down structured design has the following advantages:

- a. It allows random or re-usable, modular coding, testing, and implementation.
- b. Design problems are detected earlier when they are cheaper and easier to correct.
- c. Module development allows the programmer to concentrate solely on individual modules while treating other modules as black boxes.
- d. Fewer programming errors are likely to be made due to modules significantly reducing program complexity.
- e. Debugging is made easier due to modularity. This shortens debug time since problems are quicker to isolate in any particular module.
- f. Modules are more easily used by other programs.

DOCUMENTATION

All program modules should be documented.. The documentation must be standardized to promote uniformity in the program library. Main program documentation should contain the following information in the order presented and in the format presented in the samples in this chapter. Make sure that any additional information in the documentation is pertinent.

- a. **PROGRAM ID.** The program ID is simply a repeat of the program name.
- b. **PROGRAMMER.** Enter the name of the original programmer. If the program has be rewritten, add a second PROGRAMMER entry with the name of the responsible programmer.
- c. **CREATION DATE.** This is the date of the original program and the project number used for the creation (if known). (All of AFCCC's programs have project numbers to allow for tracking).
- d. **AS OF DATE.** This is the date of the completion of the most recent update and recataloging of the program and the project number for which it was changed.
- e. **PURPOSE.** Insert a descriptive narrative concerning the purpose of the program. Keep it short and to the point, and make it understandable. Update whenever the program is modified.
- f. **METHOD.** Tell how the program works to fulfill its purpose. Use enough detail to get the point across without being confusing. You may use blocked paragraphs or structured English format, whichever is more understandable.
- g. **REQUIRED SUBROUTINES OR MACROS.** Make a list of all required system (external) and embedded (internal) subroutines used by the program. With each subroutine name, include a very brief description of the subroutine. You should differentiate between system and embedded subroutines, and between system and embedded functions.
- h. **SAMPLE DRIVER.** For mainframe applications that run in batch, include a sample of the JCL or name of the CLIST used to run the program (if applicable). Include as many notes as needed to enable anyone to build the JCL string or access the CLIST and run the program. Include a description of any data input cards that are used in the JCL. References should be made to any samples already on the system that can be used. For micro/mini-computer programs, list how the routine is called, including arguments, and a sample script if one is used.

- i. **INPUT/OUTPUT.** Provide a complete description of the input data and the output produced. This section should agree with the sample driver section. This section should include any permanent datasets and macro variables needed to run the program.
- j. **NOTES.** Any necessary notes should be entered. This is also a good place to list any references used for making the program. This should be the final input before the history section begins. Notes may be entered anywhere in the documentation that is deemed necessary, but a grouping of the notes is easier to follow.
- k. **HISTORY.** The history section is the key to properly modifying or repairing the program. All entries in the history section must be as complete as possible. History entries should be completed for each recataloging of the program and will include the following information:
 - 1. The date of the modification and project number.
 - 2. The name, title, and section of the person or persons who worked on the program.
 - 3. A complete description of why the program was worked on, what and where changes were made, and the results of the changes. Also, include any differences between the old and new versions.
 - 4. Additional information may be included in the documentation as needed, but the documentation should not exceed eight printed pages (400 lines) as a good rule of thumb. A general overview of the program is all that is needed to fulfill documentation requirements.
- l. **Data Dictionary.** All variables (including macro variables) used in a program will be defined in the Data Dictionary in alphabetical order. This includes all scalars, vectors and arrays and their dimensioned values.

Within the body of the program, individual sections of code should be documented as well, describing what and why a particular code segment is being done. Be as precise as you can, it might be you that has to maintain that code a year from now when you've forgotten just exactly what that segment was supposed to be doing.

Without good documentation, maintenance becomes much more difficult. We're probably all heard the old hack about no documentation being good job security, but even the best programmers can't remember every detail about what they themselves wrote just a few months before. Poor documentation hurts everyone.

```

/*****
*
*   PROGRAMME      FREQDIST
*
*   PROGRAMMER:    SSGT MAX MINION
*
*   CREATED ON:    01 APR 1999
*
*   AS OF DATE:    21 MAR 2002
*
*   PURPOSE:       THIS SAS ROUTINE READS THE DATA FROM TABLES CREATED BY
*                  "FORECAST". THE DATA FROM THE RAIN TABLES IS USED TO
*                  CREATE ANALYSES OF RAINFALL BY LOCATION AND TIME. NOTE:
*                  VALUES IN AREAS WHERE RAIN IS SPARSE MAY BE SKEWED DUE
*                  TO LIMITED DATA.
*
*   METHOD:         IN THE RAIN TABLES, THE RAINFALL AMOUNTS AND RAINRATES
*                  ARE ALREADY IN ORDER BY TIME (UNLESS THE USER SELECTED
*                  RANDOM TIMES) SO UNDER MOST CIRCUMSTANCES, LITTLE OR NO
*                  SORTING IS NECESSARY. THEREFORE THE USER MUST SORT THE
*                  DATA PRIOR TO EXECUTING OF THIS PROGRAM. THIS PROGRAM
*                  TAKES THE SORTED DATA AND BY USING PROC MEANS AND PROC

```

```

*
*           FREQ PRODUCES FREQUENCY DISTRIBUTION OF RAINFALL BY THE
*           TIME OF THE REQUESTS IN "FORECAST".
*

```

```

*   SAS DATA SETS USED:   SAS.WORLD
*                           WEATHR.DATA
*                           WEATHR.RAINRATE
*                           WEATHR.RAINAMT
*                           STATION.FILE
*

```

```

*   MACROS CALLED:        %READRAIN
*                           %READDATA
*                           %READSTN
*

```

```

*   INPUT DATA FILE DESCRIPTION:
*
*           POINTS   GEOGRAPHIC POINT OR AREA
*           WXPARM   WEATHER PARAMETER TABLE
*           FREQNC   FREQUENCY OF INTEREST TABLE
*

```

```

*   INPUT VARIABLES:
*
*           RAINSUM   QUANTITATIVE SUM OF PRECIPITATION
*           PTAREA    POINT OF AREA OF CONCERN
*           TIMEINT    TIME INTERVAL OF REQUEST
*           JULHR      FIRST HOUR OF REQUEST
*           JULAST     LAST HOUR OF INTEREST
*           SCALE      SCALE FACTOR
*

```

```

*   OUTPUT:
*
*           RAINAMT    QUANTITY OF RAIN OBSERVED
*           RAINRAT *   RATE OF RAINFALL OBSERVED
*           CORCOEF     CORRELATION COEFFICIENT
*

```

```

*   NOTES:      *   IN MM/HR
*

```

```

*****
*
*   HISTORY:
*       01 APR 1999   SGT MAX MINION, WEATHER ANALYSIS SECTION
*                   CREATED FOR WORLD PRECIPITATION STUDY
*
*       21 MAR 2002   SGT MAX MINION, APPLICATIONS PROGRAMMING SECTION
*                   UPDATED TO INCLUDE CORRELATION COEFFICIENT
*

```

```

*****/
/***** DATA DICTIONARY *****/
*
*   VARIABLE      DESCRIPTION
*
*   DATE          YEAR, MONTH, DAY, HOUR; YYYYMMDDHH
*   FACTOR        .03937 FROM USAFETAC/SYT. THIS IS USED TO
*                   CONVERT FROM INCHES TO MILLIMETERS
*

```

COMMENTING THE PROGRAM

Comments should precede the executable code and be set off in a uniform manner. In-line comments should be right justified for readability. Each block of code should have a preceding block of comments pertaining to the workings of the code. Blocked comments ought to be set off with noticeable borders or blank lines. A

continuous line of asterisks is common practice and makes very noticeable borders. Single line comments should be avoided since they are easy to overlook; however, if necessary, they should also be set off with noticeable borders or blank lines to prevent code and comment confusion. Comments must be clear and concise with consideration for those who have to maintain or use the program. Having too few comments in a program is as bad as having too many. Use enough comments to make the program understandable.

```

/*****
* THIS SECTION CALCULATES THE PROBABILITY OF AN ARMORED PERSONNEL
* CARRIER GETTING STUCK IN MUD BY TESTING THE AMOUNT OF RAINFALL
*****/

```

```

    IF (RAINFALL EQ NONE) THEN
        APC = MOVING;

    ELSE IF (RAINFALL EQ ALOT) THEN
        APC = SLOW;

    ELSE IF (RAINFALL EQ FLOOD) THEN
        APC = STUCK;

```

CODE INDENTATION

The purpose of code indentation is to improve the readability and the logical structure of programs through a format which reflects the logic of the program.

- a. Each DATA and PROC statement should start in column 1.
- b. Each subsequent line should be indented evenly.
- c. Each DO group level should be indented, with DO statements starting on a new line for easy visibility.
- c. The END statement used to terminate a DO loop should be indented at the same level as the starting DO statement.
- d. Comments should not be indented to match corresponding code, and must precede the relative code.

```

DATA CONVTEMP;
  SET SITEDATA;

  /*** Convert Fahrenheit temps to celsius and output if year is 1970 or beyond ***/

  IF YEAR GE 1970 THEN
    DO;
      TEMPC = (5/9) * (TEMPF - 32);
      OUTPUT;
    END;
RUN;

```

VARIABLE NAMES

Without a standard set of variable names, code reusability means that each program must rename the other program or macro's variables in order to meet the naming convention of the calling program. It's much simpler to use a standard set of variables for all programs written in your organization. At AFCCC, we developed a SAS data dictionary for common weather items that has to be used in order for the programs to be released for production. Due to SAS's eight character restriction on variable names, this took a bit of creativity. AFCCC's standard variables are composed of a root variable with the option of a suffix. For example, ELEV is used for a generic elevation, while

ELEVFT indicates elevation in feet, and ELEVM indicates meters. These standards worked so well that our entire DB2 database naming convention was modified to meet the unit's SAS naming convention. Some of the prefixes that were created are:

TEMP - temperature CLD - cloud PRES - pressure

with suffixes of:

C - celsius L - low MB - millibars

giving variables of TEMPC, CLDL, and PRESMB.

Phil Mason, on the Internet SAS-L mailing list, states that SAS Australia recommends the following naming convention for SCL variables prefixes:

WIDGET

T_	Text entry / Text label / Graphic text
P_	Push button / Icon
S_	Slider / Scrollbar
X_	Checkbox
H_	Hotspot
B_	Block
G_	SAS/Graph output / Graphics Object
LB_	List box
RB_	Radio box
CB_	Container box
CF_	Critical success factor
XT_	Extended table
M_	Macro variable

NON-WIDGET

D_	Dataset ID
F_	File ID
L_	List ID
I_	Instance ID (of a class)
W_	Widget ID
U_	Utility (non-window variable)

CONCLUSION

Programming standards provide a means of ensuring that software written at your organization is of high quality. But in order for them to be effective, they must be followed by everyone. What has been presented here are the basics to get you going. Unfortunately, due to size constraints, this paper could only offer a brief overview of the software life cycle. For more information on the life cycle, and other aspects of programming standards, read some of the books listed in the reference section below.

REFERENCES

Freedman, Daniel P. *Handbook of Walkthroughs, Inspections, and Technical Reviews*. Dorset House Publishing, NY. 1990

Dunn, Robert. *Quality Assurance for Computer Software*. McGraw-Hill Book Company, NY. 1982

USAFETAC. *Communications-Computer Systems Automated Data System Standards and Procedures*. Instructions 33-102. Vol I, II, III. Scott AFB. 1995

SAS is a registered trademark of SAS Institute, Inc. in the USA and other countries.

CONTACTING THE AUTHOR

C. Michael Whitney, SSgt, USAF
AFCCC / SYS
Bldg. 859
Scott AFB, IL 62225-5116
(618) 256-5323