TOPS20 Coding Standards and Conventions
Revision 0 - 17 January 1974
Revision 1 - 8 September 1976
Revision 2 - 9 March 1983

1.0 INTRODUCTION

The following is the machine language source code standard for TOPS20. Portions of it are applicable to all sources while the remainder is specifically relevant to the monitor.

The following conventions apply to MACRO sources.

2.0 STATEMENTS

The general form of a statment shall be:

```
TAG:    OPCODE AC,@ADDR(X)      ;comment
```

where:

1. Tabstops are assumed to be set every 8 spaces.

2. The TAG begins at the left margin.

3. The OPCODE begins at the first tab stop. There will be one TAB before the opcode. If the TAG and colon(s) occupy 8 or more spaces, there will be nothing between the colon(s) and the opcode. Exceptions to this rule apply in blocks, multi-line literals, and following skipping subroutine calls, see below.

4. One SPACE shall follow the opcode unless there are no other fields except the comment to be specified in the statement.

5. When any field is not used by the instruction, it may be omitted along with its directly related punctuation. A field which is affected by an instruction may NOT be defaulted to 0 by omitting it.

6. The semicolon which begins the COMMENT shall be at the 4th tab stop. There shall be one or more tabs preceeding the semicolon as necessary to place the semicolon at the 4th tab stop unless the preceeding fields extend to or beyond the 4th tab stop. In this case, one SPACE shall be used to separate the last preceeding field and the semicolon.

The instruction which follows a skipping monitor call or subroutine call shall be indented 1 additional space (1 space beyond the first tab stop) to indicate the possibility of that instruction being skipped. The ERJMPx or ERCALx following a JSYS shall be similarly indented. Indenting in this manner shall NOT be done following skipping machine instructions.

All symbols shall be typed in upper case only. Appropriate upper/lower case usage is recommended for any text that will be displayed to users. See comment case conventions below.

3.0 COMMENTS

A comment on the same line as a statment shall begin at the 4th tab stop as described above. When a comment consists of a single sentence or phrase which requires more than one line, the subsequent lines should have one SPACE between the semicolon (at the 4th tab stop) and the comment to indicate to the reader that the several lines are part of one logical statement. E.g.,

```
        IDIVI T1,XYZ            ;Now compute the remainder which
                                ; will be the number of bytes.
```

A comment that is not part of a particular statement shall begin at the left margin. Such comments, whether one or more lines, shall be preceded and followed by a blank line. This type of comment appears as routine headers and as general description of the purpose or algorithm of following code, and may appear within the flow of control. E.g.,

```
        code                    ;comment
        code                    ;comment

;Long comment line 1
;long comment line 2

        code                    ;comment
        ..
```

Comments may be upper and lower case, or upper case only. Appropriate upper/lower case usage is particularly recommended for full-line comments, routine headers, and other cases where the comments may be read as english.

A long comment (10 or more lines) may be enclosed within a REPEAT 0 or COMMENT pseudo-op and the semicolons omitted.

Extensive commenting of source listings is strongly encouraged.

1. Routines, modules, sections, macro definitions, etc. should be described at their beginning. See requirements for subroutine comments below.

2. Comments should appear on almost every statement line. As the reader views the listing page, the comments (aligned at the 4th tab stop) should appear as a running commentary on what the code is doing. These on-line comments should describe the logical procedure being carried out, not just describe the obvious action of the instruction. Humorous or irrelevant comments (e.g. ;OOPS..., ;OH WELL...) are discouraged since they provide no information to the reader.

A reader should be able to read the comments WITHOUT SEEING THE CODE and obtain a coherent understanding of what the program is doing. When a variable or other mnemonic symbol is referred to in the comments, an english phrase rather than the mnemonic itself should frequently be used (e.g. "last page address" rather than "LPGADR"). Comments, particularly routine headers, should describe "why" non-obvious actions are being taken and/or what assumptions are being made (e.g. "here when ..").

4.0 LISTING PAGES

Source listings shall be divided into pages by formfeed (control-L) characters. A CRLF shall precede and follow each formfeed. The last character in a source file shall be a formfeed with no CRLF following. Source files shall be arranged so that major modules, subroutines, etc. begin at the top of a page. Only when a subroutine is a quarter page or less in total size shall it begin other than at the top of a page.

It should rarely be necessary for flow of control to cross a listing page. That is, the last instruction on each page would normally be an unconditional transfer of control not preceeded by a skipping instruction. An unbroken sequence of instructions longer than one listing page is strong evidence of insufficient subroutinization.

However, when a sequence of instructions does cross a page, the last line on the preceeding page and the first line on the following page should be a comment line of the form

```
; ..
```

where the semicolon appears at the first tab stop directly under the preceeding opcode. E.g.,

```
        MOVE A,FOO              ;comment
        ; ..
^L
;COMMENT

        ; ..
TAG:    MOVEM A,FIE             ;comment
```

5.0 MACROS

All assemblies shall begin with SALL which has been shown to produce the most readable assembly listings.

In general, macros worth defining at all are worth defining on a monitor-wide basis. Therefore, localized, special-purpose macros are discouraged.

For top-level macro definitions, the DEFINE shall appear at the left margin and be followed by one space. The name of the macro being defined shall appear next, followed by one space. The dummy argument list, if any, shall appear next, followed by the open angle-bracket. E.g.,

```
DEFINE MACNAM (A,B,C)<
```

or

```
DEFINE MACNAM <
```

A CRLF shall follow the open angle-bracket, and the body of the macro definition shall begin on the next line, except when the entire macro definition is on one line in order to be used as part of an expression.

Macro calls generally do not require parentheses surrounding the arguments and should generally be avoided for short calls, e.g.

```
        LOAD T1,STRUC,(Q1)      ;data structure ref, Q1 index
```

Parentheses may be used when helpful for clarity, e.g. in instances of the BUG. macro.

Angle brackets must be used to quote any argument containing non-alphanumeric characters, except where such characters are paired delimiters and matching open and close characters are contained within the argument.

5.0 CONDITIONALS

Top level assembler conditionals shall be indented 3 spaces from the left margin (so that tag and comment lines may always be leftmost). Lower level conditionals shall be indented 3 or more spaces. The terminating angle-bracket of a conditional shall appear:

1. Immediately following the last instruction if the conditional has a short range.

2. On a separate line indented the same amount as the pseudo-op which began the conditional.

E.g.

```
        TAG:
           IFE FTFOO,<
                MOVE A,MUMBLE           ;comment
                MOVEM A,ZOT>            ;comment
```

or

```
           IFN FTFOO,<
        TAG:    MOVE A,MUMBLE           ;comment
                ..
                JRST XYZ                ;comment
           >                            ;END OF IFN FTFOO
```

A closing anglebracket shall NEVER appear in a comment (i.e. following a semicolon on the same line). The coding shall be correct even if the assembler were made to ignore angle-brackets in comments.

Assembler conditionals are generally to be avoided; inclusion of individual features under assembly switch control is not planned for TOPS20.

7.0 INSTRUCTION MNEMONICS

The standard PDP-10 instruction mnemonics as defined by the DECsystem-10 reference manual shall be used throughout. No abbreviated opcodes shall be used.

Macro or opdef definitions shall be made to define a useful mnemonic which is related to a function being performed in the code. See the subroutine conventions below for examples. Additional definitions consistent with this philosophy and these examples may be made with the approval of the project leader.

8.0 VARIABLES AND STRUCTURES

Use of the stack variable and data structure facilities in MACSYM is recommended. See MACSYM documentation. Because of these facilities, the following should be observed:

1. Explicit PUSHing and POPing of quantities is never done.

2. Explicit referencing of the stack, e.g. as -n(P) is never done.

3. Fields within data blocks or tables are not referenced by half-word instructions or explicit byte pointers but rather by LOAD, STOR, etc. In particular, the following technique is to be avoided:

```
        ..
        LDB T1,DATPTR           ;LOAD VIA MAGIC POINTER
        ..
        ..
DATPTR: POINT 9,OFSET(Q1),17    ;POINTER TO PARTICULAR DATA ITEM
```

The above obscures the AC being used for the reference, particularly if the byte pointer is not nearby in the listing. Recommended instead is an appropriate structure definition with DEFSTR, and a reference like:

```
        LOAD T1,STRNAM,(Q1)
```

which makes the index explicit.

4. Flags are defined with DEFSTR, MSKSTR, or as full-word parameters, and are referenced with the TX, TM, or TQ macros. Flags should never be defined as half-word quantities which require the programmer to remember whether to use TL or TR.

9.0 JSYS CALLS

Monitor-call JSYSes may be used in user or monitor code. All ACs are preserved over a JSYS call unless an explicit statement to the contrary appears in the JSYS description. ACs are changed over a JSYS call only when values are to be returned to the caller.

The JSYS name shall appear as the opcode in the statment which performs the call. The JSYS mnemonic includes the instruction field, so no other fields are supplied by the user.

Unimplemented JSYSes will invoke the illegal instruction sequence (with error code ILINS2). Defined and implemented JSYSes will return to caller +1 on success, or will invoke the illegal instruction sequence on failure. The illegal instruction sequence recognizes an ERJMPx or ERCALx following the failing JSYS and causes the appropriate

action. If the following instruction is not an ERJMPx or ERCALx, an illegal instruction interrupt is requested which will be handled by the executing fork if enabled, or otherwise cause a forced fork termination. See paragraph below on JSYS returns for proper indication of JSYS failure.

All constant values, bits, and fields of JSYS arguments shall have mnemonics defined according to the rules in MONSYM. The JSYS code itself shall use these symbols for loading arguments, testing bits, etc.

When writing code to implement a JSYS, the following conventions shall be observed:

1. The entry point of the JSYS is defined by a global tag which consists of a DOT concatenated with the symbolic name of the JSYS, e.g. .GTJFN::.

2. The first statement of the JSYS code shall be MCENT (Monitor Context ENTry). This establishes the normal JSYS context for a "slow" JSYS. At this writing, MCENT is a null macro and the JSYS entry procedure is invoked automatically. The use of MCENT is required so that this implementation may be changed in the future if necessary.

3. All caller ACs are automatically preserved by the entry and exit procedures. Therefore JSYS routines specifically should NOT save and restore the ACs. The contents of the caller's ACs 1-4 are copied into the callee's ACs. No callee ACs are copied back to the caller's AC block on return however; one of the "previous context" instructions* must be used to return any values to the caller. E.g.,

```
UMOVEM T1,T1            ;store monitor T1 into user T1
```

   A previous context instruction may also be used at any time to fetch the original contents of the caller's ACs unless they have been explicitly changed by a previous context store operation. E.g.,

```
UMOVE T2,T1             ;load user T1 into monitor T2
```

4. Return from JSYS code should be effected by the statement

- - - - - - - -

* - UMOVE, UMOVEM, XCTU [instruction], etc.

```
        MRETNG                  ;Monitor RETurn Good
```

This transfers to the JSYS exit sequence (returning caller +1) and should be used to indicate successful completion of the JSYS. If the JSYS could not be completed successfully, the following statement should be used:

```
        ITERR errcod            ;causes an Instruction Trap
                                ;ERRor, leaves
                                ;the error code in LSTERR
```

Certain other statements are defined which effect JSYS returns according to a previous convention. They are:

```
        RETERR errcod           ;RETurn ERRor, return
                                ;caller +1 with error code
                                ;left in AC1 and LSTERR

        EMRETN errcod           ;Error Monitor RETurn, return
                                ;caller +1 with error code left
                                ;in LSTERR
```

RETERR and EMRETN should not be used in new JSYS code but may be needed if existing JSYSes are modified.

All error returns shall include an error code (mnemonic) which shall be defined in MONSYM.MAC. If the appropriate error code has already been loaded into AC1, then the errcod field may be omitted from the above and the contents of AC1 will be taken as the error code. No JSYS shall return other than +1 or instruction trap, therefore no occurrance of AOS 0(P) should ever be required in JSYS code.

When invoking an error return from JSYS code, care must be taken to unlock any locks and release any resources that may have been acquired in the execution up to that point. The only exceptions are:

1. An explicit OKINT need not be done.

2. Any stack usage created by STKVAR, TRVAL, SAVEAC, etc. will be released automatically.

10.0 SUBROUTINE CALLING - INTERNAL MONITOR ROUTINES

The allocation of ACs for all inter- and intra-module subroutine calls shall be:

ACs 1,2,3,4 -- Arguments for simple subroutine calls.

ACs 0, 5-14 -- Preserved, not changed by subroutine (or saved and restored if necessary).

AC 15 -- Preserved, frame pointer. (See TRVAR, BLCAL., etc.)

AC 16 -- Temporary, used by JSYS call/return procedure and reserved for use by other call/return procedures.

AC 17 -- Global stack pointer

In the past, ACs 1-4 have been used as general temporaries. However, they are not consistently saved by calling code. Hence, when modifying any existing subroutine, you must assume that any AC not already being changed by the subroutine is assumed to be preserved by the callers, and such ACs should continue to be preserved. Current recommended practice is that AC1-4 be preserved except where values are explicitly intended to be returned.

For "simple" subroutines, call and return shall be effected by PUSHJ P, and POPJ P, respectively. 'CALL' (= PUSHJ P,) shall be used to call subroutines, e.g. CALL SUBR.

For larger subroutines and for those with more than 4 arguments, BLCAL./BLSUB. shall be used.

'RET' (= POPJ P,) shall be used to return +1 from all subroutines.

'RETSKP' shall be used to return +2 from subroutines. RETSKP is equivalent to:

```
JRST [  AOS 0(P)        ;May be skipped
        RET]
```

'RETBAD errcod' may be used to return +1 with an error code from a subroutine. RETBAD assembled as one instruction and may be skipped over.

'CALLRET' may be used to call a subroutine and return immediately thereafter. It is an abbreviation for

```
CALL SUBR
RET
```

or

```
CALL SUBR
 RET
RETSKP
```

CALLRET assembles as a single instruction and may be skipped over, but all uses of it must be such that the code would still work correctly if the CALLRET were replaced by a jump to the above sequence. Keep in mind also, that use of CALLRET provides less debugging information on the stack on a crash or breakpoint.

Return may also be effected by transferring control to the global tag RTN or RSKP, e.g.

```
JUMPE A,RTN             ;equivalent to JUMPE A,[RET]
JUMPN A,RSKP            ;equivalent to JUMPN A,[RETSKP]
```

ACs 1-4 shall be used for passing arguments to "simple" subroutines and returning values. AC1 shall be used for a single argument routine, ACs 1 and 2 for a two-argument routine, etc.

The BLCAL./BLSUB. mechanism places arguments on the stack and makes them available symbolically within the subroutine. Values already in ACs may be given in the call, e.g.:

```
BLCAL. SUBR,<T2,Q1,ZOT>
```

A routine defined to return caller +2 (skip) on success and caller +1 (noskip) on failure is acceptable. Returns greater than caller +2 are not permitted.

See "Subroutine Documentation" below for commenting practices at subroutine call and entry points.

11.0 AC DEFINITIONS

The following mnemonics have been chosen to be consistent with the AC use conventions above. The preserved ACs are divided into three groups, F (1 AC) intended for Flags, and Q1-Q3 and P1-P6 intended for general use. The ACs within each group are consecutive.

0 - F	10 - P1
1 - T1	11 - P2
2 - T2	12 - P3
3 - T3	13 - P4
4 - T4	14 - P5
5 - Q1	15 - P6
6 - Q2	16 - CX
7 - Q3	17 - P

The programmer should assume that each group (Tn, Qn, Pn,) is in ascending order, e.g. that T2=T1+1, but that the specific assignment of numbers may change.

Explicit numeric offsets from AC symbols (e.g. T1+1) should NEVER be used. Instructions which use more than one AC (e.g. DIV, JFFO) must be given an AC operand such that the other AC(s) implicitly affected are in the same group. E.g. T3 (and T4) is OK for IDIV because T3+1=T4, but Q3 is not because Q3+1=??.

There are several facilities to save and automatically restore ACs. Each of these will save all of the indicated ACs on the stack at the point of execution and will place a dummy return on the stack which causes these ACs to be restored automatically when the current

routine returns. Use of these facilities eliminates the need for matching PUSH/POP pairs at the entry at exits of routines and eliminates the bugs which often arise from an unmatched PUSH or POP.

The available macros are:

```
SAVEAC <list> - save the ACs given in the list, e.g. <T2,Q1,...>
SAVEQ - saves ACs Q1-Q3
SAVEP - saves ACs P1-P6
SAVEPQ - saves ACs Q1-Q3 and P1-P6
SAVET - saves ACs T1-T4
```

Defining alternate names for ACs may be done only in specific ways. These methods ensure that only one mnemonic is valid for a particular AC at any place in the code. Multiple definitions must be avoided because of the possibility of inadvertently referring to the same AC in two different ways.

The ACVAR facility may be used to define variables which reside in ACs within a subroutine. The matching ENDAV. will purge the local AC names. The general preserved AC names should not be used within the range of an ACVAR.

In some cases, it may be desirable to define a variable as an AC throughout an entire module, or within several cooperating modules. Within these modules, the usual name for the AC must be purged so that there is no possibility of using two different symbols for the same AC.

Only preserved ACs may be used for special definitions. The procedure for declaring a functionally defined AC is:

```
DEFAC NEWAC,OLDAC
```

This must be done at the beginning of an assembly, and it defines NEWAC to be equal to OLDAC. OLDAC must be the mnemonic for one of the regular preserved ACs, and this mnemonic will be purged and therefore unavailable for use in the current assembly.

An AC with a special definition should not be used for other purposes; e.g. "JFN" should not be used to hold some quantity other than a JFN merely because it happens to be available.

Where a module has a special AC definition, or where two cooperating modules have the same special AC definition, a value may be passed in the AC for subroutine calls within or between the modules. This is the only case where subroutine arguments may be passed in preserved ACs. Only the type of data implied by the AC variable name may be passed in the AC. As above, AC "JFN" may be not used to pass some quantity that is not a JFN.

12.0 SUBROUTINE DOCUMENTATION

The following is a suggested format for documenting the calling sequence of a JSYS or subroutine. A description of this sort should appear at the beginning of every subroutine, no matter how short.

```
;name of subroutine - function of subroutine, etc.
; T1/ description of first argument
; T2/ description of second argument
;   ...
;       CALL NAME or JSYSNAME   ;(arglist)
; RETURN +1: conditions giving this return,
;  T1/ value(s) returned
; RETURN +2: conditions and values as above.
```

1. The arguments, if any, should be documented as the contents of registers and/or variables as shown. MONSYM mnemonics should be used when available; e.g. at JSYS entry points. For BLSUB. routines, the argument names rather than ACs should be used in the description.

2. The actual instruction to do the call should be shown, including the argument comment as described below.

3. The return(s) should be noted as shown; "ALWAYS" or "NEVER" may be used as the condition where appropriate; the +2 return need not be shown if it does not exist; values returned should be described in the same form as arguments.

Examples:

```
;SIN - Computes sine of an angle
; T1/ angle in radians, floating point
;       CALL SIN                ;(T1/T1)
; Return +1: Failure, unnormalized number or out of range
;       +2: Success, T1/ value, floating point

SIN::   ..
```

- - - - - - - -

```
;Assign request node
; DTENO - Number of DTE to which message is being sent.
; BLKFLG - Non-0 if blocking is prohibited.
; FAILFL - Non-0 if failure can't be handled.  If failure
;          can be handled, the reserve won't be used.
;       BLCAL. ASGNOD,<DTENO,BLKFLG,FAILFL>
; Return +1: Failure, no space available
;       +2: Success, T1/ node address

ASGNOD: BLSUB. <DTENO,BLKFLG,FAILFL>
        ..
```

- - - - - - - -

```
;GJINF - Get job information jsys
;       GJINF                   ;(/T1,T2,T3,T4)
; Return +1: always,
;  T1/ Logged-in directory number
;  T2/ Connected directory number
;  T3/ Job number
;  T4/ Terminal number or -1 if detached

.GJINF:: ..
```

12.1 Subroutine Calls

Each call to a subroutine shall be commented such as to indicate the AC variables being passed to the subroutine. The form shall be:

```
;(list of input variables/list of returned variables) comment
```

Example:

```
        CALL FOO                ;(T1,T2/T1) fix ornary overflows
```

This means that T1 and T2 are passed to the routine, and T1 is returned.

```
        CALL FORKXY             ;(FX) xy the fork
```

This indicates that an argument is passed in a preserved AC.

```
        CALL GETZOT             ;(/T1) Get a zot
```

This indicates that no arguments are accepted, but T1 is returned.

```
        CALL UPDTCK             ;() update TODCLK
```

This indicates no AC variables are accepted or returned.

The comment implies nothing about the rest of the ACs; they are presumed to be preserved or not according to the normal conventions or as documented at the head of the subroutine.

This convention need not be followed if a call syntax is used which explicitly identifies the arguments, e.g. BLCAL.

New code should follow this convention. Existing code may be modified as the opportunity arises. Any call in existing code can be updated without necessarily updating other calls in the same area or other occurances of the same call. Updating only some calls causes no ambiguity since the case of "no arguments" is distinct from "not documented".

13.0 STRUCTURE

Clean and orderly structuring of code is required. To a large extent, this is accomplished by appropriate use of subroutines according to the foregoing conventions. In order to make code modular and easy to understand, a subroutine may be created which is only called from a single place.

Beyond that, techniques for representing structure in code within subroutines is also recommended as follows.

Use IFSKP. group of macros. See MACSYM documentation for more details. For example:

```
SKIPE FLAGWD            ;TEST FLAG WORD
IFSKP.
  ..                    ;THIS EXECUTED IF FLAG WORD IS 0
  ..
ENDIF.

PMAP%
IFJER.
  ..                    ;THIS EXECUTED IF PMAP% fails
  ..
ENDIF.

IFN. T3
  ..                    ;THIS EXECUTED IF T3 IS NOT 0
  ..
ENDIF.
```

Small loops can be constructed with DO./ENDDO.

The opcode should be indented an extra two spaces for each nesting level of IF or DO. E.g.,

```
CAMN T1,FOO
IFSKP.
  MOVE T2,FIE
  TXNN T2,BIT
  IFSKP.
    CALL FUM
    SETZM FLAG
  ENDIF.
  MOVEM T1,FOO
ENDIF.
```

Generally, the block structure macros are preferred over multi-line literals for branching paths. With the macros, the binary listing is continued within the range, nesting can be added more easily, and there may be some speed advantages in a pipeline machine. The macros generally assemble the same or fewer instructions for the same semantics. For example, a common case is:

```
        SKIPx
        JRST [  code
                code
                JRST .+1]
        code
```

This can be represented as:

```
        SKIPx
        IFNSK.
          code
          code
        ENDIF.
```

which assembles as the same number of instructions. If the sense of the skip can conveniently be reversed, the representation:

```
        SKIPNx
        IFSKP.
          code
          code
        ENDIF.
```

assembles as one fewer instruction.

A further advantage of the block macros is that labels are not required. A label can potentially be referenced anywhere in nearby code or in the module, and so the flow of control is less clear. The equivalent of the above case,

```
        SKIPNx
        JRST LAB1
        code
        code
LAB1:   ..
```

leaves the reader to guess whether there is any other branch to LAB1 or consult the CREF.

If multi-line literals are used, the following rules apply:

1. The opening bracket for a multi-line literal should occur in the position that the first character of the address field would have appeared if the instruction had an ordinary address, e.g.

```
SKIPGE FOO
JRST [
```

2. The first and all following instructions within the literal shall begin at the second tabstop, e.g.

```
JRST [  MOVE A,MUMBLE   ;comment
        JRST FIE]       ;comment
```

The tab between the open bracket and the first opcode may be omitted if the line position is already at or beyond the second tab stop, e.g.

```
JUMPGE A,[MOVE A,MUMBLE
```

3. The closing bracket shall follow the last field of the last instruction (as above), and shall be before the comment on the same line.

4. Nesting of multi-line literals to a depth greater than one is discouraged because of awkward formatting problems.

5. Tags may not appear in multi-line literals.

6. Multi-line literals should be of limited length, about 10 lines maximum.

7. Use of ".+1" is legal in a literal to return to the main sequence.

13.1 Flow Of Control - Branch Conventions

In general, jumps should be to tags forward (down the page) from the point of branch except for loops. Tops of loops should be identified by comment.

The expressions ".+1" and ".-1" are the only legal uses of "." (this location). All other uses should be strictly avoided.

"Global" jumps should be avoided altogether. Higher-level languages do not permit them, and with good reason. The only exceptions are jumps to well defined and published exit sequences, e.g. RTN, RSKP (see subroutine conventions, above).

4.0 NUMBERS

There should be no occasion to use a literal number in in-line code. All parameters, bit definitions, CONO/CONI codes, etc. should be defined mnemonically at appropriate places.

Appendix A

LIVING IN AN IMPERFECT WORLD

Much of the present TOPS20 code was written before the existence of this standard and therefore does not conform to it. A great deal of systematic editing has already been done to improve conformance, but obvious irregularities exist. In general, new code being added should conform exactly to this standard even if being integrated with old code. The following are some specific problems which may arise and the recommended solutions:

14.1 AC Mnemonics

Some code uses absolute numeric ACs. If new code is being integrated into a sequence which uses numeric ACs, it is desirable that the existing code be edited to use the standard mnemonics, particularly for the preserved ACs. If the programmer cannot take the time to do that, then the mnemonics T1-T4 should be used for ACs 1-4, and other ACs should be referenced in the same way as is done by the existing code.

Some code uses mnemonics A,B,C,D for the temporary ACs. These same mnemonics should be used for new code integrated into this existing code, or all references can be edited to use the standard mnemonics.

You may write some code using the standard mnemonics for preserved ACs and then discover that the module into which you wish to put this code has redefined some of these ACs. The solution is one or a combination of the following:

1. Move the new code to a module which does not redefine the preserved ACs.

2. Use different preserved ACs -- ones which have not been redefined. (Note it is not acceptable to use an AC with a special definition for other than its special purpose.)

Clearly, code which needs some of the special definitions must be placed in a module which has these ACs defined and must therefore use only the other preserved ACs.

Note that a value which usually resides in a special AC need not ALWAYS reside there. For example, if code in JSYSF needs to call a routine in PAGEM and pass a JFN index as an argument, the JFN should be loaded into T1-T4 for the call since PAGEM does not have JFN defined and cannot accept an argument in it.

4.2 Stack Handling

Use of the several stack variable facilities defined in MACSYM is recommended. Some old code uses explicit PUSH and POP and references of the form -n(P) however, and when anything more than trivial modifications must be made to such code, it is most strongly recommended that the code be edited to use STKVAR or TRVAR. Failing that, references must be consistent with the existing code.