16 JULY 1990
CLAS NOTE 90-008

The Hitchhiker's Guide to the Galaxy: CLAS Software Manual (Rev 1.0)

Donald Joyce
Physics Division
Continuous Electron Beam Accelerator Facility
Newport News, VA 23606, USA
and
Larry Dennis
Department of Physics
Florida State University
Tallahassee, FL 32306, USA

Abstract: *This manual is intended to be the definitive reference on all matters relating to software development standards and rules for the CLAS detector software. In particular, this manual applies to the analysis software necessary to transform "raw events" to "physics" events. Although it's primary focus is software used in common among members of the CLAS collaboration, the manual is equally applicable to code being developed for individual usage.*

1. Software Overview

1.1 Introduction

This CLAS-Note is meant to be the definitive document on all matters relating to software development standards and rules for the CLAS detector software. It contains a list of the rules which must be adhered to by those writing software for the CLAS collaboration, a set of guidelines to aid in the production of correct, readable software, an outline of the online/offline analysis software and a brief description of the expected software development environment. These rules and guidelines are based on the experiences of those currently involved in CLAS software development and the experiences of other collaborations.

The purposes for these rules are to eliminate software errors from the CLAS software before they are inserted and to produce code which is easy to read and understand. One of the keys to the success of the CLAS collaboration will be its ability to produce, modify and maintain reliable software. This places an enormous burden on those writing the software. They must translate their ideas into correct, readable code and prepare for the day when their code must be modified quickly and correctly by someone who is not involved in its development.

1.2 Programming Languages

All code developed for the event analysis of CLAS data should be written in ANSI standard FORTRAN77. The only extensions to the ANSI standard allowed are those listed later in this document. On rare occasions, portions of code cannot be written to run efficiently or to run at all in FORTRAN77. These portions can be written in "C" or "C++". However, one cannot code in these languages based on whim or fancy. In general code other than FORTRAN77 will be rejected. Please discuss the problem with the software working group before launching into a "C" program development project.

1.3 Use of Software Packages

Coders for CLAS software are encouraged to incorporate existing documented and supported software packages. Although commercial software is not excluded, it must be made readily available to all members of the CLAS collaboration at minor expense. As a rule, this limits one to a choice of packages, such as those from CERN, FermiLab or other major laboratories. Given a choice between functionally equivalent routines or packages, software distributed by CERN should always be selected. This selection criterion also applies to personally developed code. On occasions, various supported and distributed routines need to be modified to satisfy particular requirements. Any changes to these routines other than *"bug fixes"*, must be accompanied by a name change of the routine, and appropriate common variables. Bug fixes must be fully documented according to the CLAS software documentation standard and reported to the program distributor.

On occasion, various useful packages exist for which there is no distribution nor maintenance responsibility. Neither is there an equivalent in the CERN software distribution. For such cases, the package must be rewritten, and documented to meet CLAS software standards, including naming conventions.

1.4 Documentation

There are two types of system level documentation which must be included with each program, module or include file. These two documentation types are stand-alone documentation and inline documentation. In addition, packages which are of general use, must have user level documentation.

All program units must be self-documenting. Following the SUBROUTINE or COMMON statement one identifies the author, and date as well as the purpose of the program unit. The person responsible for maintaining the unit should also be identified. History lines with author, date, and reason are added for each modification. Important variables in subroutines and all variables in commons must be defined in comments. A templet for producing such headers will be made available and should be filled in completely.

In addition, every step of a calculation should be documented with in-line comments. Sections are added to this manual as specific program units are written or modified.

Finally, a CLAS software history file is maintained. This history file describes each program unit, it's function, author, date, and reasons for modification.

1.5 File conventions

Subroutines, commons, and other files associated with a program unit are stored in a unique place. This means one subroutine, function or common per file. The name of the file is the name of the subroutine or common plus extensions. Commons are *included* into subroutines, and not coded directly into them. Parameters in commons are located separately and are included by the file containing the common blocks when needed. Subdirectories are named in a manner similar to that used for subroutines and commons (see the section on Naming Conventions later in this document).

1.6 Analysis Flow

Data analysis programs are typically designed with several distinct stages leading from data in *"raw event buffers"* to *"physics event buffers"* and finally to *"quantities of physical interest."* For the purpose of discussion, *"raw events"* refers to events as they are read from the detector analog converters. *"Physics events"* are then the same events, after they are analysed and reconstructed in terms of physical properties such as particle momentum, particle ID, event type, etc. Raw events

may of course come from several sources, including the data aquisition system or a Monte Carlo generator. The *"quantities of physical interest"* include such things as cross sections, polarizations, etc.

Figure 1 illustrates the typical flow of data analysis. First, data is *unpacked.* This means that the block of data associated with an event is redistributed in memory in a manner more amenable with analysis. For example, one might wish to store drift chamber data in a different set of commons than that used for the shower counter.

Secondly, one *derives* physical quantities such as time, energy, rough drift distances etc. for each counter element involved in an event. One refers to the detector *database* for *survey* and *calibration* constants as well as *conversion parameters.*

Thirdly, one groups the data into tracks, energy cluster depositions etc. with *pattern recognition and fitting* algorithms. This process may be repeated iteratively for portions of the detector with the second analysis step to better *derive* physical quantities such as drift distances.

Using the output from this stage, one then can *swim* particles through a Monte Carlo version of the detector, and compare the resulting simulated event with the actual event. Agreement between real and simulated events confirms both understanding of the detector properties and track fitting - particle ID algorithms. In addition it provides a very stringent monitor of detector performance. One may of course chose to bypass this step once the detector system is understood.

Following such a comparison, one typically sorts the data by *track-counter-particle* associations. This means that all information relating to a particle within an event is logically grouped together by appropriate cross referencing.

Events, can then be classified according to whatever scheme is appropriate. Finally, the data would be compressed and stored for later statistical analysis.

The flow and execution of the various steps in the analysis is typically controlled by a *job control* portion of the program.

1.7 Data Structures

Event analysis software for the CLAS detector is organized such that there is a clean separation of data into unique *commons*, each of which represents the state of knowledge about an event at a single point or *stage* in the analysis of an event. Job control parameters, survey and calibration constants, cable lists and patch panel maps are also segregated into commons of like types. Filling of commons is undertaken during the various stages of the analysis. For example, the track fitting stage of the analysis fills commons associated with track fitting. These commons are used as input to later stages of analysis such as track-counter cross referencing. The track-counter cross referencing stage would in turn fill it's own commons. Although commons may be used by other stages of analysis, one can not

fill or modify commons at random. Commons can only be filled or modified by that stage of the analysis for which the commons were created.

Please note that all common blocks are to be created for use within a specific analysis step or stage. If one needs access to information in *someone else's* common blocks, one cannot work directly with their common block variables. Any needed data should be obtained via access routines (written by the person originally responsible for the common blocks). This has the advantage of decoupling one coder's need for the information from another coder's need to be able to change the common blocks.

1.8 I/O AND COMMUNICATION

I/O with the analysis program, and among subroutines and functions is implemented with the above mentioned commons. Raw event commons are initially filled for data input and physics event commons are output on completion. Subroutine and function calls are implemented without argument lists wherever possible. For example, data are passed to subroutines in "input" commons and returned from subroutines with "output" commons. Of course general purpose routines which are not data specific cannot be implemented in this manner.

Each common has associated with it an output routine to dump the common contents to either a print file or intermediate data record. The print file would be used for debugging, while the intermediate data record is used to archive partially complete analysis.

1.9 NAMING CONVENTIONS

CLAS event analysis software is written such that the names of subroutines, commons, and variables can be associated with the stage of analysis, it's function, and the part of the detector with which it is associated. Names of subroutines and commons of the same type begin with the same set of characters. Names of variables stored in commons end with a set of characters identifying the common of which they are a member. Specific guidelines are given later.

One notes, however, the immediate need for exceptions. In particular, one anticipates that major code segments may be adopted from pre-existing programs without recoding. For such segments, the naming convention of the code will be maintained to assure compatability with documentation and future developments. However, this exception is limited to those codes distributed either commercially or code evidencing both distribution and support by a major facility. This basically means that pre-existing personal code must be rewritten to comply with various naming conventions.

2. Programming Rules

This chapter describes the programming rules for the coding language, naming conventions and documentation. Software which does not conform to these rules will not be accepted as part of the CLAS software system. Those writing software for the CLAS are encouraged to follow the guidelines presented in the next chapter as well. To the extent possible tools will be provided to make it easy to comply with these rules and guidelines.

2.1 FORTRAN77 Extensions

All code should be written in ANSI standard FORTRAN77. The only extensions allowed to ANSI standard FORTRAN77 are given in this document. A compliance verification tool will be made available to all collaboration members to check that a given module follows the ANSI FORTRAN77 conventions. The following extensions are allowed:

- Use of **DO, ENDO** construction.
- Use of **DO WHILE** construction.
- Names of variables, subroutines, functions and commons may be up to 32 characters long and may use the "_" (underscore) character.
- The use of "!" for the start of inline comments.
- Since all commons are kept in separate files, the use of the extension **INCLUDE** is actually required.

2.2 FORTRAN77 Restrictions

There are several programming constructs allowed by FORTRAN77 which are not easy to untangle. Those listed here are not allowed in code for the CLAS.

- Computed GOTO's should be replaced by IF(..) THEN, ELSE IF(..) THEN, END IF blocks.
- Arithmetic IF's should be replaced by IF(..) THEN, ELSE IF(..) THEN, END IF blocks.
- Always end do loops on a CONTINUE or END DO statement and do not end multiple do loops on the same CONTINUE line.

2.3 Variable Declarations

All variables must be explicitly typed. This means that each routine contains an **IMPLICIT NONE** statement. The following variable types are allowed:

- **CHARACTER**
- **LOGICAL**
- **INTEGER*2, INTEGER**
- **REAL**
- **DOUBLE PRECISION**
- **COMPLEX*8, COMPLEX*16**

2.4 Naming Conventions

Naming conventions are to be used for all subroutines, functions, common blocks, variables in common blocks, parameters and variables used in subroutine or function calls. Thus strictly local variables are not required to adhere to these standards, but important variables within any routine should adhere to them. The only exception to these conventions are routines from CERN library packages or other accepted commercial software packages.

Parameter and Common Block files should be clearly distinguished from the FORTRAN files into which they are included. Common Block files are identified with the extension "CMN" while Parameter files are identified with the extension "PAR". For example one might have a drift chamber calibration parameter file named DcCal_TIME_DISTANCE.PAR.

The previous chapter briefly outlined the flow of execution of an analysis program and identified specific stages in the analysis. The following naming convention for subroutines, functions, commons and variables are adopted to associate specific values, commons and routines with individual steps of the analysis and the portion of the detector package with which they are associated. To do this, two fields are needed at the start of each routine or common block name and all common block variables or variables in an argument list. The first field, consisting of two characters, is used to indicate the detector package with which this name is associated. The second field, consisting of three characters, is used to identify the analysis section. These two fields should be followed by an underscore "_" before continuing the name. As an example one might use the variable DcCal_TIME. (The case of the letters in the name is the suggested one.) The following descriptive strings have been identified and are reserved:

The list of character strings below are the currently accepted two character detector package strings:

- **BM** - beam monitoring devices

- **DC** - drift chambers
- **CC** - cerenkov counters
- **DA** - data acquisition
- **DB** - data base
- **EC** - shower counter
- **EL** - electronics
- **FU** - full detector (spans packages)
- **GB** - global variable or common
- **MG** - magnetic field
- **PT** - photon tagger
- **SC** - scintillators
- **SW** - software maintenance system
- **TG** - target system
- **TM** - tagger magnet
- **TR** - trigger
- **VD** - vertex detector

The above character strings precede the strings specifying specifies the "analysis section". The list of character strings below are the currently accepted three character analysis section strings:

- **CAL** - calibration constants.
- **CLS** - event classification.
- **CUT** - event cuts.
- **DBA** - data base access.
- **DRV** - derived detector hit information.
- **EVB** - raw event buffer.
- **EVU** - unpacked raw event data.
- **FIT** - track and cluster fitting.
- **IDX** - index and cross reference maps.
- **JCL** - job control.
- **PAT** - pattern recognition of tracks and clusters.
- **SIM** - simulated event data input to raw event buffer.
- **SRV** - detector survey parameters.
- **SWM** - particle swimming/tracing through detector.
- **TKC** - track-counter association.

- **DIG** - swim point digitization.
- **SLO** - slow control software.
- **MON** - detector monitoring software.
- **GRA** - graphics software.
- **GUS** - general utility software.

The above mentioned character strings would be followed by an "_" and the appropriate characters identifying the exact nature of the routine or common. Consultation with other programmers for other packages should occur to maintain similar names for similar operations or variables. Finally, the list of accepted identifiers will be updated as needed. Do not use a new identifier string without consulting with the CLAS software group.

The names chosen for routines, commons and variables in GEANT, LUND and PAW will be maintained as distributed by CERN. Software developed by the data acquisition group at CEBAF will also be used as it is delivered to the CLAS collaboration.

2.5 Documentation

There are two type of documentation required for all subroutines and functions. These are:

- **Inline documentation** - All routines must include the following information in comments following the program module statement:

(a) Name of the original author

(b) Date of the original module.

(c) Institution of the original author.

(d) Purpose of the module.

(e) Method used in this module

(f) Modification list which includes the name, date and purpose for each modification.

(g) Description of all variables in the argument list. Identify which of these variables are input, control and output variables.

(h) Comments describing all major sections of the routine. Under no circumstances will code with less than 1 comment per 10 lines of code be accepted. These comments should be complete *english* sentences which describe the routine's actions. They should not simply repeat obvious coding statements such as:

```
C       A = B + C*D - F**G.
```

- **Stand-alone Documentation** - A single common document file will be prepared for all program modules. This documentation will include the items listed above as part of the inline documentation and the following:

(a) A list of all include files used by the routine.

(b) The directory location of the source code.

(c) A library name if this module is part of a library.

(d) A list of all software libraries needed by this routine.

There are two types of documentation required for common blocks and include files. These are:

- **Inline documentation** - All include files containing common blocks or parameter statements must include the following information in comments following the last declaration statement in the file:

(a) Name of the original author

(b) Date of the original include file.

(c) Institution of the original author.

(d) General purpose of the include file.

(e) A modification list which includes the above information for each modification.

(f) Description of all parameters or of all variables in the include file.

- **Stand-alone Documentation** A document file will be prepared for all include files. This documentation will include the items listed above as part of the inline documentation and the following:

(a) A list of all modules which include this file.

(b) The directory location of the include file.

It is expected that these documents will be generated automatically and that utilities will be developed which will convert these documentation files into TEXfiles. However, any code which is not documented is considered useless until it has been documented and it will not be used as part of the CLAS software.

2.6 Code Checking Procedures

Figure 2 illustrates the expected method of enforcing these rules. Please do not try to find ways around them. The goals of this code checking procedure are to determine if a piece of code is relatively machine independent, if it follows the coding standards, if it does the operations it is designed to do and that it interacts with other portions of the online analysis code only as expected. It is not designed to require that the code follow a certain algorithm. Such decisions are made at other points in the code design.

3. Programming Guidelines

3.1 General Programming Practices

The following programming guidelines are offered as suggestions which will help make code readable. They are not included as rules because they cannot easily be checked, but programmers for the CLAS software are strongly encouraged to keep them in mind and follow them as closely as possible. Many of these guidelines have been taken directly from the CDF online system manual by Dave Quarrie, which we gratefully acknowledge.

(a) Remember that somebody else is very likely going to have to use your code during the lifetime of the CLAS. Code should not only perform its intended function, but it should be understandable. Avoid tricks unless absolutely necessary and explain why a particular approach has been taken.

(b) Provide plenty of comments. A good practice is to separate code into logical blocks and begin each block with a short description of its action. There should be about as many comment lines as code.

(c) Code for clarity and correctness rather than efficiency. The compiler is probably better at it than you are anyway and it is a waste of time to optimize an incorrect code.

(d) Remember that the cost of an error to the collaboration is very high in terms of lost time, data and money.

(e) Use meaningful names for variables. Avoid single character names - they might mean something to you now, but in three months time they won't . Their only role is as dummy indices in loops and output statements.

(f) Indent each nested block in order to highlight the start and end of each of them. This is a great improvement in legibility.

(g) Avoid implicit data-type conversions and mixed-mode arithmetic. Instead use FLOAT and INT datatype conversion operators explicitly. This has no effect on code efficiency, might help you to avoid inadvertent rounding or truncation errors and will highlight the datatype conversions to anyone attempting to follow the code.

(h) Avoid dreadfully complicated "IF" constructs. It is better to use several nested "IF" constructs rather than one complicated compound one. If it takes you more than two minutes to get the logic right (you hope) the first time, it will take you many more to decipher it the next time you look at it.

(i) Avoid multiple RETURN's from subroutines and functions. Indicate success or failure via function values rather than multiple exits. Aim to have a single "RETURN" statement for all subroutines and functions rather than many of them scattered within the code.

(j) Attempt to write linear code; i.e. code that starts at the first executable statement and flows smoothly down to a final "RETURN" or "STOP" statement.

(k) Minimize the use of statement labels. Ideally they should only be used for FORTRAN77 "DO" loops and for "FORMAT" statements. FORTRAN does not have all the constructs necessary to avoid the use of "GOTO" statements entirely, but attempt to minimize their use. They should only be necessary to exit prematurely from within a "DO" loop and to avoid multiple "STOP" or "RETURN" statements.

(l) Use parameters rather than numbers. This is something where common sense must be applied. Obviously using ZERO instead of 0 does not make the code clearer and involves much more typing; but using PI instead of 3.14159 both clarifies the code and avoids potential mis-typing each time it is used. For example a set of standard constants might be placed in the include file GbGus_CONSTANTS.PAR.

(m) There should be only one "STOP" statement in a program, being at the end of the main program module. Never ever have a "STOP" in a subroutine or function - indicate errors via function values and unwind back to the main program.

(n) Minimize the use of numbered statements. Those numbers which are used should be in numerical order from top to bottom of each module.

3.2 Use of Parameters

Whenever possible use PARAMETERS rather than numbers in your code. Each time you type in 3.14159 increases the probability that you'll mis-type it somewhere and consequently have severe problems when you start debugging your program. Even worse, such mis-typing might result in such subtle effects that it is not noticed in the course of normal debugging, but only manifests itself much later when the code has been used for production running for some considerable period of time.

If you have defined an array where each element contains some attribute, don't refer to the attribute by the element number, but by a PARAMETER having some mnemonic significance. The following code fragments illustrate this, within a CLAS FORTRAN environment.

```
      SUBROUTINE EcCls_find_electron
C----------------------------------------------------------------------
C-
C-    Purpose and Methods :  This subroutine is used to determine if
C-                           one of the particles detected in this
C-                           event was identified as an electron.
C-                           If it is, then the event selection
C-                           variable EcCls_electron_found is set
C-                           to true.  Otherwise it is set to false.
C-
C-    Inputs     :     none
C-    Outputs    :     none
C-    Controls   :     none
C-
C-    Created    :     11-JUN-1990 J. Firstone, Cassiopeia State
C-
C-    Accepted   :     15-JUN-1990 So.  Dumb, Andromeda Reform School
C-
C-   Revisions   :     15-JUN-1990 Dr.  Who ......
C-                     Reason:  code missed low energy electrons near
C-                              45 degrees
C-
C----------------------------------------------------------------------
      IMPLICIT NONE
C----------------------------------------------------------------------
C
C ****     Define the number of properties allowed.
C
      INTEGER EcCls_MAX_PROPERTIES
      PARAMETER( EcCls_MAX_PROPERTIES = 3 )
      REAL EcCls_PARTICLE_PROPERTIES( EcCls_MAX_PROPERTIES )
      REAL MASS
C
C ****    Declare the particle property displacements within the
C ****    array.
C
      INTEGER EcCls_PARTICLE_MASS
      INTEGER EcCls_PARTICLE_TYPE
      INTEGER EcCls_PARTICLE_CHARGE
      PARAMETER( EcCls_PARTICLE_MASS   = 1 )
      PARAMETER( EcCls_PARTICLE_TYPE   = 2 )
      PARAMETER( EcCls_PARTICLE_CHARGE = 3 )
C
```

```
C ****    Define the parameters for the particle types.
C
      INTEGER EcCls_PHOTON
      PARAMETER( EcCls_PHOTON = 0 )
      INTEGER EcCls_ELECTRON
      PARAMETER( EcCls_ELECTRON = 1 )
      INTEGER EcCls_MUON
      PARAMETER( EcCls_MUON = 2 )
... ... ...
C
C ****    Get the particle mass.
C
      MASS = EcCls_PARTICLE_PROPERTIES( EcCls_PARTICLE_MASS )
... ... ...
      RETURN
      END
```

The above discussion is especially true when describing something which can change often. Through the use of parameters major hardware changes, or the migration from prototypes to production models may require nothing more than editting a parameter file, rather than laboriously searching through code for all possible occurences of the affected array accesses and numerical constants. A *great deal* of thought is required to do this well, but one should attempt to identify a set of fundamental parameters from which all other parameters can be derived and to use those parameters *exclusively* to refer to dimensions, geometry, number of detectors, order of array elements, etc. The time required to find this fundamental set of parameters will be paid back the first time one of them changes.

3.3 INCLUDE FILES

A frequent occurrence is that a set of PARAMETERS or VARIABLES is accessed from many program modules. Rather than retyping them into every module, an INCLUDE file should be used. In the previous example, specifying the PARAMETERS describing the array in an INCLUDE file would allow any program module to access the appropriate property in an identical manner. The accepted format for INCLUDE files is:

INCLUDE 'logical_name:filename.PAR'

where the file type PAR is used to indicate that this file is an INCLUDE parameter file. A file type of CMN would connote an INCLUDE common block file. Note the use of a logical name rather than a directory name. This is required so that code (and include files) may be developed in the user's own area and then be transported into a program library for general use within the online system. Thus:

INCLUDE 'CLAS_ROOT:filename.par'

is acceptable, but

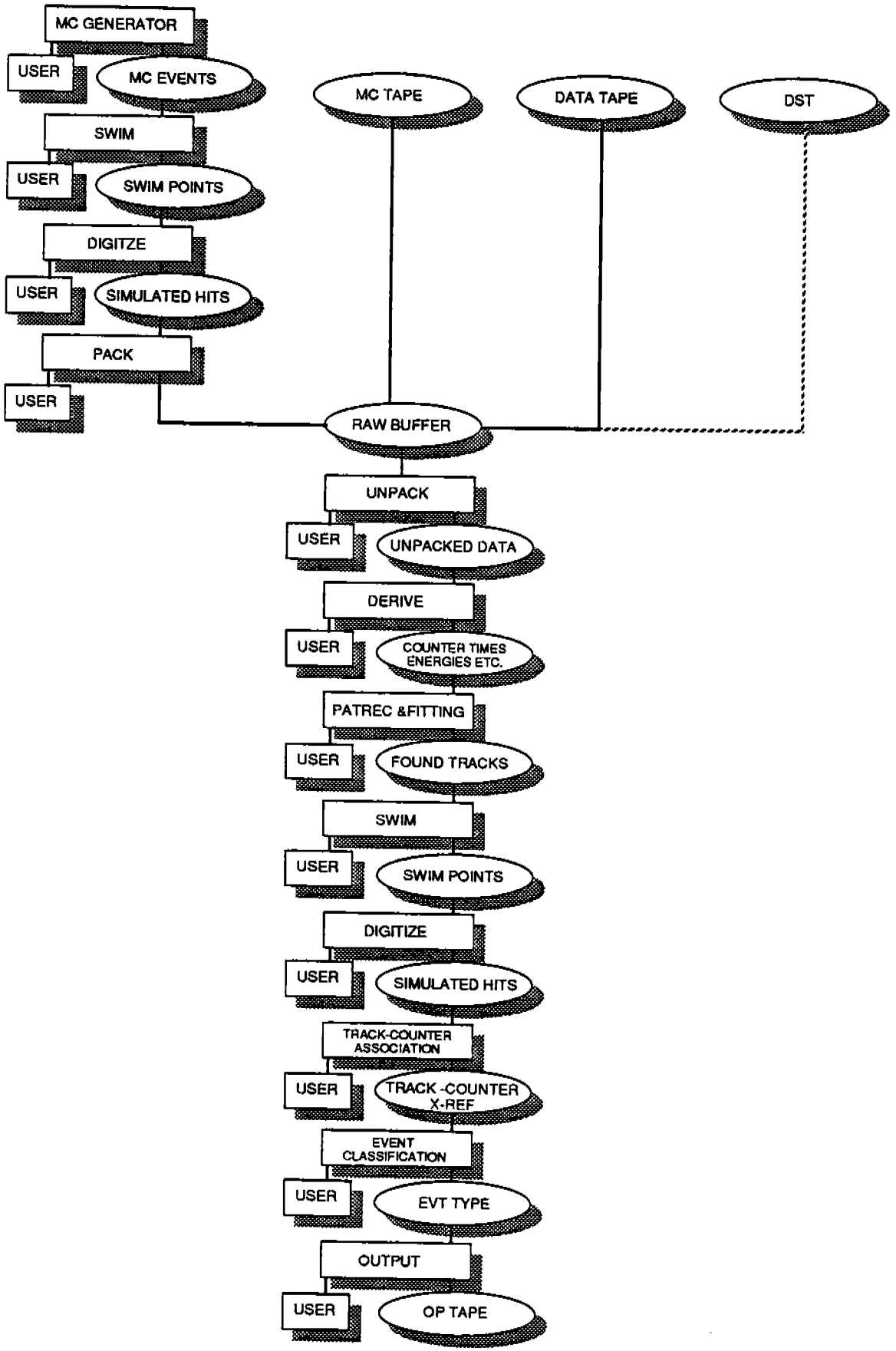
INCLUDE '[johndoe.programs]filename.par'

is not acceptable.

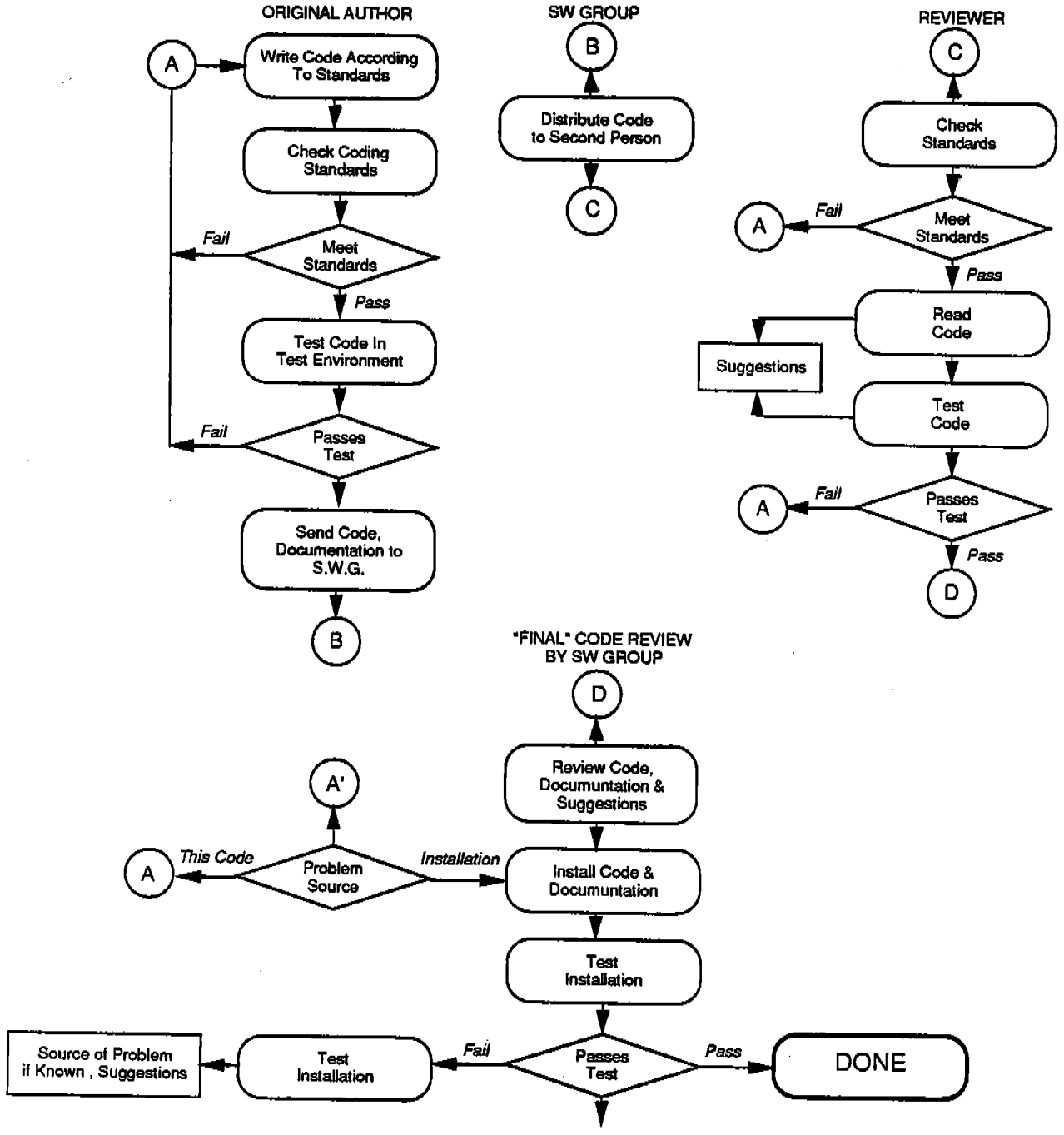
Note that the use of logical variables is not supported on many machines. A compile-time preprocessor will be run against individual code to translate the logical variables such that they match the appropriate format required on the target computer. It is expected that the filenames on none-VMS machines will match the names on the VMS machines as closely as possible.

3.4 USE OF MAKE, MMS, CMS, AND EDFOR

There are several levels of code maintenance schemes which could be used. The current choice is to use the VAXSET products CMS, MMS, SCA and the D0 tools, EDFOR (a TPU based editor), D0CHECK (a code checker) and D0FLAVOR (a code customizer). These will be augmented by the SCCS and MAKE utilities (under UNIX). This will be done because the initial software code management for the online and offline code will be done primarily on VMS based computers. It may eventually migrate to UNIX based computers.



DENNIS 002 H08



DENNIS 001 HDB