NOS SYMPL CODING STANDARD

TABLE OF CONTENTS

1.0 Introduction

The purpose of this standard is to provide a meaningful set of practices which will lead to "good", consistent, maintainable, organized and optimized SYMPL code. This document used the SYMPL Coding Standards DAP (DCS S1831), the NOS COMPASS Programming Standard, and the SYMPL Coding Standards for the SYMPL project in SVL as guidelines.

This standard is in addition to the NOS COMPASS Programming Standard. The procedures established in the COMPASS standard which are not unique to the COMPASS language (i.e. General Requirements, Code Transmittal Rules, and Dayfile Messages) are to be adhered to for SYMPL programming also.

Where the word "must" appears in this standard, deviations will not be approved. Where the word "should" appears, reviewers may allow a deviation if the analyst can present convincing reasons for the deviation.

2.0 Coding Standards

2.1 General

All declarations pertaining to a PROC or FUNC should use the following grouping

Formal Parameters
XREFs
DEFs
STATUS names
COMDECKs
ITEMs
BASED ARRAYs
ARRAYs
SWITCHes
Other

All declarations or calls to COMDECKs should be in alphabetic order.

Each declaration must start on a separate line and must be accompanied by a comment describing its function.

Each executable statement must start on a separate line.

Each BEGIN and END must be on a separate line.

A declaration which is a one-bit field should be Boolean.

Self modifying code must not be used.

All labels begin in column one. Labels must appear on lines by themselves except for embedded comments. All label names must be unique within a PROC/FUNC.

TEST must never be used without explicitly stating the induction variable it is testing.

Define CONTROL DISJOINT and CONTROL INERT in a COMDECK. Use CONTROL OVERLAP and CONTROL REACTIVE to define the exceptions.

Where numeric constants are established via DEFs or STATUS lists, the assumed numeric values should not appear in the coding documentation.

Items I, J and K should be reserved as simple loop or control variables.

2.0 Coding Standards

2.1 General

The code must not make assumptions about the octal representation of characters. This representation varies between the various NOS character sets.

Machine independent instructions when available should be used in preference to dependent structures.

2.2 Parameters

Use call-by-value parameters whenever possible. Only use call-by-address when the parameter is modified within the procedure and the new value of the parameters is returned to the calling program.

Reuse actual parameter lists whenever possible. If the parameters are used for a number of calls, use the same order of parameters for more efficient coding.

Formal parameters must be declared within the PROC/FUNC rather than in a common deck. They can be ordered alphabetically or according to the calling sequence.

An array item must not be used as a parameter where a new value of the parameter is returned, since this feature is not supported in SYMPL.

2.3 XREF

Declaration of external procedure names are to be done in the following format. The referenced PROC/FUNC names are to be in alphabetic sequence.

Example:

```
#
****  PROC Y - XREF LIST BEGIN.
#

      XREF
        BEGIN
        PROC APPLE;                   # PARES APPLE #
        PROC BANANA;                  # PEELS BANANA #
        PROC ORANGE;                  # SQUEEZES ORANGE #
        END

#
****  PROC Y - XREF LIST END.
#
```

2.4 DEF

Use DEF to provide symbolic constants for numeric constants for ease in finding, identifying and modifying parameters.

A DEF must not be used to rename a variable.

A DEF must not be used to redefine a function call, a reserved word, or an operation unless it is used consistently throughout the system to improve clarity. Otherwise, this may tend to obscure the actual code. All DEFs which redefine the code or make it a conditional compilation will be placed in a COMDECK.

The DEF format for a full word octal constant is in 4-digit parcels. For example:

```
DEF ERRMASK #O"0037 7740 0505 0000 7777"#;  # ERROR BIT MASK #
```

2.5 STATUS

Status lists should contain no unused positions. Any unused positions must be filled with a dummy argument and have a # RESERVED # or # NOT USED # comment. It may be better to use DEFs if there are many unused positions or any of the elements are expected to change.

2.6 COMDECK

Executable code should not be placed in a COMDECK.

The declarations for a data structure must be wholly contained within a single common deck. Where two or more data structures are interdependent, the declarations for the interdependent structures must be in the same common deck.

Logically associated data items and structures should be declared in one COMDECK unless they are only to be used by one module where they may be declared locally.

One or more COMDECKs must contain all declarations affecting table size which could be changed with the system. This is to facilitate maintenance.

Common decks must not be listed.

A PRGM, PROC or FUNC should only call the common decks that it references.

Every common deck must have an overview description of what it does. The following format is to be used. The list control statements begin in column 48.

```
#   deck name - description.                   #
                             CONTROL NOLIST;
                             CONTROL IFEQ LISTCON,1;
                             CONTROL LIST;
                             CONTROL FI;

#
*** deck name - description.
*
*   (purpose) (several lines can be used)
#

                             CONTROL LIST;
```

2.7 Non-array Items

The items, the variable names, the types, the presets and the embedded comments should each be vertically aligned. Leave room for ten character variable names and leave room for character counts on character type items for ease of future maintenance.

Variables should be declared alphabetically.

2.8 Arrays

Arrays used by more than one PROC must be defined in COMDECKs.

Usage of items from an array must always be subscripted. It is confusing to default subscripts.

Item declarations must be in ascending order (i.e. word 0 bit 0 to word n bit n). If overlapping declarations are used, then the item which spans other items must be first.

Array indices should start with zero.

The array name, bounds and the allocation/size must be separated by blanks (e.g. ARRAY EXMAPLE [0:10] P(2);).

Items within an array are aligned with the begin for ease of reading. Each item must be documented.

The item names, type/positions, preset values, and embedded comments should each be vertically aligned. Leave room for ten character item names and for two digit "ep", "fbit", and "size" fields and use at least two spaces after the semicolon to ease future maintenance.

2.9 FOR Loops (Fast or Slow)

FOR loops are of two types. In the slow FOR loop, the object code has a direct correspondence with the SYMPL statements. This is not the case with fast loops. A fast-for-loop is optimized by pre-evaluating the STEP and UNTIL/WHILE elements. At least one cycle of the loop is executed.

Fast FOR loops and slow FOR loops must be used. A simple FOR statement must not be used. For easier readability and programming, use DEF statements to set up FASTFOR or SLOWFOR instead of the CONTROL FASTLOOP or CONTROL SLOWLOOP. These DEF statements should be placed in a COMDECK.

```
    DEF FASTFOR    #CONTROL FASTLOOP; FOR#
    DEF SLOWFOR    #CONTROL SLOWLOOP; FOR#
```

For better optimization consider using STEP/WHILE as an alternative to STEP/UNTIL.

The induction variable must not be changed during the loop or by a FUNC called while evaluating the STEP/UNTIL/WHILE part.

The exit from a loop should be through an UNTIL/WHILE or a return statement. The entry into a loop must not be in the middle of the loop.

The executable statement(s) after the DO part of a FOR loop must be enclosed in a BEGIN/END pair.

2.10 GOTOs and SWITCHes (Case Statement)

GOTO should be employed only if the resulting source code is demonstrably superior in performance, clarity, maintainability, or extendibility. In spite of structured programming, GOTOs may make the code more efficient if employed properly. GOTOs may make it difficult to follow logic. Jumps into FOR loops must not be used. Jumps into code within a THEN or ELSE should not be used. Jumps backwards in the code should not be used.

A GOTO statement specifying a subscripted switch list may be used to simulate a case statement. Each case should end with a GOTO branching to a common exit, a RETURN statement, or an ABORT call.

Simulated case statements may use a multiplicity of labels for exits, provided that the selection of exit points is done to achieve consolidation of similar sequences of code, and that all such labels are grouped together. See the Examples section for an example of a simulated case statement.

2.11 IF

The THEN and ELSE part of an IF statement must always use a BEGIN/END pair. If embedded comments are needed to describe the condition, they should be placed with either the THEN/ELSE or the associated BEGIN/END pair rather than on the IF. A stand alone comment following the THEN or ELSE may be used instead if embedded comments would be too long or would restrict the readability of the code.

Related IF statements should not be nested more than 3 deep. A simulated case statement may be used.

Compound conditionals on an IF statement should be ordered such that the first condition is the one which will most likely terminate the condition evaluation.

2.12 Bead

Avoid using bead functions unless necessary. Instead, the use of an array with partial-word items is preferred. Bead functions are difficult to update in a program if the data item that is beaded is ever changed. If used, do not cross-type (bit functions should be used only on numeric data, byte functions only on characters).

Bead functions may be used to simulate data definition features not currently implemented with SYMPL such as repeating groups within a word.

2.13 PROCs, FUNCs, and PRGMs

XDEFs, alternate entry points, and internal PROCs should not be used. they are hard to locate in the program and will make debugging and modification more difficult.

PROCs and FUNCs must have a fixed (not variable) number of parameters.

The F option on the SYMPL command must not be used. Instead, use CONTROL FTN in the source when needed.

3.0 Naming Conventions

All declarations and PROC/FUNC names should be descriptive.

Routines may use simple local variables named TMP1, TMP2, etc. However, such names can be used only for multi-purpose items. Items with a specific computational purpose should have a meaningful name.

All external identifiers (PRGM, PROC, FUNC names) must be 7 or less characters. The loader truncates a name to 7 characters.

All internal identifiers (declarations, arrays, status list names) must be 10 or less characters. A $ may be used as another letter in the alphabet. However, $ is invalid in the deck name because of MODIFY.

All array items should be prefixed by the first 3 or 4 characters of the array name. The last 6 or 7 characters of the array item are the descriptive name.

All related DEFs should use the same prefix.

All COMDECK names should be 7 characters in length and should be in the following form

```
          COMxaaa

               where
                    aaa = Symbolic name of COMDECK
                    x = One of the COMDECK indicators:
                        A = COMDECKs used by more than one of the
                            E, U, or Z SYMPL groups
                        B = Data manager
                        C = CPU code
                        D = Display driver code
                        E = EXEC portion of MSS (SYMPL)
                        F = Full screen editor (FSE)
                        I = Initialization
                        K = Transaction subsystem
                        M = Mass storage error equivalents
                        P = PP code
                        S = Subsystem text symbols, constants
                        T = Tables
                        U = Utilities (SYMPL)
                        Z = Driver portion of MSS (SYMPL)
```

4.0 Code Readability

4.1 Format of Statements

All declarations must begin in column 7 and be finished before column 72. Column 72 must be blank to separate SYMPL code and comments from MODIFY sequence numbers. Each line of indentation is two spaces.

Each BEGIN/END is on a separate line. The first BEGIN is in column 7. Subsequent BEGINs are each indented two spaces. Code following the BEGIN, up to and including the next END, has the same indentation as the BEGIN unless exempted by some other rule (i.e. labels are in column 1). The END statement reduces the following indentation by two spaces. Any BEGIN/END pair that brackets more than ten statements should have matching embedded comments on the BEGIN and END. Redundant BEGIN/END pairs should not be used to highlight module structure. This function is better accomplished with stand alone comments.

Each THEN/ELSE/DO is on a separate line and is placed directly beneath the IF or FOR portion of the statement.

A statement which overflows the line must indent 2 spaces from the original statement.

Compound conditionals in an IF statement must be separated at the OR/AND if the entire statement does not fit on a single line. If the statement needs to be separated because of its length or at the programmer-s option, then the AND/OR plus its condition needs a separate line and is indented two spaces.

Examples

```
IF C              IF B            IF B OR C OR D
  OR (A AND B)      OR C          THEN
THEN                OR D            BEGIN
  BEGIN           THEN              .
  .                 BEGIN           .
  .                 .               .
  .                 .               .
  END               END             END
```

The format of the FOR statement follows the IF. If the entire statement will not fit on a single line, then the statement must be separated into two lines and indented two spaces.

```
   FASTFOR I=1 STEP 1
     UNTIL 7
   DO
     BEGIN
     .
     .
     .
     END
```

4.2 Column 1

The following items must begin in column 1:

Labels
PRGM/PROC/FUNC statements
Single line comments
Stand alone comments

4.3 Blank Lines

A blank line must be used in the following cases:

As the first line in each common deck
Between all declaration groupings
Before and after every stand-alone comment
Before and after all groups of conditional code
    (except COMDECK list control)
After every END statement
Before every label (or sequence of labels)

Blank lines (in addition to those required) may be used to improve the readability of the code.

4.4 Page Ejects

A page eject must be used as a separator between the declaration groups and the body of code.

If the declaration groups and the body of code will fit on a single page, five blank lines may be used rather than a page eject.

5.0 Documentation Standards

All documentation must conform to the NOS operating system requirements. This includes rules concerning complete sentences, capitalization, punctuation, abreviations, etc. All stand-alone comments are complete English sentences with correct punctuation, ending with a period.

5.1 Comment Formats and Types

Comments can appear in three different formats: stand alone, single line and embedded. Stand alone comments have four types determined by the number of asterisks on the initial line of a sequence of lines with asterisks in column 1. These four types are recognized by the DOCMENT utility and cause some comments (or code) to be included in DOCMENT output depending on DOCMENT run time parameters.

5.1.1 Embedded Comments

Embedded comments appear on the same line following a declaration or executable statement. The left delimiter must be preceded by at least two spaces and followed by only one space. At least one space follows the comment text before the right delimiter. At least one space must follow the right delimiter. Column positioning rules for the left delimiter are given in the section "Documentation with Embedded Comments".

5.1.2 Single Line Comments

These comments have a left comment delimiter in column 1, the text starting in column 3 for title lines or in column 7 for common deck headers, and a right comment delimiter proceeded by at least one space all on a single line. This comment form is used in the following cases:

-- Title lines

-- Common deck headers

5.1.3 Stand Alone Comments

These comments consist of at least 5 lines with the first and last being blank lines, the second and next to last having (only) a comment delimiter in column 1 with the comment body starting with line 3. Each line of the comment body has an asterisk in column 1 with blanks normally found in columns 2-6.

The initial line of the comment body (line 3) may have 1, 2, 3 or 4 asterisks starting in column 1 depending on the type of output desired from the DOCMENT utility.

5.1.3.1 Brackets (****)

A pair of stand alone comments of this form causes DOCMENT to copy the comment body starting with the opening bracket, and all subsequent code until the closing bracket. This is required for XREF declarations. An example is indicated with the XREF description. It may also be employed for other declarations or code which should be included on a DOCMENT run.

The comment body consists of asterisks in columns 1-4 with text on the rest of the first line. The comment text should clearly indicate which is the opening bracket and which is the closing bracket.

5.1.3.2 External Comments (***)

A comment body which is to be included in any DOCMENT run (external or internal) has 3 asterisks in columns 1-3 of the first line of the comment body. The 3 asterisk form is generally used to explain the interface to a SYMPL PRGM. It is also used in the header documentation for common decks.

5.1.3.3 Internal Comments (**)

A comment body which is to be included in a DOCMENT run selecting internal documentation in addition to external documentation has asterisks in columns 1 and 2 of the first line of the comment body. This is generally used to describe the interface for each PROC/FUNC. It may also be used to describe other important information about a PROC/FUNC/PRGM.

5.1.3.4 Module Comments (*)

A comment body which is not to be included in a DOCMENT run simply has 1 asterisk on the first line of the comment body. This type of stand alone comment is generally used to document design information which helps one maintain or code review a module.

This type of comment can present design information for the entire PROC/FUNC, or for a sequence of code. It should answer the question: "how does this PROC/FUNC code segment work?"

5.2 Program Level Documentation

Every PRGM must have an overview describing what it does and external documentation describing how it is used. The overview documentation is very general. A description of the fields is in the NOS coding standards.

```
#
***   (heading)
*
*     (purpose)
*
*     (command format)
*
*     PRGM program name.
*
*     ENTRY.       .....
*
*     EXIT.        .....
*
*     MESSAGES.    .....
*
*     NOTES.       .....
*
*     COPYRIGHT CONTROL DATA CORPORATION, 1983.
#
```

In addition, a PRGM may have internal and module comments as appropriate.

5.3 Documentation of PROCs and FUNCs

Every PROC/FUNC needs an internal documentation section. It should answer the question: "how is this PROC/FUNC used?". The description of the different fields is in the NOS Coding Standards.

```
#
**      (heading)
*
*       (purpose)
*
*       (PROC or FUNC statement with semicolon omitted)
*
*       ENTRY       .....
*
*       EXIT        .....
*
*       MESSAGES    .....
*
*       NOTES       .....
*
#
```

If a PROC or FUNC references a based array whose pointer is in a common block, and the PROC or FUNC assumes that the pointer for that array is set before the PROC or FUNC is called, the entry condition comments should state that assumption.

In addition, a PROC/FUNC may have additional internal comments and module comments as appropriate.

Where a higher level of documentation is needed for a related group of PROCs an extra PROC should be added to contain the unifying documentation.

5.4 Documentation with Embedded Comments

Embedded comments are of two documentation forms (i.e. data declaraction or action code). This is the only type of a comment that need not be a complete sentence. This type of comment should not be continued onto another line. If absolutely necessary, the comment may be continued on the following line. In this case the second line must not contain code.

```
      THEN                                   # comment which is too long
                                               continuation of commment #
```

5.4.1 Data Declaration Embedded Comments

Every array, item, status item, DEF and XREF item must be documented with appropriate information. Each declaration should appear on a separate line accompanied by embedded comments describing its function (optionally, if this is an important array, it may be bracketed by comment lines with asterisks in columns 1 through 4 so that DOCMENT will process it.

Presets should be commented individually to reflect the function of the preset.

The left delimiter of the embedded comment should be in column 38 unless the statement extends beyond column 35, in which case the delimiter is placed at least two spaces to the right of the statement.

5.4.2 Action Code Embedded Comments

For BEGIN and END statements, the embedded comments are placed two spaces to the right of the statement. For other statements the embedded comments begin in column 38 unless the statement extends beyond column 35 in which case the delimiter is placed at least two spaces to the right of the statement.

5.5 General Documentation for PROCs, PRGMs or FUNCs

Each PRGM, PROC, FUNC statement must have a corresponding END statement followed by the PRGM, PROC, FUNC name as a comment on that same line. SYMPL comments containing COMPASS-like title pseudo-ops must appear as the second line in a SYMPL PRGM, PROC or FUNC.

```
PRGM OK;
# TITLE OK - description of PRGM OK.                                   #

      BEGIN  # OK #
      .
      .
      .
      END  # OK #
```

6.0 Examples

6.1 COMDECK Examples

```
1     7                              38        48                     71
+-----+------------------------------+---------+----------------------+
COMASPC
COMMON

#     COMASPC - STEP POINT CONTROL.                                   #
                                               CONTROL NOLIST;
                                               CONTROL IFEQ LISTCON,1;
                                               CONTROL LIST;
                                               CONTROL FI;
      BEGIN  # COMASPC #

#
***   COMASPC - STEP POINT CONTROL.
*
*     *COMASPC* CONTAINS DECLARATIONS USED FOR CONTROL OF STEP MODE.
#


      DEF STEPCNT    #4#;            # NUMBER OF STEP POINTS - 1 #
      DEF STEPPNT    (I) #B<(I),1>STEPMASK#;  # STEP POINT #

      STATUS STEPVAL                 # STEP POINT VALUES #
        S1,                          # STAGING STEP POINT 1 #
        S2,                          # STAGING STEP POINT 2 #
        S3,                          # STAGING STEP POINT 3 #
        D1,                          # DESTAGING STEP POINT 1 #
        D2;                          # DESTAGING STEP POINT 2 #

      COMMOM ASPCCOM;

        BEGIN  # ASPCCOM #

        ITEM HPMASK     U;           # HALTED PROCESS MASK #
        ITEM STEPMASK   U;           # STEP POINT MASK #

        ARRAY HPT [0:STEPCNT] P(1);  # HALTED PROCESS TABLE #
          BEGIN
          ITEM HPT$LINK   U(00,42,18);  # HALTED PROCESS CHAIN LINK #
          END

        END  # ASPCCOM #

      END  # COMASPC #

                                            CONTROL LIST;
```

6.2 PROC Example

```
1     7                              38        48                    71
+-----+------------------------------+---------+----------------------+
PROC PSFIN((NDVALUE),(SPVALUE));
# TITLE PSFIN - INITIALIZES THE CONFIGURARTION.                       #

      BEGIN  # PSFIN #

#
**    PSFIN - INITIALIZES THE CONFIGURATION.
*
*     *PSFIN* INITIALIZES THE CONFIGURATION OF A FAMILY OF
*     DEVICES.
*
*     PROC PSFIN((NDVALUE),(SPVALUE))
*
*     ENTRY   (NDVALUE) = NUMBER OF DEVICES IN A FAMILY.
*             (SPVALUE) = SPACE ASSIGNED TO EACH DEVICE.
*             ARRAY HEADER = PSEUDO PFC.
*
*     EXIT    CONFIGURATION IS INITIALIZED.
*
*     NOTES   THE SPECIFIED VALUES ARE PLACED IN THE HEADER.
#

      ITEM NDVALUE    U;             # NUMBER OF DEVICES #
      ITEM SPVALUE    U;             # SPACE AVAILABLE PER DEVICE #

#
****  PROC PSFIN - XREF LIST BEGIN.
#

      XREF
        BEGIN
        PROC PSLOCK;                 # INTERLOCKS THE PSEUDO PFC #
        PROC PSUNLCK;                # RETURNS THE PSEUDO PFC #
        END

#
****  PROC PSFIN - XREF LIST END.
#
```

6.0 Examples
6.2 PROC Example

```
      DEF OFFSET  #4#;               # DEVICE ENTRY OFFSET IN PFC #

      DEF LISTCON #0#;               # DO NOT LIST COMDECKS #
*CALL COMAMSS
*CALL COMZHED

      ITEM I         I;              # LOOP VARIABLE #
      ITEM NUM       U;              # CALCULATED NUMBER #
                                               CONTROL EJECT;

      PSLOCK(HEADER);

#
*     SET VALUES IN THE HEADER.
#

      HEAD$ND[0] = NDVALUE;
      HEAD$SPDEV[0] =  SPVALUE;
      NUM = NDVALUE * SPVALUE;
      HEAD$SPFAM[0] = NUM;
      HEAD$SPAVF[0] = NUM;
      SLOWFOR I = 1 STEP 1 UNTIL NDVALUE
      DO                             # SET SPACE AVAILALBE #
        BEGIN
        HEAD$XX[I + OFFSET] = SPVALUE;
        END

      PSUNLCK(HEADER);
      RETURN;
      END  # PSFIN #

      TERM
```

6.3 Status List/Status Switch Example

```
      STATUS ERSTAT                   # ERROR STATUS #
        ERRORNO,                      # NO ERROR #
        ERRORFE,                      # FILE ALREADY EXISTS #
        ERRORFN,                      # FILE NOT FOUND #
        ERRORNW,                      # UNABLE TO WRITE PFC #
        ;                             # END OF *ERSTAT* #

      ITEM FLAG S:ERSTAT;             # ERROR CONDITION #

      SWITCH ERRCASE:ERSTAT           # ERROR LIST #
                  OK:ERRORNO,         # NO ERROR #
            PFEXISTS:ERRORFE,         # FILE ALREADY EXISTS #
             NOENTRY:ERRORFN,         # FILE ONT FOUND #
             WRITERR:ERRORNW;         # UNABLE TO WRITE PFC #
```

A status list may also be defined with an upper limit entry put at the end of the list. This upper limit can be used in the code to test that a variable is within its defined range. In this style the upper limit entry is terminated with a semi-colon on the same line.

Example:

```
      STATUS ERSTAT                   # ERROR STATUS #
        ERRORNO,                      # NO ERROR #
        ERRORFE,                      # FILE ALREADY EXISTS #
        ERRORFN,                      # FILE NOT FOUND #
        ERRORNW,                      # UNABLE TO WRITE PFC #
        ERROREND;                     # END OF *ERSTAT* #
```

6.0 Examples
6.3 Status List/Status Switch Example

```
#
*     PROCESS THE ERROR RESPONSE.
#

      GOTO ERRCASE[FLAG];

#
*     stand alone comment here or an embedded comment on the label.
#

PFEXISTS:                                   # embedded comment #
      -------
      -------
      GOTO ENDCASE;

NOENTRY:                                    # embedded comment #
      -------
      -------
      GOTO ENDCASE;

WRITERR:                                    # embedded comment #
      -------
      -------
      GOTO ENDCASE;

OK:                                         # embedded comment #
      ----------
      GOTO ENDCASE;

ENDCASE:
      -------

#
*     PROCESS THE ERROR RESPONSE.
#
```

A1.0 Addendum for SMF Project

This addendum describes changes to the NOS SYMPL coding standard for the Screen Management Facility (SMF) project. Certain parts of this change in the standard shall be relevent only to the Full Screen Editor and not to the screen formatter.

1. Structural changes

a. Nested procedures/functions are allowable under the following conditions. The terminology used here shall be "compilation unit" for an outermost PRGM/PROC/FUNC, since that is the scope of the map and cross-reference in the listing.

Procedures and functions may be nested. A compilation unit may contain XDEF-ed internal routines provided that a PROC/FUNC compilation unit is never called via the main entry point. Any routine may contain internal routines which are not XDEF-ed. That is, nesting of XDEF-ed PROCs is only allowed one level deep.

The second level of nesting is used only for routines which perform an algorithm not expected to be of value outside of the parent routine. Second level nested routines should be very simple in their logical structure. The same principles will apply for deeper level routines.

Non-XDEF internal procedures must have the same header documentation as any external procedure.

b. External symbols may be more than 7 characters long. The programmer is responsible to assure uniqueness within the first 7 characters. These oversize external names, while permissible, are discouraged and should be used only when the programmer cannot reduce the routine name to a 7 character name with sufficient clarity.

c. COMPASS subroutines are allowed for optimization of tight loops. Such routines should be designed to contain a minimum of decision-making logic.

A1.0 Addendum for SMF Project

d. Each compilation unit in the editor shall call COMAFSE as its first common deck. This deck contains symbol and macro definitions which must appear early in the source code. Other common decks may be called either in alphabetic order or in functional order. One example of functional order would be the storage mapping of a common block which can only be described by using several common decks (this can arise in a situation where nested common decks would be desired but the product is is built via MODIFY) correct storage mapping would thus require that the common decks be called in a particular order for which alphabetic naming may not be reasonable.

2. Statement formats

a. The FOR keyword may be used. CONTROL FASTLOOP (FASTFOR) is not permitted.

b. FOR loops and simulated case statements are allowed to terminate with a RETURN statement or the IORET macro. In the editor, the ERRJUMP call may be used to terminate any block of code. ERRJUMP will be a procedure which is itself allowed to execute a jump into a procedure. ERRJUMP is used to clear the editor into a nominal condition after encountering a syntax error. In the editor, code may also be terminated by a call to a fatal-error routine.

Loops may be based on labels and GOTO-s in place of FOR only when the programmer can defend this usage as substantially more efficient or as being simpler to maintain than functionally equivalent structured code.

Simulated case statements may use a backward jump to achieve the common exit when the case is embedded in an iterative structure for which labels and GOTO-s are allowed.

c. A PROC/FUNC/PRGM statement shall begin in column 1 for a compilation unit and for a first-level nested PROC/FUNC. PROC/FUNC statements nested to deeper levels shall be indented 2 columns per level. The body of code in a routine shall be indented 2 columns from the PROC/FUNC statement. Code contained in a CONTROL IF bracket shall be indented 2 columns from the CONTROL statement. BEGINs and ENDs shall be indented 2 columns, and the code within the BEGIN/END shall be aligned with the BEGIN/END. In the editor, IOBEGIN and IOEND macros shall be indented as though they are BEGIN/END.

3. Documentation

a. Documentaton of ENTRY/EXIT conditions and of storage usage must include assumptions regarding manipulations of pointer words for based arrays.

b. For compilation units whose main entry point is uncalled, the main entry may carry documentation considered applicable to all embedded procedures.

c. XREF and XDEF may be provided by lists of routine names in common decks. Such lists of XDEF should be listed, but such lists of XREF should not be listed except for a comment noting the call to the common deck. DOCMENT brackets are not required.

d. Stand alone comments may be a single line starting with a pound sign in column 1 and ending with a pound sign in column 71, rather than the COMPASS style comment (asterisk in column 1 of the comment body).

The use of preceeding and proceeding blank lines is negotiable between the programmer and reviewer to achieve a mutually satisfactory visual effect. Note that this simplified form for stand alone comments is only applicable for comments not intended to be printed by the DOCMENT utility.

4. Pseudo-reentrancy considerations (for FSE and SMFEX only).

a. The SMFEX Executive may contain a limited number of labels within if or for blocks, and external labels within procedures, as necessary to implement pseudo-reentrancy.

b. SMFEX and FSE will contain procedures subject to reentry under control of the SMFEX Executive. A reentrant procedure is a procedure which calls another reentrant procedure or uses the delay or recall statements. There cannot be reentrant functions.

c. The reentrancy technique severly restricts the usage of local storage and of parameters. The programmer should dedicate common block storage to the functions performed by a reentrant routine, in preference to locals. Note that the common block includes one general purpose variable which is stackable, so that reentrant routines can dynamically allocate storage on a limited scale.

d. Reentrant procedures must minimize the use of local storage. Any sequence of code in a reentrant procedure which uses local storage must be preceded and followed by stand alone comments of the form

```
    # LOCAL #

    # END LOCAL #
```

The code within the comments cannot call any reentrant routines.!

e. Reentrant procedures must minimize the use of parameters. When parameters are used, it is essential that the parameters be read-only (i.e. the subroutine does not compute a new value), and they must be used before any reentrant procedure is called. Use of parameters shall be followed by a stand alone comment of the form:

```
    # END PARAMETERS #
```

f. Reentrant routines lose control by calling DELAY or RECALL. In the single-user version, these are COMPASS subroutines which execute recall macros. In the multi-user version, these are DEF-ed to be calls into certain entry points within SMFEX to invoke the multi-tasking executive.

g. Reentrant routines are bracketed by the IOBEGIN and IOEND macros. In the single-user version, these are DEF-ed to simply yield BEGIN and END. In the multi-user version, these are DEF-ed to generate code to maintain data structures which help the SMFEX multi-task executive supervise the reentry. Reentrant routines cannot use the RETURN statement, but can use the IORET macro.

h. Reentrant routines must be restricted as to the type of monitor calls they can issue either explicitly or by calling other routines. In particular, reentrant code must use only CIO and each CIO call must be explicit. This effectively bans the use of the standard NOS common decks. Furthermore, the only file which can be dealt with by reentrant code is the editor workfile. Terminal I/O will be funneled into one module of code, which shall conditionally compile to yield conventional FET-s and CIO calls for FSE, and calls to the SMFEX Executive for SMF.

i. The only writeable storage which can be used other than local storage as described above shall reside in a single common block, or shall reside in based arrays whose pointer words are in the common block. The common block shall be organized into several sections based on the various degrees of reentrancy services provided by the SMFEX Executive. In the single-user editor, portions of this common block must be compiled to map exactly the same as the multi-user version, since that portion of the common block is tranferred verbatim through the workfile for communication between the two versions of the editor. All critical storage mapping must be identified as such in documentation.

j. Reentrant code shall minimize dynamic relocation of based arrays. Relocation is allowed if the pointer word is treated as non-reentrant. Relocation is possible with limited reentrancy provided the pointer word is mapped into the reentrant section of the common block. Note that while this will keep a pointer value alive for the duration of disk I/O, it is not able to keep any pointer valid across terminal I/O unless the pointer points within reentrant common itself. This is due to the re-mapping of array locations performed by the SMFEX Executive upon internal swaps. For those arrays re-mapped by SMFEX swapping, no module except SMFEX can ever change the pointer word.