X-603-74-34
PREPRINT

NASA TM X- 70599

STRCMACS: AN EXTENSIVE SET OF MACROS FOR STRUCTURED PROGRAMMING IN OS/360 ASSEMBLY LANGUAGE

C. WRANDLE BARTH

JANUARY 1974

GSFC — GODDARD SPACE FLIGHT CENTER —
GREENBELT, MARYLAND

Reproduced by
NATIONAL TECHNICAL
INFORMATION SERVICE
US Department of Commerce
Springfield, VA. 22151

X-603-74-34

STRCMACS: AN EXTENSIVE SET OF MACROS FOR STRUCTURED PROGRAMMING IN OS/360 ASSEMBLY LANGUAGE

C. Wrandle Barth

January 1974

GODDARD SPACE FLIGHT CENTER
Greenbelt, Maryland

CONTENTS

CONTENTS (continued)

STRCMACS: AN EXTENSIVE SET OF MACROS FOR STRUCTURED PROGRAMMING IN OS/360 ASSEMBLY LANGUAGE

The latest buzzword in the computer profession is "structured programming." The term has been applied to any of a number of techniques all of which are claimed to improve software reliability and modifiability. These various techniques have been eyed with suspicion by some and praised by others. Performance levels of greater than 1800 lines of code per error have been reported.[1] In a world where late deliveries, release *n* (where *n* grows without bound), PTFs, and bugs in production programs have been everyday experiences, such methods certainly seem to have something to offer.

We will discuss the two techniques which have been most often referred to as "structured programming." One is that of programming with high-level control structures (such as the if and while) replacing the branch instruction ("goto-less programming"); the other is the process of developing a program by progressively refining descriptions of components in terms of more primitive components (called "stepwise refinement" or "top-down programming"). In addition to discussing what these techniques are, we will try to show why their use is advised and how both can be implemented in OS assembly language by the use of a special macro instruction package.

The use of assembly language itself is being questioned by many as being counter-productive to software reliability. The trend is for moving away from assembly language and its preoccupation with machine level details towards higher-level languages. Even operating systems are being written more and more in high-level languages. There are still many programs in the real world, however, which must be written in assembly languages, either due to efficiency, interface problems, or to provide certain capabilities. Since these programs often have strict reliability requirements, it makes sense to provide a mechanism for using structured programming techniques directly in assembly language. Much of what will be said in the following pages, however, is applicable to programming in higher-level languages as well.

Structured programming is not, of course, a panacea. Nor will switching to structured programming automatically improve the quantity or quality of programs produced by every programmer. Terrible programs can be written in any language, for any system, and using any techniques in the world. But

[1] Baker, F. T., "System Quality through Structured Programming," *FJCC* 1972, pp. 339-343.

the majority of those who have used the structured programming techniques have found that the code they write is better, contains fewer bugs, and is easier to maintain and modify than that which they had written previously using conventional programming.

GOTO-LESS PROGRAMMING

One of the biggest controversies in the programming community in recent years is the worth of the goto statement (*i.e.*, the unrestricted branch instruction) in programming languages. While it has been known for some time that it is theoretically possible to program any problem capable of algorithmic solution without the use of gotos, attitudes on the practicality of writing computer programs in such a style have ranged from total disbelief to reverential zeal. However, as more and more people become involved in the issues, the number of people advocating goto-less programming seems to be increasing continually.

One of the first printed objections to the goto was E. W. Dijkstra's letter to the editor of the Communications of the ACM in which he suggested that the "quality" of a programmer was inversly proportional to the density of goto statements in his program. When the concept of goto-less programming is introduced to most programmers, there is understandable skepticism. The suspicion is often voiced that it must be terribly awkward to program after deleting what seems to be the most basic control mechanism of programming languages; and what code would be written must surely be inefficient and difficult to understand and debug. Yet, most persons who have attempted to write any amount of goto-less code are quick to state that the exact opposite is true: such programs are, they say, easier to understand and contain many fewer bugs and are often more efficient than their goto counterparts. It should be pointed out that programs may use well-defined control structures and still contain goto statements. The objection is not to the goto per se, but to its use for arbitrary branching. Branching is certainly required to choose between alternatives. For a FORTRAN programmer to write the equivalent of the ALGOL code:

```
if I < 5
 then
   J := 5;
 else
   J := I;
```

he would use:

```
    IF (I-5) 10, 20, 20
10  J = 5
    GO TO 30
20  J = I
30  CONTINUE
```

Branching occurs in either case. Properly optimized, the same object code is probably produced by either. The former is certainly goto-less; the latter is as goto-less as one can be in FORTRAN. The technique, then, is really to limit oneself to standard and properly nested control structures, the argument being that this improves the intellectual manageability of the program. When a language provides these control structures directly, programs can then be written using such statements instead of synthesizing them from the goto. Such code as the FORTRAN segment above which contains gotos, but only "good" gotos (*i.e.*, gotos which represent standard control structures) are sometimes called goto-less; we shall refer to such code as quasi-goto-less, to distinguish between it and truely goto-free code. In such quasi-goto-less code, the standard control structures are not always quite so obvious and it is easier to make mistakes than when the proper control structures are provided directly in the language. We shall return to a discussion of quasi-goto-less code later when we discuss language requirements for goto-less programming.

At the 1972 National Conference of the ACM, a debate was held on whether the goto should even be a part of future programming languages. The interesting thing about the debate was that even those who were trying to justify the retention of the goto did not do so on the grounds that it was required for good programming. In fact, one debater stated: "In my opinion, there have been far too many gotos in most programs .... The no goto rule....does improve the code produced by most programmers.... If I were teaching a beginning programming class, I would not teach the goto."[1] In the final analysis, all of the debaters seemed to agree that at least the vast majority of programming should be done without using gotos, with the only controversy being whether future programming languages should still allow its use or not even provide such a statement.

[1]Martin E. Hopkins, "A Case for the goto," Proceedings of the ACM, 1972 Annual Conference, pp. 787-790, August 1972. The last statement was mentioned as an aside during the debate and does not appear in the proceedings.

An excellent discussion of the rational for goto-less programming is given in "A Case Against the goto" by William A. Wulf. This speech was given in the goto controversy debate at ACM 1972 mentioned above. What follows is an attempt to summarize some of the major points of that paper. The reader is referred there for a more complete discussion.

The main objection to the goto is that it is possible to construct such a maze of gotos that control flow becomes completely obscured and such uses seem to be altogether too common. There are certain uses of the goto which form easily recognizable control structures and are, therefore, more intellectually manageable. But by providing these structures specifically by name (if, do, *etc.*), they become that much more recognizable. The reason for this emphasis on the intellectual manageability of programs is in recognition of the reliability problems which have occurred with major programming systems in the past decade. The modularization and proving of correctness of programs is going to be of primary interest if future systems are to provide substantial improvements over past performance. Both of these goals are greatly simplified in a goto-less environment.

Consider the example on the following page. On the left is a portion of a subroutine written in FORTRAN. (To avoid extraneous detail, conditional expressions have been represented by lower case letters and blocks of code containing no control statement are shown as capital letters in angle brackets.) On the right is the same program in SIMPL-M, a goto-less language. (The keywords "fi" and "od" are used to terminate the "if" and "while...do" constructs.) Suppose the program abended during the execution of <J>. What can be said about the truth of q? Of s? Is it possible that <F> was executed? If so, in what case? All of these can be answered from either program; but the results are more easily seen in the SIMPL-M version since the control flow is more graphic. In particular, it is much easier to trace the execution paths backwards when necessary than when gotos and labels are used since the immediate predecessor of any statement is easily determined.

The fact that it is theoretically possible to write programs in a goto-less environment is not particularly surprising since there is no explicit branching mechanism in a number of the formal systems of computability theory (*e.g.*, recursive functions, Post systems, Markov algorithms, *etc.*) and yet these systems have the same computational power as, say, FORTRAN. However, as Wulf points out:

> "this does not say that an algorithm for the [solution of an arbitrary problem] is especially convenient or transparent

Conventional programming

```
      IF (q)   GO TO 20
      IF (r)   GO TO 10
      ⟨A⟩
      GO TO 60
 10   ⟨B⟩
      GO TO 60
 20   IF (s)   GO TO 80
      ⟨C⟩
      ⟨D⟩
      ⟨E⟩
 30   IF (.NOT.(t))   GO TO 40
      ⟨F⟩
      GO TO 30
 40   IF (u)   GO TO 70
      ⟨G⟩
 50   IF (.NOT.(v))   GO TO 60
      ⟨H⟩
 60   ⟨I⟩
      RETURN
 70   ⟨J⟩
      GO TO 50
 80   IF (w)   GO TO 90
      ⟨C⟩
      ⟨D⟩
      GO TO 110
 90   ⟨C⟩
      ⟨D⟩
      ⟨K⟩
100   IF (.NOT.(x))   GO TO 110
      ⟨L⟩
      GO TO 100
110   IF (.NOT.(w))   GO TO 120
      ⟨M⟩
120   IF (.NOT.(v))   GO TO 60
      ⟨H⟩
      GO TO 60
```

goto-less programming

```
IF q
  THEN
     ⟨C⟩
     ⟨D⟩
     IF s
       THEN
          IF w
            THEN
               ⟨K⟩
               WHILE x
                  DO
                     ⟨L⟩
                  OD
               ⟨M⟩
            FI
       ELSE
          ⟨E⟩
          WHILE t
            DO
               ⟨F⟩
            OD
          IF u
            THEN
               ⟨J⟩
            ELSE
               ⟨G⟩
            FI
          IF v
            THEN
               ⟨H⟩
            FI
       FI
  ELSE
     IF r
       THEN
          ⟨B⟩
       ELSE
          ⟨A⟩
       FI
  FI
⟨I⟩
```

> in goto-less form. Alan Perlis has referred to similar situations as the 'Turing Tarpit' in which everything is possible, but nothing is easy."[1]

This brings up the practicality of writing and debugging goto-less programs. Wulf offers his experience with the designing, implementation, and use of the goto-less systems implementation language BLISS as a subjective argument for the method. He lists a number of large scale systems which have been written in BLISS and states: "Programmers familiar with languages in which the goto is present go through a rather brief and painless adaptation period. Once past this adaptation period, they find that the lack of a goto is not a handicap; on the contrary, the invariant reaction is that the enforced discipline of programming without a goto structures and simplifies the task."[2] Such subjective judgements seem to be fairly common among those who have done any appreciable amount of goto-less programming, while the majority of the reservations seem to be expressed by those who have never attempted it.

The main arguments for goto-less programming are:

- goto-less programs are easier to understand, debug, and modify.
- It is easier to prove assertions (in particular, to prove program correctness) about goto-less programs.
- Goto-less programs are less likely to contain bugs due to their intellectual manageability.
- Compilers are able to understand, and therefore to optimize, goto-less programs to a larger extent.
- Languages which contain the goto construct invite its misuse to make a "rat's nest" of control flow.

The first three of the above arguments provide sufficient reasons for programming goto-less in any language which provides the requisite control structures regardless of whether an actual goto is also present in the language or not.

---

[1] William A. Wulf, "A Case Against the goto," Proceedings of the ACM, 1972 Annual Conference, p. 794, August 1972.

[2] ibid, p. 795.

It should be mentioned that the languages in which goto-less is really feasable are more than bare-minimum languages. The theoretical considerations show that the required constructs are:

- some form of grouping statements into nestable "blocks"
- a conditional statement (such as the ALGOL or PL/I if)
- a repetition statement (such as the ALGOL for or the PL/I iterated do)

Other minimum sets of constructs may be selected which are equivalent (for example, CALL/RETURN, CASE [an n-way conditional], and recursion). However, there is no reason to limit ourselves to a minimum set, particularly since we are attempting to make the programming as straight-forward and perspicuous as possible. By providing a number of basic constructs, we avoid the need to contort the available forms to produce desired constructs.

One such special form is the BLISS leave statement. This provides for exiting from a loop (or other block) upon the discovery of unusual conditions before the normal termination test is satisfied. Such an exiting statement may allow the jumping out of several levels of blocks. This is no different than a series of ifs. The program:

```
        :
        :
OUTER:  begin;
          :
          :
INNER:    begin;                          [The BLISS language has been
            :                                simplified somewhat.]
            :
            if I = 0
              then
                leave OUTER;
            α
          end;
          β
        end;
        γ
```

has the same effect as:

```
          :
          :
OUTER:  begin;
          :
          :
INNER:    begin;
            :
            :
            if I ≠ 0
              then
                begin;
                  α
                end;
          end;
          if I ≠ 0
            then
              begin;
                β
              end;
        end;
        γ
```

Notice, however, that by using the leave statement, the immediate predecessors of γ are not quite as obvious; as a result, the compiler should give appropriate warning messages to flag the targets of leave instructions.

In languages which do not provide the necessary control structures, one must resort to quasi-goto-less code. Unfortunately, many of the advantages in the ease of reading and understanding and the avoiding of bugs is almost nullified when programming in the quasi-goto-less manner. The best approach when programming in such languages is to do the initial design programming in an "abstract programming language" — an arbitrary language (real or imagined) which provides sufficient high-level features to allow one to program the algorithm without being bogged down in extraneous detail — and then to translate (by hand or using a preprocessor) the abstract program to the required target language.

Such a method was used in the programming of the structured macros themselves. Since 360 macro assembly language contains no statement grouping capability nor any looping construct, the actual programming was done in an imaginary abstract programming language called SIMPL-M. This not only simplified the writing of the macros, but it also is the "source" language for documentation and certification purposes. The listing of the macros in Appendix

C is in SIMPL-M. This source was then hand-translated into macro assembly language in a straight-forward manner. Any changes or extentions are always made in both to assure the "source" is kept current.

More will be said about abstract programming languages in the section on stepwise refinement.

Those defending the retention of the goto in the ACM 72 debate used the following arguments:

- the goto is desirable for abnormal exits from a block or procedure
- code written with the goto can be more efficient than code written without
- the goto is useful for synthesizing new control structures

An excellent discussion of these points is provided in "A Case for the goto" by Martin E. Hopkins.[1] It is this author's feeling that perhaps a compromise solution to the controversy is in order, at least for the present. The goto could be provided, but with the status of a "disfavored instruction." As such, it would require the specification of a compiler option before it would be accepted at all. Even with the option turned on, each use would produce a warning diagnostic message.

An early version of the structured macros included a facility to assign a level-6 warning every time a branch instruction was generated. Since most standard cataloged procedures will not continue if any message higher than 4 occurs, this was treated as an error. If the user required the branch (as when it was generated by an OS/360 macro), he could raise the conditional-execution threshold to 7, thereby allowing the branch message to be treated as a warning, but still bypassing execution on any standard level-8 error messages. As of the release 20 assembler, however, it is no longer possible to use the technique which implemented the level-6 warning.

[1] Hopkins, ibid., pp. 787-790.

Three classes of programming languages may be distinguished. In the first, only goto-less programming may be done since no goto is provided. This group includes such languages as (pure) LISP,[1] ISWIM,[2] BLISS,[3] OREGANO,[4] GEDANKEN,[5] and SIMPL-X.[6] In the second class are languages which provide a goto, but also provide sufficient control structures to do goto-less programming. PL/I and assembly language using the STRCMACS are examples of languages in this class. In the remaining class (which unfortunately includes our most popular languages—FORTRAN and COBOL), insufficient control structures prevent doing anything beyond quasi-goto-less programming. (It is possible to do truely goto-less programming in both FORTRAN and COBOL by using the CALL/SUBROUTINE or PERFORM/SECTION mechanism. But when every block of two or more statements [in FORTRAN] or every nested structure [in COBOL] requires another SUBROUTINE or SECTION, the result is an overwhelming proliferation of modules and often a high linkage overhead. Furthermore, since the code is always out-of-line, readability is totally destroyed.)

The main advantage, then, of goto-less coding can be summed up as follows: by limiting the flow of control in modules to a few well-understood and carefully-defined constructs, one's understanding of the flow is aided and, therefore, the overall logic of the module is brought more within the grasp of the programmer, reader, and later the modifier of the program.

---

[1]McCarthy, J., *et al.*, *LISP 1.5 Programmers Manual*, The M.I.T. Press, Cambridge, Mass., 1962.

[2]Landin, P. J., "The Next 700 Programming Languages," *Comm ACM*, 9:3, pp. 157–166, March 1966.

[3]Wulf, W. A., *et al.*, "BLISS: A Language for Systems Programming," *Comm ACM*, 14:12, pp. 780–790, December 1971.

[4]Berry, D. M., "Introduction to Oregano," Proc. Symposium on Data Structures in Programming Languages, *SIGPLAN Notices* 6:2, February 1971.

[5]Reynolds, J. C., "GEDANKEN: A Simple Typeless Language Based on the Principle of Completeness and the Reference Concept," *Comm ACM* 13:5, pp. 308-319, May 1970.

[6]Basili, Victor R., "SIMPL-X: A Language for Writing Structured Programs," University of Maryland Technical Report TR-233, January 1973.

STEPWISE REFINEMENT

As was mentioned earlier, we use the term "stepwise refinement" to mean the process of developing a program by progressively refining descriptions of components in terms of more primitive components. (Some use the term "structured programming" to mean only "stepwise refinement.") It would be somewhat redundant for us to go to great depth into the subject of stepwise refinement, as there already exists a number of excellent papers on the subject. Predominent among these are Dijkstra's "Notes on Structured Programming,"[1] Wirth's "Program Development by Stepwise Refinement,"[2] and Hoare's "Notes on Data Structuring."[3] We will give here only a basic overview of the topic and refer the reader to the above papers for more details and examples.

Stepwise refinement is an outgrowth of the problem-solving process. Consider the following:

A. A problem is posed which requires solving; it is deemed capable of algorithmic solution and appropriate to computer solution. That is, *what* is to be done is well-defined; *how* it is to be done is not yet specific.

B. An algorithm is developed expressed in terms intelligible to an appropriate computer (possibly utilizing a translater as intermediary). A "*how*-to-do-it" is now well-defined; it purports to accomplish the "what" of step A.

C. A convincing argument is put forth that the "how" of B accomplishes the "what" of A.

The process of going from A to B involves a number of activities including: formalizing such terms as "find", "search", "summarize", and the like; defining data items to hold real-world quantities; and deleting vagueness. Such activities are the heart of programming.

The process of going from A to B need not be done in a single pass. The process is greatly simplified and the results are more understandable and

[1]Dijkstra, E. W., "Notes on Structured Programming," *Structured Programming*, Academic Press, 1972.
[2]Wirth, N., "Program Development by Stepwise Refinement," *Comm ACM*, 14:4, pp. 221-227, April 1971.
[3]Hoare, C. A. R., "Notes on Data Structuring," *Structured Programming*, Academic Press, 1972.

reliable if a number of levels are used. At the outermost level, the "what" to be accomplished is the "what" of A. But instead of moving directly to the "how" of B, we go to a "how" B_1 for some abstract super-machine with arbitrarily complex instructions. Most of the instructions of B_1 are not intelligible to our real computer. But the number of instructions are few (maybe 50 or so), so we can feel that, if there were a machine which could understand B_1, it would surely accomplish the task A. We can now take the instructions of B_1 (call them the $A_{2,i}$) and for whichever are not understandable to our real computer repeat the problem-solving process producing a program $B_{2,i}$ —the "how" for each $A_{2,i}$ in more primitive terms. This process is continued until eventually all instructions are in terms intelligible to our computer.

At this point, we have the program B written entirely in some machine-understandable language and all the intermediate "super-instructions" may be discarded. However, for the purpose of documentation and maintenance, it is probably desirable to save these intermediate programs. This may be accomplished in the following ways. (1) The name of the super-instruction can appear as comment cards surrounding the final instructions defining the super-instruction. (2) The super-instruction can be replaced by a call instruction and the definition of the super-instruction can be made a module (subroutine, procedure, or whatever) of it's own. (3) The super-instruction can be replaced by an invocation of a macro (compile-time call or INCLUDE statement) and the definition of the super-instruction can be made a macro. Each of these methods have advantages and disadvantages.

The use of in-line code with the super-instruction as comments makes reading the final code difficult. The outermost routines will run over many pages, interrupted by many levels of definitions of super-instructions. When macros are used, a similar problem occurs if one attempts to read a listing which includes the expansions. If, on the other hand, one reads the macro definitions themselves, each macro is a module by itself and the code is much more understandable. The macro listings, however, do not correspond to core dumps, so debugging is often difficult without sophisticated debugging aids. By allowing the definitions to correspond to modules evoked by run time calls, the program's topography is maintained. Care must be taken, though, to assure the calling overhead does not become excessive.

By using this method of programming, the modules developed during designing are both the natural modules for coding and also the modules of documentation. By limiting each module to about 50 lines (one page), one not only helps such typographical aid as the indention of control structures but also limits the breadth of the activity of the module to a reasonable size, improving the overall intelligibility of the program.

Designing a large program from the top down is not all new; nor is the breaking of code into modules. Such techniques have been used under the name "modularity" for some time. The extension here is to break up the modules by stepwise refinement and code them in the same fashion. In addition, the modularity is carried down to much lower levels. The requirements ("what") of each module are well-defined and the method by which these requirements are fulfilled (the "how") is limited in detail to about a page.

This top-down approach may be used in the coding and testing phase as well as the design phase. The highest level modules are written first and are tested by providing dummy versions of the super-instructions evoked. These dummy "stubs," as they are called, are then replaced with the code necessary to perform the required function. New stubs are inserted for any new super-instructions evoked but not yet written. By writing the code in this top-down fashion, most of the interfacing among modules is designed early and errors are exposed before much effort is lost in incompatabilities. In addition, an attempt is made to keep communications along well-defined paths; *i.e.* instead of coding data references arbitrarily throughout the program, interfacing is done only between a module and the modules it calls directly. Such a communication discipline makes modules more independent, providing easier debugging, easier maintenance, and a simplified interface for later replacement of modules by different algorithms for the same function. When making changes (whether to fix bugs, change subfunctions, or change algorithms), one searches down the hierarchy to the highest level module, say M, at which the change is no longer transparent. Since typically many levels exist where the change is transparent, much of the code need never be considered during the change. Module M and its descendants are then discarded, redesigned, and rewritten, at least in theory. In practice, many of the same functions will still probably be required, so the modules providing those functions may often be retained virtually unmodified. Other functions may be close enough to the discarded modules to allow simple modification or adaptation. In short, the "rewrite" spoken of above is often not much more than one would need to change in a conventional look-around-and-change-whatever-is-necessary fashion; but the scope of the change is more well-defined and the module independence both simplifies the task and yields a higher confidence that all necessary changes have been made.

A number of the above techniques were developed or refined by Mills and Baker of IBM in connection with the New York Times Information Bank program.[1]

[1] Baker, F. T., "Chief Programmer Team Management of Production Programming," *IBM Systems J*, 11:1, pp. 56-73, 1972.

EFFICIENCY OF STRUCTURED CODE

There is some concern about the efficiency of structured code, and rightly so. Efficiency is an important consideration and is often one of the primary benchmarks used in deciding among programming languages. Although it is true that carefully customized control structures can often save a few branch instructions or test instructions over using the limited set provided by goto-less languages and that a program with no call statements saves linkage overhead compared with its modular equivalent, the structured programming techniques provide a number of opportunities for efficiency, some of which are not available in conventional programs.

Compilers for the few high-level languages which have been designed for doing structured programming have already begun to reap some efficiency benefits. In ALGOL and PL/I, it is possible to goto out of a procedure into some other active block. When such a goto is executed, variables local to the procedure (in PL/I, only those with the AUTOMATIC attribute) must have their storage freed. This requires extra overhead (even if such gotos never occur) which is not necessary in goto-less languages.

Conventional programs which are a rat's nest of control flow are not only hard for humans to understand; compilers often get confused, too. Major optimizers (such as FORTRAN H) must spend much time figuring out the structure which is implicit in goto-less languages. And most compilers must finally give up when they cannot resolve the flow into standard constructs. Loops which don't look like loops can't be optimized like loops. But since standard control structures are always headed by the appropriate keyword in structured programming languages, optimizers can always recognize them and therefore produce more efficient results.

Structured programming often results in many procedures which are each called from only one place and frequently have no formal parameters. High-level language compilers can easily expand such calls as in-line routines to bypass calling execution overhead.

These automatic methods are not available to the assembly language programmer (at least not with most current assemblers), but other techniques are useful. Various studies have indicated that, for most programs, the overwhelming majority of the execution time is spent in a relatively small part of the code. This fact can be exploited as follows. One writes a program in a structured fashion. Then, once the code is debugged, timing estimates (or at least module execution counts) are obtained to locate the critical sections of the code. These sections may then be optimized using various techniques including

the addition of customized control structures constructed from gotos and the in-line expansion of modules.

Other considerations also point to increased efficiency in structured programs. Since such programs tend to have many fewer errors, saving in debug time can be applied to optimization efforts. The greater intellectual manageability of structured programs may lead to the use of better algorithms. The ease of modifiability opens the door for replacing entire algorithms within working programs with a minimum of problems. Techniques such as these can make major increases in program efficiency and can make up for a myriad of redundant tests or occurrences of branch-to-a-branch.

Finally we should realize that even a certain loss in efficiency would be an acceptable cost for greatly increased program reliability. For no matter how efficient, a program with bugs doesn't really solve the problem it was supposed to solve. And a good deal of computer time can be wasted when even a simple bug requires a rerun of the program.

INTRODUCTION TO STRCMACS

In Appendix A, each of the STRCMACS macros is listed, along with a complete discussion of its possible operands. The pages which follow are intended to provide an informal introduction to the use of the STRCMACS in a tutorial manner.

The STRCMACS are used to provide the basic control structures which replace the use of branch instructions and to provide aids for doing stepwise refinement programming. As noted in the discussion of goto-less programming, three things must be provided: a method of grouping statements into units, a decision structure, and an iteration structure. The STRCMACS provide each of these as well as some additional "convenience" macros to simplify conceptualization and coding.

Defining blocks

The instruction grouping capability is provided by the defining of "blocks" of code. Such blocks are delineated by coding a block-initiating macro before the first instruction of the block and a block-terminating macro after the last instruction of the block. The simplest block defining macros are the BLOCK/BLEND pair. For example, the following block is a unit whose purpose is to increment the integer WORD:

```
BUMP    BLOCK
          L       1,WORD
          LA      1,1(1)
          ST      1,WORD
        BLEND BUMP
```

An optional block name may be specified on the block-initiating macro. Since there are no branch instructions in goto-less programs, the name field "BUMP" is basically a comment. If a name is provided, it will appear in the cross-reference table of the assembly.

A number of other macros also define a block. For example, the IF macro below not only tests the indicated condition, but initiates a block definition; the FI macro terminates the block.

```
TRY     IF    (LTR,3,3,Z)
          L       1,WORD
          LA      1,1(1)
          ST      1,WORD
        FI    TRY
```

More will be said about the IF and FI macros later.

A block may contain machine instructions, evocation of subroutines, OS or user macros, or other blocks. Coding one block inside another is called nesting. In the following example, block B is nested inside of block A, and block C is nested inside of both A and B.

```
A       BLOCK
          L       1,WORD
B         IF      (LTR,3,3,Z)
            A       1,INCR
C           IF      (LTR,4,4,NZ)
              S       1,FUDGE
            FI      C
          FI      B
          BAL     14,XYZSUB
        BLEND A
```

We will now define a few terms which will be useful in discussing nested blocks. The *current nest level* of any statement in a program is the number of block initiating macros (that is, macros which start blocks such as BLOCK and IF) up to and including the given statement minus the number of block termination macros. In the code segment above, if no blocks are defined in the program before that segment, the "A BLOCK" macro is at a current nest level of 1; the "S 1,FUDGE" instruction is at a current nest level of 3. The current nest level of a block is the current nest level of the macro initiating the block. The current nest level of block B above is 2.

A block X *surrounds* a block Y if X is initiated before and terminated after block Y. X *immediately surrounds* Y if X surrounds Y and there is no block Z such that X surrounds Z and Z surrounds Y. A block X is *properly nested* if it is terminated before the termination of any block which was initiated before the initiation of X. A program is properly nested if all its blocks are properly nested. At any point in the program, the *current block* is that block most recently initiated which has not yet been terminated.

Using the above definitions the following statements can be easily verified. If block X surrounds block Y, the current nest level of X will be less than the current nest level of Y. If X immediately surrounds Y, the current nest level of X will be exactly one less than the current nest level of Y. In a properly nested program, block termination macros always terminate the current block.

The structured macros are used to define properly nested programs. Error messages occur if a block terminating macro is issued for other than the current block. If no block name is coded as the operand of a block terminating

macro, the current block is assumed. Blocks may be nested up to some depth which is built into the macros. As distributed, this depth is 100.

Decision making

As shown in the previous section, an IF macro is provided to make conditional tests. So far we have shown IF macros with operands which were simple conditionals, such as:

```
(LTR,3,3,Z)
```

The first three operands in the list give an instruction to be executed to set the condition code. The fourth operand specifies the mnemonic (from the extended branch mnemonic BZ) for the block which follows to be executed. Hence the code:

```
TRY     IF      (LTR,3,3,Z)
          L       1,WORD
          LA      1,1(1)
          ST      1,WORD
        FI      TRY
```

will increment the fullword WORD by one if register 3 is zero. The conditional may also be given in two other equivalent forms:

```
(LTR,3,3,Z)
(LTR,3,3,REL=Z)
(LTR,3,3,MASK=8)      [The mask of a BC instruction.]
```

Note again that the mask or relation specified is that for *executing* the block. The code generated for the above simple conditionals is actually:

```
LTR   3,3                    LTR   3,3
BNZ   end-of-block    or     BC    7,end-of-block
```

Any valid machine operation code (other than branching instructions) may be specified followed by the relation or mask. *E.g.*

```
IF   (TS,SPOT,MASK=8)
IF   (CR,3,4,E)
IF   (CLM,3,X'C',BYTE,REL=E)
```

The following relations may be used:

H or GT	(High)	N H or LE	(Not High)
L or LT	(Low)	N L or GE	(Not Low)
E or EQ	(Equal)	N E	(Not Equal)
O	(Ones or Overflow)	N O	(Not Ones or Not Overflow)
P	(Plus)	N P	(Not Plus)
M	(Minus)	NM	(Not Minus)
Z	(Zero or Zeros)	N Z	(Not Zero or Not Zeros)

The FI macro terminates the conditional block. The keyword FI has been used in a number of recent languages (most notably ALGOL 68) to mean "the end of an IF block" and is a convenient specific delimiter. For those who prefer, the macros IFEND or BLEND may be used in place of FI. (BLEND may, in fact, be used to terminate any block.)

If the condition code has already been set, it can be tested by coding only the branch mnemonic or mask, as:

```
IF    (MASK=X'C')
  SR    3,5
FI
```

which subtracts register 5 from 3 if the condition code is either zero or one.

Simple conditionals may be joined by ANDs or ORs to make more complex conditional expressions. For example,

```
QTEST      IF        (LTR,5,5,Z),OR,(CH,3,HWORD,NE)
             L         7,SPOT
MORECHK      IF        <,(CR,7,5,E),OR,(SR,3,1,Z),>,AND,(LTR,1,1,
                         MASK=SYMMASK)
               L         1,WORD
               LA        1,1(1)
               ST        1,WORD
             FI        MORECHK
             A         7,WORD
           FI        QTEST
```

The entire QTEST block is bypassed unless either register 5 is zero or register 3 differs from the halfword at HWORD. If the QTEST block is executed, another conditional expression is evaluated at MORECHK. Note the use of angle brackets to group operands. These must be coded as separate macro operands—*i.e.*,

<(CR,7,5,E),OR,(SR,3,1,Z)> is invalid. The symbols "+" and "/" may be used instead of "<" and ">" for those installations whose print chains will not print the latter. If brackets are omitted, the OR is treated as having higher precedence than the AND. (If the brackets were omitted in MORECHK above, the operation would be performed as "(CR,7,5,E),OR,<,(SR,3,1,Z),AND, (LTR,1,1,MASK=SYMMASK),>".) Instructions which do more than just set the condition code (such as the SR above) may be used within conditional expressions. It should be realized, however, that such operations may not always be executed. In the MORECHK block above, register 1 will not be subtracted from register 3 if registers 7 and 5 are equal.

An ELSE macro is provided to define a block which is to be executed if and only if the preceding IF block fails. The ELSE macro terminates the IF's true block and initiates the IF's false block.

```
LIMIT    IF          (C,7,=F'100',H),ELSE=TRY0
          L            7,=F'100'
TRY0     ELSE         BLEND=LIMIT
           IF           (LTR,7,7,M)
             SR           7,7
           FI
         FI           TRY0
```

The above block limits the value of register 7 to an integer between 0 and 100. Here, as before, the block name LIMIT and TRY0 are optional as are the ELSE=TRY0 and BLEND=LIMIT operands. They may be coded to cause the macros to do checks to insure that a FI has not been accidentally added or omitted. Note that a FI for a block headed by an ELSE macro must either specify the else-block name or have a blank operand field.

A special form of the IF is provided to handle asynchronous branch points, particularly for the EODAD-point of data sets. The following illustrates a typical use of this form:

```
         GET  (IN,CARDAREA)
         IF   ASYNCH
INPUTEND   OI   FLAG,EOF
         FI
         .
         .
         .
IN       DCB  ...,EODAD=INPUTEND
```

The asynchronous IF generates an unconditional branch around the block. Note that if a label occurs on the IF macro, it will be defined on the branch instruction. As a consequence, the label "INPUTEND" is specified on the first instruction inside the block rather than on the IF macro itself.

Iteration

Iteration is provided in the STRCMACS by the DO macro. A conditional expression is specified similar to that in the IF macro, following a keyword such as "WHILE". With the WHILE keyword, the block is executed if the condition is true and execution is repeated as long as the condition remains true. For example:

```
DO   WHILE,(TM,0(5),X'80',Z)
 L       5,0(5)
OD
```

follows a chain of pointers until one is found with the high-order bit on. If the keyword "UNTIL" is the first operand, the block is always executed once, and execution continues until the conditional expression becomes true.

```
SEARCH      DO      UNTIL,(CLC,A,ARG,REL=E),OR,(TM,FLAG,EOF,O)
              GET   (IN,A)
              IF    ASYNCH
INPUTEND        OI      FLAG,EOF
              FI
            OD            SEARCH
```

The above code always reads at least one record. Records continue to be read until the value read is the same as the value in ARG or end of file is reached. Logically, the UNTIL test occurs at the end of the loop SEARCH.

Both WHILE and UNTIL tests may be provided. In the previous example, we wished for the end of file test to occur before the first loop execution, we could code:

```
DO      WHILE,(TM,FLAG,EOF,Z),AND,UNTIL,(CLC,A,ARG,EQ)
```

The WHILE and UNTIL tests may be coded in either order and may be separated by either "OR" or "AND".

The 360/370 provides three instructions which are particularly well suited for the construction of loops: BXH, BXLE, and BCT. Use of these looping

branches is provided for in the DO macro either in place of or in addition to conditional expressions.

```
          LA  1,1              FILL ARRAY
          LA  3,ARRAY            WITH 1's.
          LA  4,4
          LA  5,ARRAYEND
FILL1S    DO    UNTIL,(BXLE,3,4)
            ST    1,0(3)
          OD
```

Normally, looping branches are coded as UNTIL tests to place them at the logical end of the loop. Coding them as WHILE tests will cause the index to be incremented once before the first execution.

```
          LA  3,5
          DO  WHILE,(BCT,3)
              .
              .
              .
          OD
```

The above loop will execute only four times.

If both a looping branch and a conditional expression are specified following a keyword (WHILE or UNTIL), the looping branch must appear first, then either "AND" or "OR", and then the conditional expression. A DO macro may have only one looping branch (BXH, BXLE, or BCT).

```
X       DO  UNTIL,(BCT,5),OR,(LTR,4,4,Z),AND,(TM,FLAG,X'80',Z)
        OD
```

Brackets are assumed to be around the conditional expression, so the loop X will repeat until either register 5 is decremented to zero or both register 4 contains a zero and the high order bit in FLAG is off. The code generated is:

```
X       B     α
γ       LTR   4,4
        BNZ   α
        TM    FLAG,X'80'
        BZ    β
α       DS    0H
        block code
        BCT   5,γ
β       DS    0H
```

Appendix B shows the code generated for all possible combinations of DO operands.

The OD macro terminates the block. It may also be coded as DOEND or BLEND.

Multiple decisions

As was pointed out earlier, the block, if-then-else, and do-while constructs are sufficient for any programming task. Several additional macros are provided, however, for convenience in coding or conceptualizing the program. One of these is the DOCASE statement.

In its simplest form, the DOCASE statement defines the start of a block and defines an indexing variable whose value is, say, *i*. Inside the DOCASE block are some number (say *n*) of CASE blocks. The *i*th CASE block is executed and the remaining blocks are skipped.

Example:

```
UPDATE      DOCASE              REQWORD
ADD             CASE
                  .
                  .
                  .
                ESAC            ADD
REPL            CASE
                  .
                  .
                  .
                ESAC
CHANGE          CASE
                  .
                  .
                  .
                ESAC
                CASE
                  .
                  .
                  .
                ESAC
            ESACOD
```

If the word REQWORD contains a 2, the CASE block labeled REPL will be executed. If REQWORD is not a positive integer less than or equal to four, no CASE block will be executed.

One of the CASE macros (usually the last of the list) may have the operand "MISC" to indicate that it is to be executed only if no other block is appropriate

(that is, if the index is less than one or greater than n, in the form we have discussed so far). This miscellaneous block is not counted in locating the ith block. In our example, if the CASE labeled REPL had the operand MISC, then an index value of 2 would execute the CHANGE case, and any index less than 1 or greater than 3 would execute the MISC case REPL.

A number of extensions to the DOCASE are provided to increase its usefulness. Operands may be specified on the CASE macros to indicate for which values of the index they are to be selected, rather than allowing selection to occur by ordinal position number. By using this feature, multiple index values may be made to select the same CASE. Even entire ranges of operands may be made to select the same CASE.

```
        DOCASE  I
A       CASE    3,7
          .
          .
          .
        ESAC
B       CASE    0,2,8
          .
          .
          .
        ESAC
C       CASE    4,(9,13),X'1C'
          .
          .
          .
        ESAC
D       CASE    FIVE,(FOURTEEN,SIXTEEN)
          .
          .
          .
        ESAC
E       CASE
          .
          .
          .
        ESAC
      ESACOD
        .
        .
        .
```

```
FIVE        EQU     5
FOURTEEN    EQU     X'E'
SIXTEEN     EQU     FIVE+11
```

Case A will be executed if I contains either 3 or 7; case C for I of 4, 9, 10, 11, 12, 13, or 28 (=X'1C'). As indicated, values may be specified symbolically (although slower code is generated). All values must be in the range 0-4095. (Again, slower code is generated for values greater than 255.)

The index has been shown as being specified by giving its fullword address. It is also possible to specify halfword, byte, and register indexes as follows:

```
DOCASE      I       }
        or          } Fullword
DOCASE      (I,W)   }
DOCASE      (I,H)     Halfword
DOCASE      (I,B)     One byte
DOCASE      (3)     }
        or          } Register index
DOCASE      (R3)    }
```

Note that the latter indicates the index itself (not the address of the index) is in register 3 (or whatever register R3 is equated to).

The normal expansion of the DOCASE uses a branch vector to branch to the proper CASE block. Two special operands are provided to allow better code to be generated in certain special cases:

- Code "DOCASE I,SPARSE" when the number of values specified on the CASE blocks is small compared with the range of zero to the largest value accepted. By coding SPARSE, each CASE tests for the values appropriate to it and passes control to the next CASE on failure using a compare-and-branch sequence.

- Code "DOCASE I,SIMPLE" when each CASE block is for a single index value and those value are the numbers 1, 2, 3, . . . , n, for small n. By coding SIMPLE, the index is loaded into register 1 and each CASE does a BCT against register 1 to the next case. This is usually best when $n \leq 6$ (if no MISC CASE is present) or $n \leq 12$ (with a MISC CASE).

In addition, the DOCASE macro will automatically optimize for the case where all of the CASE macros specify operands which are exact multiples of 4.

Another form of the DOCASE allows the selection to be performed on the basis of character strings. The CASE macros may specify selection values in any of the ways shown:

```
DOCASE          (OPCODE,4)
    CASE  =C'ADD␣'          (Literal)
      .
      .
      .
    ESAC
    CASE  C'REPL','CHNG'    (Literal without leading "=" or "=C")
      .
      .
      .
    ESAC
    CASE  ('FIX1','FIX9'),'FIX␣'   (A range FIX1,FIX2,...,FIX9 or
      .                             the literal =C'FIX␣')
      .
      .
    ESAC
    CASE  SPECLOP,'NONE',X'00000000'   (An address containing a
      .                                 character string, "NONE",
      .                                 or the literal =X'00000000')
      .
    ESAC
ESACOD
```

Yet another form of the DOCASE allows selection based on arbitrary conditional expressions.

```
DOCASE
          CASE    (LTR,3,3,Z)
            .
            .
            .
          ESAC
          CASE    (CR,1,2,EQ),OR,(TM,FLAG,X'80',O)
            .
            .
            .
          ESAC
          CASE    (S,5,WORD,P)
            .
            .
            .
          ESAC
ESACOD
```

The conditional expressions are evaluated until one is found that is true. That case block is then executed and the rest are bypassed. Note that no index is specified.

Any of the previous special forms may include a miscellaneous case.

One other pair of options is provided which is of use mainly when the DOCASE is implemented by a branch vector (that is, when an index is specified, neither SPARSE nor SIMPLE is specified, and one or more CASEs are for self-defining terms in the range 0-255) and no miscellaneous case is present, although it may be specified in any DOCASE. The options IFANY or ONLY may be coded as the second or third operand. When IFANY is specified, code is included to bypass all CASE blocks if the index is out of the range of the branch vector. ("Do case I, *if any* such case exists; else do nothing.") When ONLY is coded, the range test is not included and the result if the index takes on an out-of-range value is undefined — and invariably disasterous. ("Do case I, and *only* such cases can exist.") If neither IFANY nor ONLY is coded, the tests are generated. ONLY is invalid when a MISC CASE is present. IFANY and ONLY may be coded with the non-branch vector forms of the DOCASE, but since the test occurs automatically and entails no overhead, it will be ignored.

The ESAC macro marks the end of a CASE block; it may also be coded as CASEND or BLEND. The ESACOD macro marks the end of the entire DOCASE. It may also be coded as DOCASEND or BLEND.

Abnormal block exit

Another convenience macro is the EXIT. It causes immediate transfer to the end of some containing block. It is particularly useful in situations such as searching or making error terminations in a loop.

```
DOINFILE DO     WHILE,(TM,FLAG,EOF,Z)
           <Read a control card>
SCAN        DO    WHILE,(TM,FLAG,ENDOCARD,Z)
              :
              :
WHOOPS        IF      (CLI,DELIMITR,C',',NE)
               <Print "BAD DELIMITER" message>
                EXIT
               FI
               :
               :
             OD
         OD      DOINFILE
```

In this code segment, a delimiter other than the comma will cause abnormal termination of the control card scan loop after printing a message. Since the EXIT macro has no operand, the exit is to the end of the block containing the block containing the EXIT (the surrounding block whose nest level is one less than the current nest level; in our example, the block SCAN). Any surrounding block's name may be specified as the EXIT's operand to cause transfer to the end of that block. In our example, adding the operand DOINFILE to the EXIT would skip the rest of the input when the error occurred.

Any code immediately following the EXIT macro cannot be reached, so the EXIT is usually the last instruction of an IF block as shown. Any instructions (even other blocks) could appear in the IF block labeled WHOOPS. The case where the only instruction in the IF block is the EXIT appears so frequently that a special form is provided to simplify its coding. We could have written the IF/EXIT/FI as the single macro:

```
IF    (CLI,DELIMITR,C',',NE),EXIT=SCAN
```

or

```
IF    (CLI,DELIMITR,C',',NE),EXIT=
```

This will cause control to transfer to the end of SCAN if the delimiter is not a comma. No FI need be coded (nor may it be) since the IF block is generated only long enough to perform the exit.

One disadvantage of using an EXIT is that it is no longer possible to follow code backwards. By looking at the OD macro in our example, it is not immediately obvious that there are two possible predecessors — the last instruction of the loop and the EXIT. In order to flag such occurrences, a warning message (MNOTE, severity 0) is generated at the end of any block which is the target of an EXIT macro to indicate the presence of the unexpected predecessor.

At times, the only terminating condition for a loop will be that specified by an EXIT macro. In such cases, the DO can be specified as

```
DO FOREVER
```

or just

```
DO
```

This will cause an infinite loop to be generated which can be terminated only by the inclusion of an EXIT.

Another situation which frequently occurs in search situations is that two blocks exist, one of which is to be performed if the search is successful, another if it is not. Using only the macros we have discussed so far, we could code this in the style of the block shown below, which updates the count in an identifier table if the required entry is present, otherwise it adds a new entry.

```
UPDATE  BLOCK
          LA          1,1                          Put a 1 into register 1.
          L           2,IDTAB                      Point to first entry.
SEARCH    DO          WHILE,(CLI,0(2),X'00',NE)    Null entry indicates
                                                   table end.
            IF          (CLC,ARG(8),0(2),EQ)       If entry matches ARG:
              A           1,8(2)                     Increment
              ST          1,8(2)                        count.
              EXIT        UPDATE                     Break out of block.
            FI
            L           2,12(2)                      Advance to next entry.
          OD
          MVC         0(8,2),ARG                   Argument not in table;
          ST          1,8(2)                       add it at end with count
                                                   of 1.
        BLEND       UPDATE
```

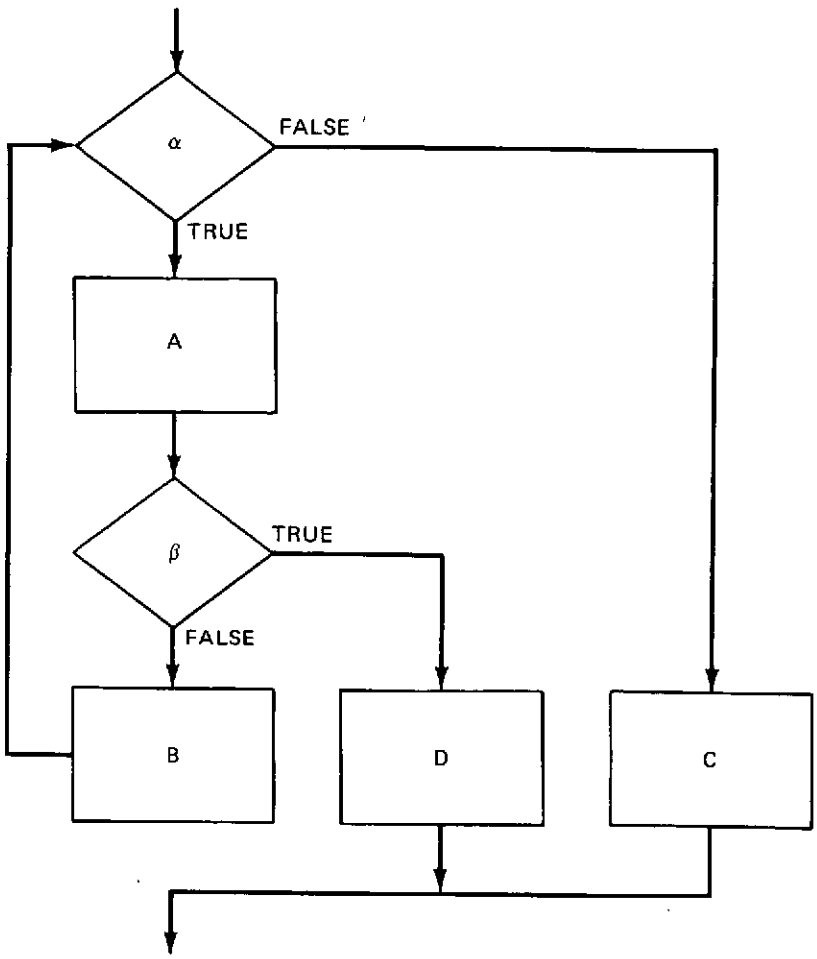
The BLOCK UPDATE is defined strictly to allow the EXIT to occur properly. An alternative form is produced by using the ATEND and ONEXIT macros:

```
          LA  1,1                                    Put a 1 into register
                                                     1.
          L   2,IDTAB                                Point to first entry.
SEARCH    DO  WHILE(CLI,0(2),X'00',NE)               For all entries in
                                                     table:
            IF  (CLC,ARG(8),0(2),EQ),EXIT=SEARCH     Exit on hit.
            L   2,12(2)                              Advance to next entry.
          ATEND                                    Not found:
            MVC 0(8,2),ARG                           Add new entry.
            ST  1,8(2)                               Set count to one.
          ONEXIT                                   Found:
            A   1,8(2)                               Bump
            ST  1,8(2)                                  count.
          OD
```

The looping segment of the block is the IF and load instructions which follow the DO. If the loop terminated normally, (that is, because of the DO macro's

test), the ATEND code segment will be executed; if the loop terminates abnormally (due to some EXIT being executed for the DO), the ONEXIT segment will be executed. The flow chart below shows the relationship of the various blocks.

```
LOOP DO WHILE, α
    <A>
      IF β, EXIT = LOOP
    <B>
    ATEND
    <C>
    ONEXIT
    <D>
    OD
```



The ATEND and ONEXIT may be coded in either order. Each is optional. The name of the active DO block may be specified as an operand of either or both as a check. ONEND may be used in place of ATEND; ATEXIT may be used in place of ONEXIT.

Defining modules

To aid stepwise refinement, it is desirable to have a simple method for defining modules which entails a minimum of execution overhead and provides a maximum of module independence. Such modules normally are called procs (for procedure), involve about a page of code, and invoke other procs via a calling sequence.

The macros PROC and CORP are provided to delineate such modules. There are two types of PROCs: the normal type involves minimal overhead (normally just the saving and restoring of registers) and is used for the majority of modules created during the stepwise refinements; the other involves standard OS linkage conventions and is usually used for the main proc of the CSECT or other places where the evoking routine is expecting OS linkage.

The simplest non-OS proc is coded as:

```
X    PROC
       .
       .
       .
     CORP
```

The PROC macro saves registers 14 through 12 (that is, registers 14, 15, and 0 through 12) in an in-line save area of fifteen words and branches around the area. The CORP restores 14 through 12 and branches to the address in register 14. Evoking the routine is accomplished by a simple

```
BAL  14,X
```

instruction.

The basic form of the OS-linkage proc is:

```
Y    PROC    LINKAGE=OS
       .
       .
       .
     CORP
```

The PROC macro now generates code similar to the IBM macro SAVE. The assumption is made that register 15 is pointing at the PROC macro. A branch is made around an in-line identifier which is taken from the label field of the PROC. Registers 14 through 12 are saved in the previous save area, pointed to by register 13. A new 18 word in-line save area is provided and chained to the previous save area. A "USING" is issued for register 13 to allow it to be used as a base register for the module's code as well as a pointer to the current save area. Register 1 is not modified by the macro. The corresponding CORP restores register 13 to point to the previous save area, restores all the registers except 15 which is set to zero as a return code, stores X'FF' as the high-order byte of word four of the old save area, and branches to the address in register 14. Evoking the OS proc may be accomplished by using the IBM CALL macro.

A number of operands are provided on the PROC and CORP macros to extend or modify these basic capabilities for both OS and non-OS procs, although it is expected that these defaults will often suffice.

If a proc (particularly a non-OS proc) modifies no registers or if registers are expected to be volatile across the proc's call, coding SAVE=NONE as a PROC operand will omit register saving and restoring. A register or range of registers may also be coded as

```
SAVE=3
```

or

```
SAVE=(15,7)
```

to cause limited saving and a correspondingly smaller save area. These registers (as all registers specified for the STRCMACS) may be specified symbolically. For example:

```
        PROC    SAVE=(R5,LAST)
          .
          .
          .
R5      EQU     5
LAST    EQU     9
```

The range must be a sub-sequence of 14 through 12 (that is, specifications such as SAVE=(0,15) are invalid).

Normally, all the registers saved will be restored by the CORP macro. The restore can be limited to a sub-sequence of those saved by coding:

```
CORP  RESTORE=(first,last)
```

or limited to a single saved register by coding

```
CORP  RESTORE=reg
```

An additional mechanism is provided to allow the specifying of a list of registers which are to be unrestored. It is often the case that the purpose of a proc (again, mainly on non-OS procs) is to calculate some result and return it in some particular register. Here restoring that register would destroy the returning value.

```
X     PROC
        .
        .
        .
      CORP  RETURN=(2,7,9)
```

All the registers (except 13) are saved by the PROC macro in this example. All the registers except 2, 7, and 9 are restored by the CORP; 2, 7, and 9 will be returned containing the values calculated by the PROC. The registers specified by the RETURN= operand must be registers which would have otherwise returned. For example, in

```
CORP  RESTORE=(2,7),RETURN=(5,9)
```

register 9 need not and must not be specified as a returning register, since it is not among those indicated to be restored. The specification of 5 is proper.

Two other suboperands of SAVE= are provided for OS-linkage procs to specify how the new save area is to be provided. The examples up to now have all used an in-line save area which is generated by default. If the user wishes to provide his own save area, he may do so by coding its label as the third suboperand of the SAVE:

```
Z      PROC    LINKAGE=OS,SAVE=(,,MYSAVE)

PROC1  PROC    LINKAGE=OS,SAVE=(14,2,MYSAVE)
```

The user's save area is assumed to be addressable by the base registers indicated by the BASE= operand, to be discussed below.

If the proc is to be reentrant or recursive, a dynamic save area is required. To specify this, code

```
SAVE=(,,DYNAM)
```

A GETMAIN will be issued for the save area and the corresponding FREEMAIN will be issued by the CORP.

By coding SAVE=(,,NONE), the user requests that the registers be saved in the old OS save area, but that no new save area be obtained.

OS-linkage save areas are normally 18 words long. To specify another size, give the length (in words) as the fourth suboperand of SAVE, either as a decimal integer or symbolically. (For in-line save areas, the symbolic length must be a previously defined symbol.) A typical use for a reentrant program is:

```
RENTPROC    PROC     LINKAGE=OS,SAVE=(,,DYNAM,WORKSIZE)
            USING     WORKSECT,13
              .
              .
              .
```

```
WORKSECT    DSECT
            DS    18F          New save area.
              .
              .                Other work variables.
              .
WORKSIZE    EQU   (*-WORKSECT+3)/4, Length, in words, rounded up.
```

This obtains core for the dummy section WORKSECT and provides addressability. Dynamic save areas cannot be specified for non-OS linkage procs; but since in-line save areas are generated by default, SAVE=NONE must be specified on all non-OS procs within reentrant or recursive code.

The in-line identifier generated for OS-linkage procs containing the proc's name may be modified by using the ID= keyword of the PROC macro. By coding ID=NONE, the identifier (and the branch around it) will not be generated. By coding ID=* on a non-OS proc, the proc name will be generated as for OS procs. A character string other than the proc name may be specified for either type of proc by coding

ID=*char-string*

Surrounding quotes may be specified on the character string where macro syntax requires (as when the string contains blanks or commas).

A base register is provided by default for OS procs. With the normal in-line save area, register 13 serves this function. If the user provides his own save area or requests a dynamic one, register 12 is the default base register. To specify the loading of a base register other than the default (or to request a base register load for non-OS procs), use the BASE= keyword, as:

```
PROC  BASE=7
```

In this case, register 7 will be loaded and a USING will be issued. Multiple base registers may also be specified. For example,

```
PROC  BASE=(7,8,9)
```

will cause register 7 to be loaded with an address within the macro, 8 to be loaded with that address plus 4096, 9 with that address plus 8192, and a USING will be issued for the three registers. By omitting the first register, the default register will be used as the first base register. For example:

```
PROC  LINKAGE=OS,BASE=(,10,9)
```

will use 13 as the first base register and 10 and 9 as the second and third. Register 13 should not be explicitly listed as an operand of BASE=.

To bypass base register loading for OS procs, use BASE=NONE.

Although a USING is issued for each base register, no DROPs are issued during the corresponding CORP. It is the user's responsibility to be sure DROPs are issued at such times as are necessary to prevent invalid code. In most cases, this only requires providing total addressability at the entry to the main proc and never changing or DROPping any base registers.

The main proc of an assembly usually is the first proc and uses OS linkage. By coding

```
X       PROC  LINKAGE=(OS,CSECT)
```

a CSECT pseudo-operation is generated with the name X. LINKAGE=(,CSECT) may be used to define a non-OS proc as a CSECT, if desired. Following the CSECT pseudo-op, a "USING *, 15" is also generated to provide addressability during the macro. A "DROP 15" is generated at the end of the PROC macro. If the CSECT operand is not specified, the user is expected to provide addressability and have a valid outstanding USING instruction.

The STRCMACS, like any macros, must use certain registers as work registers. Normally, only registers 0 and 1 are vulnerable to destruction by the STRCMACS. For OS-linkage procs, however, register 1 is typically used to point to a parameter list. As a result, register 2 is used as a second work register. The user may specify that some other register be used as a work register in place of the default (register 2 for OS procs, register 1 for non-OS procs) by coding

```
PROC  WORK=5
```

or the like. By using WORK=NONE, the default will be used, but will be restored in the code generated by the PROC macro. In any case, register 0 is still volatile.

Register 15 is loaded with a zero by default in the CORP expansion of all OS procs. To specify a different return code (or any return code for non-OS procs), use:

```
CORP  RC=value
```

If the value to be returned is contained in a register, use:

```
CORP  RC=(reg)
```

By coding RC=NONE, no special return-code processing is performed; the value returned in register 15 will be determined by whether it is being restored, as for any other register.

The last instruction normally generated by a CORP is a

```
BR  14
```

to return to the address in register 14. To cause a different register to be used for the subroutine linkage, use:

```
CORP  LINK=linkreg
```

By coding LINK=NONE, the returning branch will be omitted and control will fall out the bottom of the macro.

This allows two methods of proc linkage. The normal method is to use the standard execution-time linkage:

```
A   PROC    LINKAGE=(OS,CSECT)
      .
      .
      .
    BAL     14,B
      .
      .
      .
    CORP    A

B   PROC
      .
      .
      .
    BAL     14,C
      .
      .
      .
    CORP
```

```
C    PROC
       .
       .
       .
     CORP
```

The alternate method is to define the procs as user macros to perform the linkage at assembly time:

```
     MACRO
     BMAC
B    PROC
       .
       .
       .
     CMAC                                        (*)
       .
       .
       .
     CORP    B,LINK=NONE
     MEND

     MACRO
     CMAC
     PROC
       .
       .
       .
     CORP    LINK=NONE
     MEND

A    PROC    LINKAGE=(OS,CSECT)
       .
       .
       .
     BMAC                                        (**)
       .
       .
       .
     CORP
```

This causes the macro BMAC to be expanded at the point (**). During that expansion, the macro CMAC is evoked when line (*) is generated. Since LINK=NONE is specified on the macros' CORPs, control falls out the bottom of each macro.

The macros PROCEND and BLEND may be used in place of CORP.

Special services

Two minor services are provided by the STRCMACS which may be useful from time to time.

As was pointed out earlier, any block-terminating macro which is the target of an EXIT receives a message warning of the unexpected predecessor instruction. This message normally receives a severity code of 0. It therefore does not affect the execution of later job steps (such as linkage editing), but a reference to the message does appear in the list of diagnostic messages. The user may change the severity of the EXIT message by coding.

PROC EXIT=*severity*

on any PROC. All EXIT messages thereafter will receive the indicated severity code. The severity must be specified as either an integer from 0 to 4095 or as an *, (the latter avoiding the reference to the message in the diagnostic message list).

The macro FINAL may be coded after all other code to provide a check that all blocks have been terminated. This use of the FINAL macro is optional. Another use is described in the next section.

STRCMACS debugging aids

A number of debugging aids have been designed into the structured macros. Although some of the options exact fairly heavy penalties in memory or execution time requirements, the ease with which the debug options may be turned on and off allow large amounts of execution information to be gathered with a minimum of programmer effort for the isolation of any given bug.

The various options may be specified on any PROC macro by coding:

PROC DEBUG=(*list of options*)

In the list, one can specify that various options be turned on (or off); the indicated options will then be on (or off) for the duration of the proc. At the CORP, the status of the options will revert to their status before the PROC macro. To avoid this restoration, one may code "GLOBAL" or "GBL" in the list of options. One may also code "ALL" or "NONE" as options indicating that all options are

to be turned on or off, respectively. After the ALL or NONE, exceptions may be listed. For example:

```
A     PROC
        .
        .
        .
      CORP
B     PROC    DEBUG=(BLOCKNAMES,PROCTRACE,GBL)
        .
        .
        .
      CORP    B
C     PROC    DEBUG=(NOPROCTRACE,PROCCOUNTS)
        .
        .
        .
      CORP    C
D     PROC    DEBUG=(ALL,NOSAVETRACE)
        .
        .
        .
      CORP
```

In the above code, proc A requests no debug processing; all debug options remain off. Proc B turns on block-names and proc-tracing (discussed below), and specifies that the CORP B is not to revert the options to their former state (all off). Proc C turns off proc-tracing and turns on proc-counting. At the CORP C, the options revert to those specified in proc B. Proc D turns on all options except the save-trace.

We will now discuss each of the options in turn.

The LISTBLOCKS option causes the name, sequential number, and static nesting depth of each block to be printed on the assembly source listing as comment messages (severity "*") at the beginning and end of each block.

The PROCNAMES options forces all proc names to be generated as in-line character constants as though ID=* had been coded on every PROC macro. These names make it easy to find the corresponding code quickly in dumps. The process can be carried a step further; by turning on the BLOCKNAMES option, all blocks will contain such in-line identifiers. This is mainly of use with the BLOCKCOUNTS option.

The PROCCOUNTS and BLOCKCOUNTS options cause various statistics to be maintained on the execution of proc blocks or all blocks, respectively. The statistics maintained are:

- On PROCs—The number of times the proc has been executed. This count is kept if either PROCCOUNTS or BLOCKCOUNTS is specified.
- On IFs—The number of times the condition was evaluated as true.
- On DOs—The number of times the loop body has been executed during the run (the overall loop count) and the number of times the loop body has been executed since the DO was most recently entered (the current loop count).
- On DOCASEs—The ordinal number of the last nonmiscellaneous case executed; note that this is not necessarily the value of the most recent index. If the most recent execution caused the miscellaneous case to occur, the value 255 (X'FF') is stored.
- On CASEs—The number of times this case has been executed.
- On BLOCKs—The number of times the block has been executed.

If both BLOCKNAMES and BLOCKCOUNTS are coded, the counts are stored immediately following the block names* to aid locating them in dumps.

By coding the option PROCTRACE, a record of the last 257 procs executed is maintained. The record is kept as a 258-byte vector of one-byte binary numbers. (The 258th byte is not used; it always has the value X'FF'.) As each proc is entered, the vector is shifted one to the left and the proc's identifying number is stored in the 257th byte. The proc's identifying number appears not only in the instruction which stores it into the vector, but also in all labels generated by the PROC and CORP macros when PROCTRACE is turned on. These labels are of the form "$P*hhxxx*" where the *hh* is the proc's identifying number (in hex) and *xxx* varies with the particular label. The vector itself appears as:

```
                DC   C'$TRACE'
$TRACE          DC   258X'FF'
```

and is generated in the first proc which requests PROCTRACE.

A free piece of debugging information is provided by the in-line save area of the non-OS procs. The values in all registers specified in the SAVE=operand (or by default, all registers) are stored in this area. During the CORP, any registers specified in the RETURN=operand (and register 15, if a return code is provided) are individually stored into the PROC's save area. Then the range

*An exception to this is proc counts, for reasons which will be discussed later.

of registers indicated by the RESTORE= operand (or all the saved registers, by default) are reloaded from the PROC's save area. As a result, the save area will contain the registers on entry to the proc or those being returned by the proc or some mixture depending on whether the dump occurred before, after, or during CORP register restoring.

By coding the debug option CORPVALUES, additional save areas are provided. In addition to the PROC's main save area, a save area is generated by the CORP macro (called the CORPVALUES save area) and all the registers (14 through 12) are stored before doing register restoring to provide a copy of the values calculated by the proc. If one or more registers are to be returned (either by being listed in the RETURN= operand or because the RC= operand was specified), a third save area (called the BACK save area) is provided. The PROC's main save area is copied to the BACK save area and the value to be returned in the RETURN= registers (and in 15, for RC=) are stored into it before loading all the registers in the RESTORE= range. Hence, the PROC's main save area contains the values in the registers the last time the proc was evoked, the CORPVALUES save area contains the values in the registers before register restoring the last time the proc completed processing, and the BACK save area contains the values returned to its caller (if different from the values saved at proc entry).

These various save areas provide a wealth of information, but locating particular values can be a painstaking and somewhat error-prone process. A final debug option provides the mechanism for having these areas formatted automatically in OS dumps. To request the formatting, the first proc must be an OS-linkage proc and the SAVETRACE debugging option must be turned on in it. In addition the FINAL macro must be coded following the last proc. The SAVETRACE option causes all non-OS save areas to be generated as full 18 word save areas linked statically (that is, at assembly time) according to OS conventions. On entry to the first proc, the entire list of non-OS save areas are linked between the old (caller's) OS save area and the new save area. Since these save areas are formatted like OS save areas, they will be printed in the save area trace portion of the OS dump.

Word 1 of each non-OS save area is used to identify it. The high-order byte indicates the type of save area as follows:

X'FF' or X'FE': The PROC's main save area: The byte is initialized to X'FF'; it is set to X'FE' each time the proc is entered and is reset to X'FF' each time the proc is "finished" (each time it returns).

X'FC': The CORPVALUE's save area, for those procs in which the CORPVALUES option is turned on.

X'FB': The BACK save area for those procs in which the CORPVALUES option is turned on and in which one or more registers are returned.

Byte two of word one contains the one byte hex proc identifying number used in that proc's labels and (if PROCTRACE is turned on) for proc tracing. The last half of word one of the PROC's main save area contains the proc count (if PROCCOUNTS or BLOCKCOUNTS is turned on).

Word one of the first OS save area contains the address of the trace vector (if PROCTRACT is turned on).

The above may seem somewhat confusing, but the example on the following page should clear it up somewhat.

When OURPROG is called it evokes SUBX and SUBZ each twice. On its second execution, SUBZ evokes SUBY which calls NEXTPROG which abends. On the following pages the assembly, a diagram of the debugging blocks, and a part of the dump are shown. Note the save areas formatted in the dump and the trace vector and block counts.

It should be noted that turning on all debugging facilities can double the length of a CSECT or more. In programs in which these aids are to be used from time to time, one must be sure to set aside sufficient registers to be used as base registers to provide addressability.

Addressability, labels, and reentrant code

Care must be taken that sufficient addressability is provided by the base registers to handle references made by the structured macros. In particular, it should be noted that since literals are generated by some PROC forms and by character string CASEs, the literal pool must be addressable to these macros. In addition, CORPs must be able to address their own PROCs.

All labels generated by the STRCMACS (except those specified by users in macro name fields) begin with the "$". Users should not use such labels to avoid conflicts.

Reentrant code is generated except for in-line register saving and most of the debug aids. To bypass the former, use SAVE=(,,DYNAM) on OS procs and SAVE=NONE on non-OS procs. To bypass the latter, do not use the debug aids. (Sorry about that!)

```
 TITLE 'EXAMPLE OF DEBUG FACILITIES'
DUPROG   PROC  LINKAGE=(OS,CSECT),DEBUG=(ALL,NOLISTBLOCKS,GLOBAL)
*              :
*              :
               LA     5,2
               DO     UNTIL,(BCT,5)
                 BAL    14,SUBX
                 BAL    14,SUBZ
               OD
*              :
*              :
               CORP
               EJECT
SUBX     PROC  SAVE=(3,5)
*              :
*              :
               L      3,XID
*              :
*              :
               CORP  SUBX
               EJECT
SUBY     PROC  DEBUG=NOCORVALUES
*              :
*              :
               L      3,YID
               CALL  NEXTPROG
*              :
*              :
               CORP  RETURN=3
               EJECT
SUBZ     PROC
*              :
*              :
               L      3,ZID
               LR     6,3
               IF     (C,5,=F'1',EQ)
                 BAL    14,SUBY
               FI
*              :
*              :
               CORP  RETURN=6
               EJECT
               FINAL
               DS     0F
XID      DC     C'XXXX'
YID      DC     C'YYYY'
ZID      DC     C'ZZZZ'
               LTORG
               SPACE 3
               END
```

```
 LOC  OBJECT CODE     ADDR1 ADDR2  STMT   SOURCE STATEMENT                                                    23 JAN 74

        *** MNOTE ***                          2 OURPROG  PROC  LINKAGE=(OS,CSECT),DEBUG=(ALL,NOLISTBLOCKS,GLOBAL)      00004000
                                               3               4, STRC8103 WARNING--SAVETRACE REQUIRES "FINAL" MACRO
                                               4               *, STRC8108 PROC OURPROG, DEBUG ID=X'01'
000000                                         5+OURPROG  CSECT
000000                                         6+         USING *,15
000000 47F0 F00C              0000C            7+         B     $P01AA
000004 07D6E4D9D7D9D6C7                        8+         DC    AL1(7),CL7'OURPROG'
00000C 90EC D00C              0000C            9+$P01AA   STM   14,12,12(13)
000010                                        10+         CNOP  0,4
000010 4520 F05C              0005C           11+         BAL   2,$P01BB
000014 0000007E00000000                       12+$P01ISV  DC    A($TRACE),(18-1)F'0'
00005C 50D0 F1FC              001FC           13+$P01BB   ST    13,$FIRSTSV+4
000060 D203 D008 F4A8 00008 004A8             14+         MVC   8(4,13),=A($FIRSTSV)
000066 58D0 F4AC              004AC           15+         L     13,=A($LASTSAV)
00006A 502D 0008              00008           16+         ST    2,8(13)
00006E 50D2 0004              00004           17+         ST    13,4(2)
000072 18D2                                   18+         LR    13,2
000014                                        19+         USING $P01ISV,13
000074 47F0 D16E              00182           20+         B     $P01EE
000078 5BE3D9C1C3C5                           21+         DC    C'$TRACE'
00007E FFFFFFFFFFFFFFFF                       22+$TRACE   DC    256X'FF'
000180 0000                                   23+$P01PCT  DC    H'0' PROC COUNT
000182 D2FF D06A D06B 0007E 0007F             24+$P01EE   MVC   $TRACE(256),$TRACE+1
000188 9201 D16A      0017E                   25+         MVI   $TRACE+256,X'01'
00018C 4820 D16C              00180           26+         LH    2,$P01PCT
000190 4122 0001              00001           27+         LA    2,1(2)
000194 4020 D16C              00180           28+         STH   2,$P01PCT
                                              29+         DROP  15
                                              30 *          :                                                     00005000
                                              31 *          :                                                     00006000
000198 4150 0002              00002           32            LA    5,2                                             00007000
                                              33            DO    UNTIL,(BCT,5)                                   00008000
00019C 1B11                                   34+        SR    1,1
00019E 4010 D1AE              001C2           35+        STH   1,$2DOL
0001A2 4810 D1AE              001C2           36+$2BEG   LH    1,$2DOL
0001A6 4111 0001              00001           37+        LA    1,1(1)
0001AA 4010 D1AE              001C2           38+        STH   1,$2DOL
0001AE 4810 D1B0              001C4           39+        LH    1,$2DTR
0001B2 4111 0001              00001           40+        LA    1,1(1)
0001B6 4010 D1B0              001C4           41+        STH   1,$2DTR
0001BA 47F0 D1B2              001C6           42+        B     $2GO
0001BE C2D3D2F2                               43+        DC    C'BLK2',0H'0'
0001C2 0000                                   44+$2DOL   DC    H'0' CURRENT LOOP COUNT
0001C4 0000                                   45+$2DTR   DC    H'0' OVERALL LOOP COUNT
0001C6                                        46+$2GO    DS    0H
0001C6 45E0 D1D2              001E6           47             BAL   14,SUBX                                       00009000
0001CA 45E0 D33C              00350           48             BAL   14,SUBZ                                       00010000
                                              49           OD                                                    00011000
0001CE 4650 D18E              001A2           50+        BCT   5,$2BEG
                                              51 *          :                                                     00012000
                                              52 *          :                                                     00013000
                                              53         CORP                                                    00014000
0001D2 58D0 D1E8              001FC           54+        L     13,$FIRSTSV+4
0001D6 1BFF                                   55+        SR    15,15
0001D8 50FD 0010              00010           56+        ST    15,16(13)
0001DC 98EC D00C              0000C           57+        LM    14,12,12(13)
0001E0 92FF D00C      0000C                   58+        MVI   12(13),X'FF'
0001E4 07FE                                   59+        BR    14
```

```
  LOC  OBJECT CODE    ADDR1 ADDR2  STMT   SOURCE STATEMENT                                                    23 JAN 74
                                     61 SUBX     PROC  SAVE=(3,5)                                                    00016000
                                     62                *, STRC8108 PROC SUBX, DEBUG ID=X'02'
0001E6 47F0 D1DA            001E8    63+SUBX     B     $P02AA
0001EA E2E4C2E7                      64+         DC    CL4'SUBX'
0001EE 90EC D1F0            00204    65+$P02AA   STM   14,12,$P02SV+12
0001F2 47F0 D22C            00240    66+         B     $P02DD
0001F8                               67+$P02SV   DS    0F
0001F8 FF020000                      68+         DC    X'FF020000' FLAG (FF=FINISHED,FE=ENTERED), ID, COUNT
0001F8                               69+$FIRSTSV EQU   $P02SV
0001FC 0000000000000270              70+         DC    A(0,$P02NXT)
000204 0000000000000000              71+         DC    (15)F'0'
000240 5810 D49C            00480    72+$P02DD   L     1,=A($TRACE)
000244 D2FF 1000 1001 0000) 00001    73+         MVC   0(256,1),1(1)
00024A 9202 1100    00100            74+         MVI   256(1),X'02'
00024E 4810 D1E6            001FA    75+         LH    1,$P02SV+2
000252 4111 0001            00001    76+         LA    1,1(1)
000256 4010 D1E6            001FA    77+         STH   1,$P02SV+2
00025A 92FE D1E4      001F8          78+         MVI   $P02SV,X'FE'
                                     79 *          :                                                           00017000
                                     80 *          :                                                           00018000
00025E 5830 D484            00498    81            L     3,XID                                                 00019000
                                     82 *          :                                                           00020000
                                     83 *          :                                                           00021000
                                     84          CORP  SUBX                                                    00022000
000262 90EC D268            0027C    85+         STM   14,12,$P02CRP+12
000266 92FF D1E4      001F8          86+         MVI   $P02SV,X'FF'
00026A 9835 D204            00218    87+         LM    3,5,$P02SV+32
00026E 07FE                          88+         BR    14
000270                               89+$P02CRP  DS    0F
000270 FC020000                      90+         DC    X'FC020000'
000270                               91+$P02NXT  EQU   $P02CRP
000274 000001F8000002C8              92+         DC    A($P02SV,$P02FWD)
00027C 0000000000000000              93+         DC    15F'0'
```

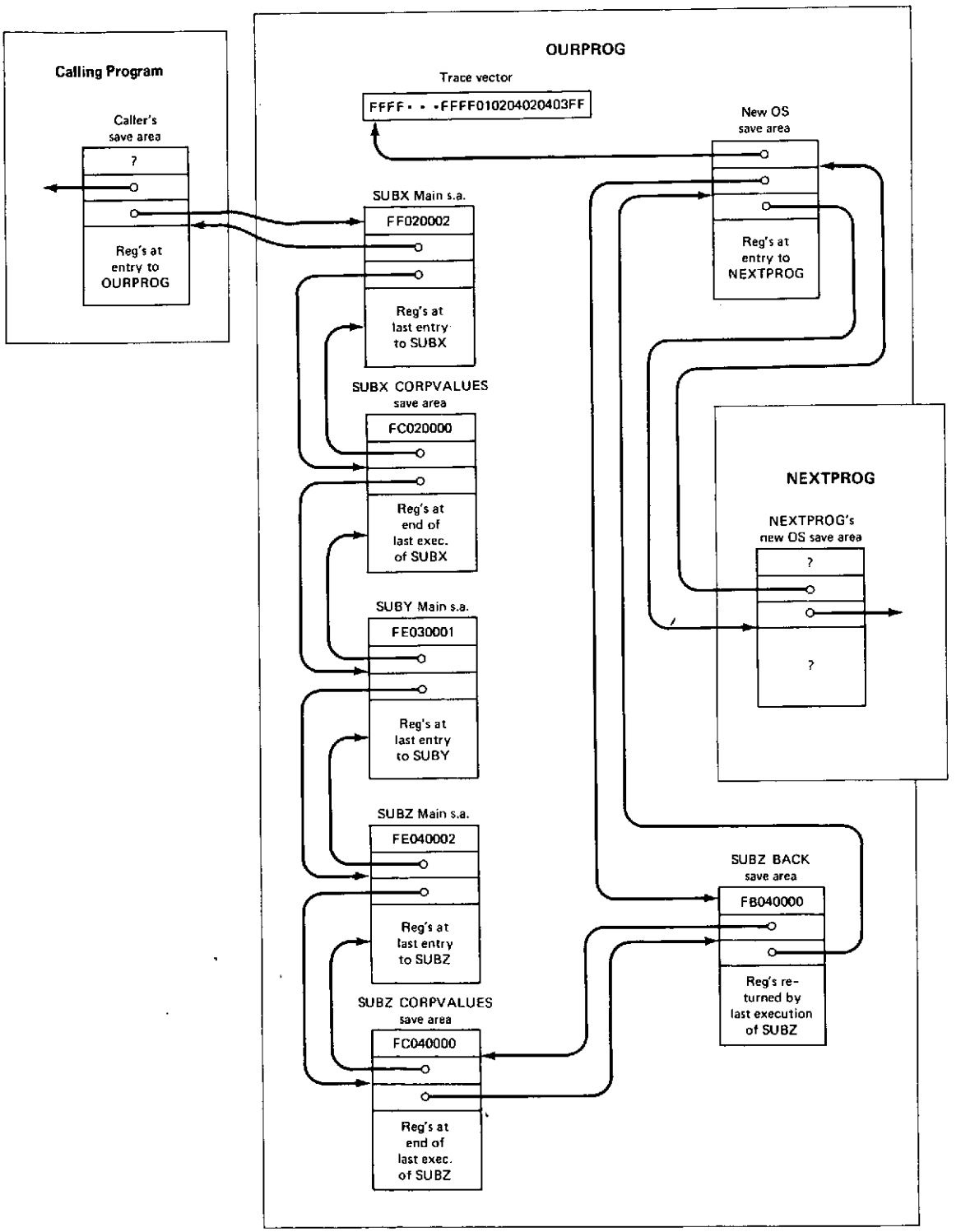
```
 LOC  OBJECT CODE    ADDR1 ADDR2  STMT   SOURCE STATEMENT                                                    23 JAN 74
                                    95 SUBY     PROC  DEBUG=NOCORPVALUES                                      00024000
                                    96                *, STRC8108 PROC SUBY, DEBUG ID=X'03'
0002B8 47F0 D2AC            02C0    97+SUBY     B     $P03AA
0002BC E2E4C2E8                     98+         DC    CL4'SUBY'
0002C0 90EC D2C0            02D4    99+$P03AA   STM   14,12,$P03SV+12
0002C4 47F0 D2FC            00310  100+         B     $P03DD
0002C8                             101+$P03SV   DS    0F
0002C8 FF030000                    102+         DC    X'FF030000' FLAG (FF=FINISHED,FE=ENTERED), ID, COUNT
0002C8                             103+$P02FWD  EQU   $P03SV
0002CC 0000027000000360            104+         DC    A($P02CRP,$P03NXT)
0002D4 0000000000000000            105+         DC    (15)F'0'
000310 5810 D49C            04B0   106+$P03DD   L     1,=A($TRACE)
000314 D2FF 1000 1001 00000 00001  107+         MVC   0(256,1),1(1)
00031A 9203 1100      00100        108+         MVI   256(1),X'03'
00031E 4810 D2B6            02CA   109+         LH    1,$P03SV+2
000322 4111 0001            00001  110+         LA    1,1(1)
000326 4010 D2B6            02CA   111+         STH   1,$P03SV+2
00032A 92FE D2B4      02C8         112+         MVI   $P03SV,X'FE'
                                   113 *           :                                                         00025000
                                   114 *           :                                                         00026000
00032E 5830 D488            049C   115           L     3,YID                                                 00027000
                                   116           CALL  NEXTPROG                                              00028000
000332 0700                        117+         CNOP  0,4
000334 47F0 D328            033C   118+         B     *+8 BRANCH AROUND VCON
000338 00000000                    119+IHB0040B DC    V(NEXTPROG) ENTRY POINT ADDRESS
00033C 58F0 D324            0338   120+         L     15,IHB0040B LOAD 15 WITH ENTRY ADR
000340 05EF                        121+         BALR  14,15 BRANCH TO ENTRY POINT
                                   122 *           :                                                         00029000
                                   123 *           :                                                         00030000
                                   124           CORP  RETURN=3                                              00031000
000342 5030 D2D4            02E8   125+         ST    3,$P03SV+32
000346 92FF D2B4      02C8         126+         MVI   $P03SV,X'FF'
00034A 98EC D2C0            02D4   127+         LM    14,12,$P03SV+12
00034E 07FE                        128+         BR    14
```

```
  LOC  OBJECT CODE     ADDR1 ADDR2  STMT   SOURCE STATEMENT                                                    00033000
                                    130 SUBZ     PROC
                                    131               *, STRC8108 PROC SUBZ, DEBUG ID=X'04'
000350 47F0 D344            00358   132+SUBZ     B     $P04AA
000354 E2E4C2E9                     133+         DC    CL4'SUBZ'
000358 90EC D359            0036C   134+$P04AA   STM   14,12,$P04SV+12
00035C 47F0 D394            003A8   135+         B     $P04DD
000360                              136+$P04SV   DS    0F
000360 FF040000                     137+         DC    X'FF040000' FLAG (FF=FINISHED,FE=ENTERED), ID, COUNT
000360                              138+$P03NXT  EQU   $P04SV
000364 000002C800000408             139+         DC    A($P03SV,$P04NXT)
00036C 0000000000000000             140+         DC    (15)F'0'
0003A8 5810 D49C            004B0   141+$P04DD   L     1,=A($TRACE)
0003AC D2FF 1000 1001 00000 00001   142+         MVC   0(256,1),1(1)
0003B2 9204 1100      00100         143+         MVI   256(1),X'04'
0003B6 4810 D34E            00362   144+         LH    1,$P04SV+2
0003BA 4111 0001            00001   145+         LA    1,1(1)
0003BE 4010 D34E            00362   146+         STH   1,$P04SV+2
0003C2 92FE D34C      00360         147+         MVI   $P04SV,X'FE'
                                    148 *          :                                                          00034000
                                    149 *          :                                                          00035000
0003C6 5830 D48C            004A0   150            L     3,2ID                                                00036000
0003CA 1863                         151            LR    6,3                                                  00037000
                                    152            IF    (C,5,=F'1',EQ)                                       00038000
0003CC 5950 D4A0            004B4   153+         C     5,=F'1'
0003D0 4770 D3DA            003EE   154+         BNE   $6END
0003D4 4810 D3D4            003E8   155+         LH    1,$6IFC
0003D8 4111 0001            00001   156+         LA    1,1(1)
0003DC 4010 D3D4            003E8   157+         STH   1,$6IFC
0003E0 47F0 D3D6            003EA   158+         B     $6GO
0003E4 C2D3D2F6                     159+         DC    C'BLK6',OH'0'
0003E8 0000                         160+$6IFC    DC    H'0' IF COUNT
0003EA                              161+$6GO     DS    0H
0003EA 45E0 D2A4            002B8   162              BAL   14,SUBY                                            00039000
                                    163            FI                                                         00040000
0003EE                              164+$6END    DS    0H
                                    165 *          :                                                          00041000
                                    166 *          :                                                          00042000
                                    167            CORP  RETURN=6                                           00043000
0003EE 90EC D400            00414   168+         STM   14,12,$P04CRP+12
0003F2 D23B D448 D358 00458 0036C   169+         MVC   $P04BCK+12(15*4),$P04SV+12
0003F8 5060 D468            0047C   170+         ST    6,$P04BCK+44
0003FC 92FF D34C      00360         171+         MVI   $P04SV,X'FF'
000400 98EC D448            0045C   172+         LM    14,12,$P04BCK+12
000404 07FE                         173+         BR    14
000408                              174+$P04CRP  DS    0F
000408 FC040000                     175+         DC    X'FC040000'
000408                              176+$P04NXT  EQU   $P04CRP
00040C 0000036000000450             177+         DC    A($P04SV,$P04BCK)
000414 0000000000000000             178+         DC    15F'0'
000450                              179+$P04BCK  DS    0F
000450 FB040000                     180+         DC    X'FB040000'
000454 0000040800000000             181+         DC    A($P04CRP,$P04FWD)
00045C 0000000000000000             182+         DC    (15)F'0'
```

```
  LOC  OBJECT CODE     ADDR1 ADDR2  STMT   SOURCE STATEMENT

000450                              184          FINAL
000000                              185+$LASTSAV EQU   $P04BCK                00045000
000498                              186+$P04FWD  EQU   0
000498 E7E7E7E7                     187          DS    0F
00049C E8E8E8E8                     188 XID      DC    C'XXXX'                00046000
0004A0 E9E9E9E9                     189 YID      DC    C'YYYY'                00047000
0004A8                              190 ZID      DC    C'ZZZZ'                00048000
0004A8 000001F8                     191          LTORG                        00049000
0004AC 00000450                     192                =A($PIRSTSV)           00050000
0004B0 0000007E                     193                =A($LASTSAV)
0004B4 00000001                     194                =A($TRACE)
                                    195                =F'1'


                                    197          END                          00052000
```


Calling Program
Caller's save area
?
Reg's at entry to OURPROG
OURPROG
Trace vector
FFFF• • •FFFF010204020403FF
New OS save area
Reg's at entry to NEXTPROG
SUBX Main s.a.
FF020002
Reg's at last entry to SUBX
SUBX CORPVALUES save area
FC020000
Reg's at end of last exec. of SUBX
SUBY Main s.a.
FE030001
Reg's at last entry to SUBY
SUBZ Main s.a.
FE040002
Reg's at last entry to SUBZ
SUBZ CORPVALUES save area
FC040000
Reg's at end of last exec. of SUBZ
NEXTPROG
NEXTPROG's new OS save area
?
?
SUBZ BACK save area
FB040000
Reg's returned by last execution of SUBZ

```
SAVE AREA TRACE

GSFC      WAS ENTERED VIA LINK          AT EP OURPROG
  Caller's Save area
SA   155768  WD1 00000000   HSA 00000000   LSA 00144CB8   RET 000184F8   EPA 01144AC0   R0  FD000008
             R1  001557F8   R2  0003DF60   R3  5C03E528   R4  0002BE58   R5  0002C2C0   R6  0002A670  Regs at entry to OURPROG
             R7  0002F148   R8  0003E500   R9  0003FFF8   R10 0003E528   R11 00000000   R12 401FE482

GSFC      WAS ENTERED VIA CALL          AT EP OURPROG   <- Ignore these; they are based on what reg 15 points at
            Finished -> FF  SUBX -> Count = 2
SA   144CB8  WD1 FF020002   HSA 00155768   LSA 00144D30   RET A0144C8A   EPA 01144AC0   R0  FD000008
             R1  00000002   R2  00000001   R3  5C03E528   R4  0002BE58   R5  00000001   R6  E9E9E9E9  Regs at entry to SUBX
             R7  0002F148   R8  0003E500   R9  0003FFF8   R10 0003E528   R11 00000000   R12 401FE482

GSFC      WAS ENTERED VIA CALL          AT EP OURPROG
        CORPVALUES -> SUBX
SA   144D30  WD1 FC020000   HSA 00144CB8   LSA 00144D88   RET A0144C8A   EPA 01144AC0   R0  FD000008
             R1  00000002   R2  00000001   R3  E7E7E7E7   R4  0002BE58   R5  00000001   R6  E9E9E9E9  Regs at CORP SUBX
             R7  0002F148   R8  0003E500   R9  0003FFF8   R10 0003E528   R11 00000000   R12 401FE482

GSFC      WAS ENTERED VIA CALL          AT EP OURPROG
            Entered -> FE  SUBY -> Count = 1
SA   144D88  WD1 FE030001   HSA 00144D30   LSA 00144E20   RET 80144EAE   EPA 01144AC0   R0  FD000008
             R1  00000001   R2  00000001   R3  E9E9E9E9   R4  0002BE58   R5  00000001   R6  E9E9E9E9  Regs saved at entry to SUBY
             R7  0002F148   R8  0003E500   R9  0003FFF8   R10 0003E528   R11 00000000   R12 401FE482  possibly modified at CORP

No CORPVALUES for SUBY                     AT EP OURPROG
            Entered -> FE  SUBZ -> Count = 2
SA   144E20  WD1 FE040002   HSA 00144D88   LSA 00144EC8   RET A0144C8E   EPA 01144AC0   R0  FD000008
             R1  00000002   R2  00000001   R3  5C03E528   R4  0002BE58   R5  00000001   R6  E9E9E9E9  Regs at entry to SUBZ
             R7  0002F148   R8  0003E500   R9  0003FFF8   R10 0003E528   R11 00000000   R12 401FE482

GSFC      WAS ENTERED VIA CALL          AT EP OURPROG
        CORPVALUES -> SUBZ
SA   144EC8  WD1 FC040000   HSA 00144E20   LSA 00144F10   RET 80144C8E   EPA 01144AC0   R0  FD000008
             R1  00000001   R2  00000001   R3  E9E9E9E9   R4  0002BE58   R5  00000002   R6  E9E9E9E9  Regs at CORP SUBZ
             R7  0002F148   R8  0003E500   R9  0003FFF8   R10 0003E528   R11 00000000   R12 401FE482

GSFC      WAS ENTERED VIA CALL          AT EP OURPROG
          BACK S.A. -> SUBZ
SA   144F10  WD1 FB040000   HSA 00144EC8   LSA 80144AD4   RET 80144C8E   EPA 01144AC0   R0  FD000008
             R1  00000001   R2  00000001   R3  5C03E528   R4  0002BE58   R5  00000002   R6  E9E9E9E9  Regs returned by SUBZ
             R7  0002F148   R8  0003E500   R9  0003FFF8   R10 0003E528   R11 00000000   R12 401FE482

GSFC      WAS ENTERED VIA CALL          AT EP NEXTPROG
  OURPROG Main S.A. -> 00144B3E <- Address of trace vector
SA   144AD4  WD1 00144B3E   HSA 00144F10   LSA 80144F90   RET 40144E02   EPA 00144F78   R0  FD000008
             R1  00000001   R2  00000001   R3  E8E8E8E8   R4  0002BE58   R5  00000001   R6  E9E9E9E9  Regs at CALL to NEXTPROG
             R7  0002F148   R8  0003E500   R9  0003FFF8   R10 0003E528   R11 00000000   R12 401FE482
NEXTPROG S.A.
SA   144F90  WD1 00000000   HSA 80144AD4   LSA 00000000   RET 00000000   EPA 00000000   R0  00000000
             R1  00000000   R2  00000000   R3  00000000   R4  00000000   R5  00000000   R6  00000000
             R7  00000000   R8  00000000   R9  00000000   R10 00000000   R11 00000000   R12 00000000

INTERRUPT AT 144FE4
```

```
PROCEEDING BACK VIA REG 13

SA    144F90  WD1 00000000   HSA 80144AD4   LSA 00000000   RET 00000000   EPA 00000000   R0  00000000
              R1  00000000   R2  00000000   R3  00000000   R4  00000000   R5  00000000   R6  00000000
              R7  00000000   R8  00000000   R9  00000000   R10 00000000   R11 00000000   R12 00000000

GSFC      WAS ENTERED VIA CALL          AT EP NEXTPROG

SA    144AD4  WD1 00144B3E   HSA 00144F10   LSA 80144F90   RET 40144E02   EPA 00144F78   R0  FD000008
              R1  00000001   R2  00000001   R3  E8E8E8E8   R4  0002BE58   R5  00000001   R6  E9E9E9E9
              R7  0002F148   R8  0003E500   R9  0003FFF8   R10 0003E528   R11 00000000   R12 401FE482

REGS AT ENTRY TO ABEND

     FLTR 0-6       40F1F0F2F9F3F140       F0F0F0F040C9C5C5              F1F0F2C940C9D5C9       0000000000000000

     REGS 0-7       FD000008   00000001   80144F90   E8E8E8E8           0002BE58   00000001   E9E9E9E9   0002F148
     REGS 8-15      0003E500   0003FFF8   0003E528   00000000           401FE482   80144F90   40144E02   00144F78

LOAD MODULE    GSFC

144AC0   47F0F00C 07D6E4D9 D7D9D6C7 90ECD00C   4520F05C 00144B3E 00144F10 80144F90   *.00..OURPROG.....0.............*
144AE0   40144E02 00144F78 FD000008 00000001   00000001 E8E8E8E8 0002BE58 00000001   * ...................YYYY........*
144B00   E9E9E9E9 0002F148 0003E500 0003FFF8   0003E528 00000000 401FE482 50D0F1FC   *ZZZZ..1...V....8..V..... .U...1.*
144B20   D203D008 F4A858D0 F4AC502D 000850D2   000418D2 47F0D16E 5BE3D9C1 C3C5FFFF   *K...4...4.....K...K.OJ..TRACE..*
144B40   FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF   FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF   *................................*
         LINES 144B60-144C00 SAME AS ABOVE                                  Trace vector
144C20   FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF   FFFFFFFF FFFFFFFF FF010204 020403FF   *................................*
144C40   0001D2FF D06AD06B 9201D16A 4820D16C   41220001 4020D16C 41500002 1B114010   *..K.......J...J..... .J....... .*
144C60   E4810 D1AE4111 00014010 D1AE4810   D1B04111 00014010 D1B047F0 D1B2C2D3   *J...J..... .J...J..... .J..0J.BL*
144C80   D2F20002 0002 45E0 D1D245E0 D33C4650   D18E58D0 D1E81BFF 50FD0010 98ECD00C   *K2......JK..L...J...JY.........*
144CA0   92FF[illegible]47F0 D1DAE2E4 C2E790EC   D1F047F0 D22CF1F6 FF020002 00155768   *.......0J.SUBX..J0.0K.16.......*
144CC0   0014[illegible]4C8A 01144AC0 FD000008   00000002 00000001 5C03E528 0002BE58   *.....................*.....V.....*
144CE0   00000001 E9E9E9E9 0002F148 00C3E500   0003FFF8 0003E528 00000000 401FE482   *....ZZZZ..1...V....8..V..... .U.*
144D00   5810D49C D2FF1000 10019202 11C04810   D1E64111 00014010 D1E692FE D1E45830   *..M.K...........JW.... .JW..JU..*
144D20   D48490EC D26892FF D1E49835 D20407FE   FC020000 00144CB8 00144D88 A0144C8A   *M...K...JU..K.................*
144D40   01144AC0 FD000008 00000002 00000001   E7E7E7E7 0002BE58 00000001 E9E9E9E9   *................XXXX........ZZZZ*
144D60   0002F148 0003E500 0003FFF8 0003E528   00000000 401FE4E2 47F0D2AC E2E4C2E8   *...1...V....8..V..... .U..OK.SUBY*
144D80   90ECD2C0 47F0D2FC FE030001 00144D30   00144E20 80144EAE 01144AC0 FD000008   *..K..OK.....................*
144DA0   00000001 00000001 E9E9E9E9 0002BE58   00000001 E9E9E9E9 0002F148 0003E500   *........ZZZZ........ZZZZ..1...V.*
144DC0   0003FFF8 0003E528 00000000 401FE482   5810D49C D2FF1000 10019203 11004810   *...8..V..... .U...M.K..........*
144DE0   D2B64111 00014010 D2B692FE D2B45830   D4880700 47F0D328 00144F78 58F0D324   *K..... .K...K...M....0L......0L.*
144E00   05EF5030 D2D492FF D2B498EC D2C007FE   47F0D344 E2E4C2E9 90ECD358 47F0D394   *....KM..K...K....0L.SUBZ..L..0L.*
144E20   FE040002 00144D88 00144EC8 A0144C8E   01144AC0 FD000008 00000002 00000001   *.....H.....................*
144E40   5C03E528 0002BE58 00000001 E9E9E9E9   0002F148 0003E500 0003FFF8 0003E528   *..V.....ZZZZ..1...V....8..V.*
144E60   00000000 401FE482 5810D49C D2FF1000   10019204 11004810 D34E4111 00014010   *.... .U...M.K.....L..... .*
144E80   D34E92FE D34C5830 D48C1863 5950D4A0   4770D3DA 4810D3D4 41110001 4010D3D4   *L...L...M.....M...L...LM.... .LM*
144EA0   47F0D3D6 C2D3D2F6 000145E0 D2A490EC   D400D23B D448D358 5060D468 92FFD34C   *.OLOBLK6....K...M.K.M.L...M...L.*
144EC0   98ECD448 07FE40F4 FC040000 00144E20   00144F10 80144C8E 01144AC0 FD000008   *..M... 4.....................*
144EE0   00000001 00000001 E9E9E9E9 0002BE58   00000002 E9E9E9E9 0002F148 0003E500   *........ZZZZ........ZZZZ..1...V.*
144F00   0003FFF8 0003E528 00000000 401FE482   FB0400C0 00144EC8 80144AD4 80144C8E   *...8..V..... .U.........H...M....*
144F20   01144AC0 FD000008 00000001 00000001   5C03E528 0002BE58 00000002 E9E9E9E9   *.....................V.........ZZZZ*
```

DO loop counts

OURPROG count

APPENDIX A

MACRO INSTRUCTION DESCRIPTIONS

This appendix contains the formal descriptions of all of the STRCMACS which may be coded by the user. The format is similar to that of "OS Data Management Macro Instructions" IBM Form GC26-3794-1. The reader is referred to this publication for a description of the terms used and the format. The macros are listed in alphabetic order.

Conditional Expressions

A number of macros allow the coding of a group of operands as a conditional expression. This is a group of instructions and test conditions connected by the logical operators AND or OR.

The basis for the conditional expression is the bc-spec which indicates possible values of the 360/370's condition code. The bc-spec may be any one of the following:

- An assembly-language extended branch mnemonic excluding the initial "B" (for example, "Z" from the mnemonic "BZ") or one of the following: "GT", "GE", "EQ", "LT", or "LE". Any of these may be optionally preceded by "REL=".

- "MASK=" followed by an absolute expression (limited to 8 characters) defining the mask of a BC instruction.

The logical value of the bc-spec is <u>true</u> if the corresponding branch instruction would branch. (The branch instruction corresponding to "GT" is "BH"; for "GE", "BNL"; for "EQ", "BE"; for "LT", "BL"; and for "LE", "BNH".)

A simple conditional consists of either a bc-spec alone or a condition code setting instruction and a bc-spec inclosed in parentheses and separated by commas:

(*opcode*, *op1*, . . . , *opn*, *bc-spec*) or (*bc-spec*)

The simple conditional has the logical value <u>true</u> if the *bc-spec* is <u>true</u> after executing the indicated instruction, if any.

A conditional expression consists of one or more simple conditionals separated by the logical connectors AND or OR (and also by the commas required

in macro syntax). In addition, angle brackets "<" and ">"* may be specified as operands for the grouping of subexpressions. For example:

cond1,AND,< , *cond2*,OR, *cond3*, > (*)

The OR is of higher precedence than the AND. That is

cond1 ,AND, *cond2* ,OR, *cond3*

is the same as

<, *cond1* ,AND, *cond2*, >,OR, *cond3*

The logical value of a conditional expression is true if the logical result of the indicated operations on the values of the simple conditionals is true.

Only as many of the simple conditions are evaluated as are required to determine the value of the entire conditional expression. In the example (*) above, if the value of *cond1* is false, the expression must be false so the remaining two simple conditionals are not evaluated.

*The character "+" may be used in place of " < " and "/" in place of " > ".

ATEND–Define Normal Loop Termination Code

The ATEND macro is used to terminate loop definition (if not already terminated by an ONEXIT macro) and to define the start of the code segment which is to be executed when the current DO loop terminates normally (that is, by the condition indicated on the DO macro). The end of the ATEND code segment is defined by the first ONEXIT or OD macro which occurs at the same nest level.

	ATEND	[*block-name*]

block-name sym

Indicates that this ATEND is intended to be a part of the DO block named *block-name*. If coded, checks will be made to assure it is the current block.

ATEXIT—Define Abnormal Loop Termination Code

ATEXIT is provided as an alias for ONEXIT. See description of ONEXIT.

	ATEXIT	[*block-name*]

BLEND—Terminate Current Block

The BLEND (Block End) macro is used to terminate specifically the blocks defined by the BLOCK macro and to act as a generic alias for the FI, OD, ESACOD, ESAC, and CORP macros. The block termination code is generated and the current nest level is decremented by one.

	BLEND	[*block-name*] [*other-ops*]

block-name sym

Indicates that this BLEND is intended to match the BLOCK or other block-defining macro named *block-name.* If coded, checks will be made to assure it is the current block.

other-ops

Any operands which may be specified on the appropriate block-terminating macro may be coded.

BLOCK—Define a Simple Block of Code

The BLOCK macro defines the beginning of a simple block of code. The current nest level is increased by one to cause the BLOCK block to be nested immediately inside any previous current block. The block is terminated by the first BLEND macro that occurs at the same nest level.

[*blname*]	BLOCK	

blname

sym

The name associated with this BLOCK block and to be defined on the first instruction generated.

CASE–Define a DOCASE Alternative

The CASE macro defines the beginning of a block which is to be one of the alternatives for the immediately surrounding DOCASE block. The operands indicate those values which the index must have or a conditional expression which must evaluate to true for the CASE block to be executed. The current nest level is increased by one to cause the CASE block to be nested immediately inside the previous current DOCASE block. The CASE block is terminated by the first ESAC, CASEND, or BLEND macro which occurs at the same nest level.

[*blname*]	CASE	[MISC *index-list* *char-index-list* *conditional-test*]

blname sym

The name associated with this CASE block and to be defined on the first instruction generated.

MISC

Indicates this CASE is to be executed only if no other CASE applies. If this operand is coded, the surrounding DOCASE block cannot have the ONLY operand coded.

index-list

A list of values for which this case will be chosen. Each item in the list must be a self-defining term (*e.g.*, 13 or X'1C'), an absolute expression (*e.g.*, VAL where VAL EQU X'10'), or a pair of such items enclosed in parentheses (*e.g.*, (13,VAL)) indicating that all values in the range (13, 14, 15, and 16=VAL=X'10' in our example) are to select this CASE. *index-list* is invalid with the character-string or conditional-test forms of the DOCASE. If *index-list* is specified for a SIMPLE DOCASE, it must contain a single self-defining term. All values must be in the range 0-4095.

char-index-list

A list of values for which this CASE will be chosen. This form is coded when the immediately surrounding DOCASE is of the character-string format (indicated by the specification (*index*, *length*) on the DOCASE macro).

Each value in the list is interpreted as a character string and may be one of the following:

- A literal (*e.g.*, =C'ABC' or =X'12CF').
- A literal without the leading equal sign (*e.g.*, C'ABC' or X'12CF').
- A string of characters in quotes (*e.g.*, 'ABC' or '12CF'—note that the latter is the same as C'12CF', not X'12CF').
- An address at which there is a character string to be compared (*e.g.*, ABCCODE where ABCCODE DC C'ABC'. Note that an operand such as 15 would be interpreted as this form and would mean absolute address 15—probably not what was intended).
- Any two of the above enclosed in parenthesis indicating a range of values (*e.g.*, ('ABC','ABE')).

conditional-test

Indicates this CASE is to be executed if this conditional expression evaluates to true and no previous CASE of the same DOCASE evaluated as true. A conditional expression is coded when the immediately surrounding DOCASE contained no index specification. See beginning of this appendix for definition of a conditional expression.

If no operands are coded on this CASE macro, then no operands should be coded on any of the CASE macros which are immediately contained within the same DOCASE (excepting, of course, any MISC CASE). The first CASE will then be assumed to be CASE 1, the second to be CASE 2, and so forth.

CASEND–Terminate a DOCASE Alternative

CASEND is provided as an alias for ESAC. See description of ESAC.

	CASEND	[*block-name*]

CORP–Terminate a Procedure

The CORP macro defines the end of a procedure block. Code may be generated to restore appropriate registers to their contents at the evocation of the proc, to pass back a return code, and to transfer into the evoking routine immediately following the point of evocation. The static block nest level is decremented by one.

[*label*]	CORP	[*proc-name*] [RESTORE=(*first* [,*last*])] [RETURN=*reg-list*] [RC= {NONE / *value* / (*reg*)}] [LINK= {NONE / *linkreg*}]

label sym

If present, *label* will appear on the first instruction generated.

proc-name sym

Indicates this CORP is intended to match the outstanding PROC block named *proc-name*. If coded, checks will be made to assure it is the current block.

RESTORE=*(first,last)* dec dig, sym

Indicates the first and last registers to be restored. These must be a subsequence of those saved. If *last* is not specified, only register *first* will be restored. If the entire operand is omitted, all registers saved will be restored.

RETURN=*reg-list* dec dig, sym

One or more registers which would otherwise be restored but which are to be exceptions. The registers in the RETURN= list may be thought of as output values being returned to the caller. Used mainly for non-OS procs.

RC=NONE

Indicates no return code processing is to be performed. Register 15 will be handled as indicated by the RESTORE= and RETURN= operands.

RC=*value* abs ex

Indicates the number *value* is to be returned in register 15.

RC=*(reg)* dec dig, sym

Indicates the value in register *reg* is to be returned in register 15.

If RC= is not coded the defaults are:

For OS procs:	RC=0
For non-OS procs:	RC=NONE

LINK=NONE

Indicates the returning branch is to be omitted and control be allowed to fall out the bottom of the CORP.

LINK=*linkreg* dec dig, sym

Indicates a final "BR *linkreg*" instruction is to be used to return to the proc's caller.

If LINK= is omitted, LINK=14 is assumed.

DO–Define Iterative Block

The DO macro defines the beginning of a segment of code to be executed repetitively until some condition occurs. The current static nest level is increased by one to cause the DO block to be nested immediately inside any previous current block. The DO block is terminated by the first OD, DOEND, or BLEND that occurs at the same nest level. The looping segment itself is terminated by the first OD, DOEND, BLEND, ATEND, ONEND, ONEXIT, or ATEXIT that occurs at the same nest level.

[*blname*]	DO	[FOREVER] [WHILE,*looping-group* [,{AND / OR},UNTIL,*looping-group*]] [UNTIL,*looping-group* [,{AND / OR},WHILE,*looping-group*]]

blname sym

Name associated with this DO block and to be defined on the first instruction generated.

FOREVER

Indicates the main looping control of the block is to contain no test for loop termination.

WHILE,*looping-group*

Indicates that the tests indicated by the looping group are to be performed logically before the execution of the loop and the loop is to be executed as long as the looping group evaluates true.

UNTIL,*looping-group*

Indicates that the tests indicated by the looping group are to occur logically after loop execution—*i.e.*, the first execution of the loop is not dependent on the UNTIL looping group. The looping will continue as long as the looping group evaluates false.

The order of the WHILE and UNTIL is not significant.

AND

Indicates that the WHILE group must be true *and* the UNTIL group must be false for loop execution to continue.

OR

Indicates that either the WHILE group must be true *or* the UNTIL group must be false for the loop execution to continue.

looping-group

Specifies the test to be made. The looping group is:

$$\left\{ \begin{array}{l} \textit{looping-branch}\left[,\left\{\begin{array}{c}\text{AND}\\ \text{OR}\end{array}\right\}\ ,\textit{cond-test}\right] \\ \textit{cond-test} \end{array} \right\}$$

looping-branch

One of the special looping instructions specified as:

(BCT,*reg1*)
(BXH,*reg1* ,*reg2*)
(BXLE,*reg1* , *reg2*)

In an UNTIL looping group, the looping branches are considered to be <u>true</u> when they fall through. In a WHILE looping group, the looping branches are considered to be <u>true</u> when they branch. Note: DO WHILE,(BCT,*reg1*) will loop one time less than the initial value in *reg1*.

cond-test

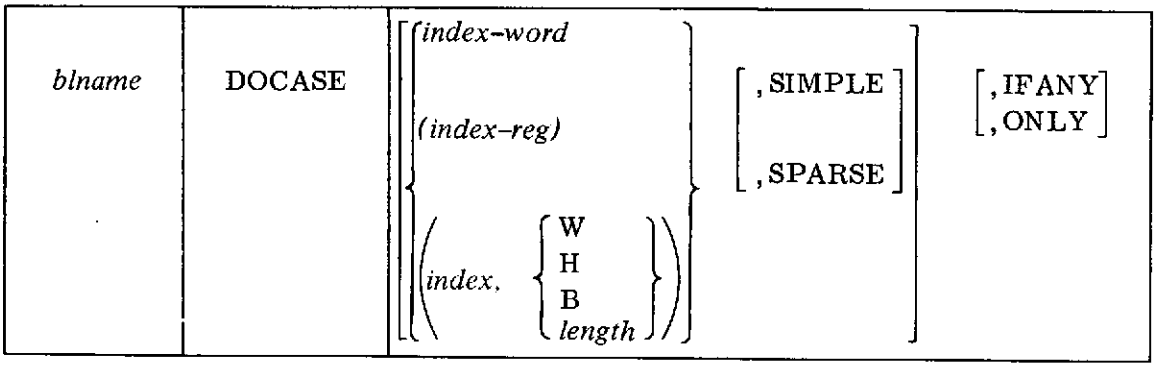
Is a conditional expression. See beginning of this appendix for the definition of a conditional expression.

The DO may contain at most one looping branch—that is, the WHILE and UNTIL may not both contain the operations BCT, BXH, or BXLE.

A DO macro with no operands defaults to a "DO FOREVER".

DOCASE–Define a Selection Among Alternatives

The DOCASE macro defines the beginning of a block in which it is immediately nested a number of CASE blocks. An appropriate one (or possibly none) of these CASE blocks will be selected for execution as directed by the operands of the DOCASE and CASEs. The current static nest level is increased by one to cause the DOCASE to be nested immediately inside any previous current block. The block is terminated by the first ESACOD, DOCASEND, or BLEND that occurs at the same nest level. Nothing should be immediately contained within the DOCASE block except CASE blocks. (That is, the DOCASE macro should be immediately followed by the first CASE macro.)



blname sym

Name associated with this DOCASE block and to be defined on the first instruction generated.

index-word RX-type

Indicates the DOCASE index is located in the word at address *index-word*.

(*index-reg*) dec dig, sym

Indicates the DOCASE index is located in the register *index-reg*.

(*index*,W) RX-type

Indicates the DOCASE index is located in the word at address *index*. Same as first alternative.

(*index*,H) RX-type

Indicates the DOCASE index is located in the half-word at address *index*.

(*index*,B) A-type

Indicates the DOCASE index is located in the byte at address *index*.

(*index*,*length*) *index*-A-type
length-abs exp

Indicates the DOCASE is to select a CASE on the basis of character strings; the "index" string is at address *index* and of length *length*. SIMPLE and SPARSE are invalid with this option.

If none of the indexing operands are coded, the DOCASE is implied to be of conditional test type—each of the CASE macros, which are nested immediately within the DOCASE, must have a conditional test as its operand.

SIMPLE

Indicates the DOCASE will contain immediately nested within it a small number of CASE blocks. If there are n such blocks (ignoring any MISC CASE which may be present), they are to be associated with index values 1, 2, 3, . . . , n. Better code is produced for such situations when SIMPLE is coded and $n \leq 6$ (if no MISC CASE is present) or $n \leq 12$ (if a MISC CASE is present).

SPARSE

Indicates the number of CASE blocks which follow is small compared with the range of values (between zero and the maximum index specified on any CASE block). Better code is produced for such situations when SPARSE is coded.

IFANY

Indicates that if none of the immediately nested CASE blocks apply on any given index value, then either the MISC CASE is to be executed (if one is present) or no block is to be executed and control is to continue following the ESACOD.

ONLY

Indicates that the only values of the index which can occur are provided for by the immediately nested CASE blocks and no test need be made for other values. If ONLY is coded, no MISC CASE may be present. If neither IFANY nor ONLY is coded, IFANY is assumed.

***DOCASEND**–Terminate Alternative Selection*

The DOCASEND macro is provided as an alias for the ESACOD macro. See ESACOD for description.

	DOCASEND	[*block-name*]

DOEND–Terminate Iteration Block

The DOEND macro is provided as an alias for the OD macro. See OD for description.

	DOEND	[*block-name*]

ELSE–Define IF Alternative and Terminate True Condition

The ELSE macro terminates the definition of the true block of the IF (which is the current block) and initiates a block which is to be executed if and only if the IF block is bypassed. The ELSE block is terminated by the first FI, IFEND, or BLEND macro which occurs at the same nest level.

[*else-name*]	ELSE	[BLEND=*if-name*]

else-name sym

Name associated with this ELSE block and to be defined on the first instruction generated. If the ELSE= operand was coded on the corresponding IF, a check will be made to assure that the else-names match.

BLEND=*if-name* sym

Indicates that this ELSE is intended to match the IF block named *if-name*. If coded, checks will be made to assure that it is the current block.

ESAC–Terminate a DOCASE Alternative

The ESAC macro is used to terminate the current CASE block. The block termination code is generated and the current nest level is decremented by one. The ESAC should be immediately followed by either another CASE macro or the ESACOD.

	ESAC	[*block-name*]

block-name sym

Indicates that the ESAC is intended to match the outstanding CASE block named *block-name*. If coded, checks will be made to assure that it is the current block.

ESACOD–Terminate a Selection Among Alternatives

The ESACOD macro is used to terminate the current DOCASE block. The block termination code is generated and the current nest level is decremented by one.

	ESACOD	[*block-name*]

block-name sym

Indicates the ESACOD is intended to match the outstanding DOCASE block named *block-name* . If coded, checks will be made to assure that it is the current block.

EXIT—Abnormally Exit to the End of a Containing Block

The EXIT macro causes control to immediately transfer to the end of some containing block. Since control cannot pass out the bottom of an EXIT macro, it is usually immediately followed by the block terminating macro of its containing block (often a FI). If the EXIT is nested at some depth within a proc, the EXIT may be made to the end of the proc, but not to the end of any block which may surround the proc. The EXIT does not affect the current nest level.

[*label*]	EXIT	[*block-name*]

label sym

If a label is coded, it will be generated for cross-reference purposes.

block-name sym

The name of the block from which control will exit. Neither the block immediately surrounding the EXIT nor any block surrounding the proc surrounding the EXIT may be specified. If no operand is specified, the second containing block (the block containing the block containing the EXIT macro) is assumed.

FI–Terminate a Conditional Block

The FI block is used to terminate the current IF or ELSE block. The block termination code is generated and the current nest level is decremented by one.

	FI	[*block-name*]

block-name sym

Indicates the FI is intended to match the outstanding IF or ELSE named *block-name*. If an ELSE has been coded, the IF block name cannot be specified. If *block-name* is specified, checks will be made to assure that it is the current block.

FINAL–Insure Structures are Terminated

The FINAL macro checks to be sure that all blocks have been terminated (that the current nest level is zero). If SAVETRACE debugging is being performed, the final static save area links are defined. The FINAL macro should not be coded more than once in an assembly and should follow the last block defined. It is optional unless SAVETRACE debugging has been requested.

	FINAL	

IF–Define Conditional Block

The IF macro defines the beginning of a block of code to be executed only under certain conditions. The static nest level is increased by one to cause the IF block to be nested immediately inside any previous current block. The construct is terminated by the first FI, IFEND, or BLEND that occurs at the same nest level. The IF block itself is terminated by the first FI, IFEND, BLEND, or ELSE that occurs at the same nest level.

[*blname*]	IF	{ASYNCH / *cond-test*} [EXIT= {*exit-block* / *}] [ELSE=*else-block*]

blname sym

Name associated with this IF block and to be defined on first instruction generated.

ASYNCH

Indicates control is to never fall through into the block; an unconditional branch around the block will be generated. EXIT= must not be coded.

cond-test

The conditional expression which, if it evaluates to true, will cause the block to be executed. If the EXIT= operand is specified, the exit will occur if the conditional expression is true. See the beginning of this appendix for the definition of conditional expressions.

EXIT=*exit-block* sym

If *cond-test* is true, control will pass to the end of the block named *exit-block*. No block surrounding the proc surrounding the IF may be specified as *exit-block*.

EXIT=*

If *cond-test* is true, control will pass to the end of the block immediately containing the IF macro.

If the EXIT= operand is coded, ASYNCH and ELSE= may not be coded. In addition, no FI is required (and must not be coded) to terminate the IF, since the block is defined only long enough to take the exit.

ELSE=*else-block* sym

Indicates an ELSE macro will follow at the same nest level with the name *else-block*. If the ELSE= operand is specified, a check will be made to assure the ELSE block is coded and properly named. The ELSE= operand need not be coded even if an ELSE macro follows—it is provided only as a check.

IFEND—Terminate a Conditional Block

The IFEND macro is provided as an alias for the FI macro. See FI for description.

	IFEND	[*block-name*]

OD–Terminate Iterative Block

The OD block is used to terminate the current DO block. The end of the loop segment is defined if it did not previously occur by the coding of an ATEND or ONEXIT macro. If either an ATEND or ONEXIT segment is outstanding, it is terminated. The current nest level is decremented by one.

	OD	[*block-name*]

block-name sym

Indicates the OD is intended to match the outstanding DO block named *block-name*. If coded, checks will be made to assure that it is the current block.

ONEND–Define Normal Loop Termination Code

The ONEND macro is provided as an alias for the ATEND macro. See ATEND for description.

	ONEND	[*block-name*]

ONEXIT—Define Abnormal Loop Termination Code

The ONEXIT macro is used to terminate loop definition (if not already terminated by an ATEND macro) and to define the start of the code segment which is to be executed when the loop defined by the DO macro at the current nest level terminates abnormally (that is, by the execution of an exit specifying the DO as its target). The end of the code segment is indicated by the first ATEND or OD macro which occurs at the same nest level.

	ONEXIT	[*block-name*]

block-name sym

Indicates that this ONEXIT is intended to be a part of the DO block named *block-name*. If coded, checks will be made to assure it is the current block.

PROC–Define a Proc

The PROC macro defines the beginning of a proc block. The proc may follow OS linkage conventions or be of a simpler non-OS type. The current nest level is increased by one to cause the PROC to be nested immediately inside any previous current block, although procs are normally outermost blocks. The proc is terminated by the first CORP, PROCEND, or BLEND macro that occurs at the same nest level.

[*proc-name*]	PROC	[LINKAGE=([OS],[CSECT])] [ID={NONE / *id-string*}] [SAVE={NONE / (*first* ,[*last*] ,[DYNAM / NONE / *savearea*] ,[*length*])}] [BASE={NONE / *basereg* / (*baselist*)}] [WORK={NONE / *workreg*}] [EXIT=*severity*] [DEBUG=*options-list*]

proc-name sym

Name associated with this PROC block and to be defined on first instruction generated.

LINKAGE=OS

Indicates this PROC will be invoked following standard OS conventions—entry point in register 15, return point in register 14, save area address in register 13. If coded, any save area linkage will follow OS standards. If omitted, a simpler non-OS proc is generated.

LINKAGE=(,CSECT)

Indicates a CSECT pseudo-operation is to be generated using *proc-name* in the name field.

ID=NONE

No in-line identifier is to be generated.

ID=*id-string*

The character string *id-string* is generated in-line similar to that generated by the OS SAVE macro. (The length field is omitted if the PROC is not OS LINKAGE.) The character string may optionally be surrounded by apostrophes.

ID=*

The proc name is generated as an in-line character constant. (If *proc-name* is not specified, the internal block name is used for non-OS procs, "$PRIVATE" for OS procs.)

If the ID= operand is not coded, the defaults are:

For OS procs, ID=*

For non-OS procs, ID=NONE

SAVE=NONE

No registers are to be saved and no new save area is to be provided.

SAVE=(*first,last*) dec dig, sym

All of the registers in the range *first* through *last* are saved in the appropriate save area (the previous standard save area pointed to by register 13 for OS procs, or an in-line save area for non-OS procs). The sequence of registers must be a sub-sequence of the standard 14 through 12 (*i.e.*, something like "(10,15)" is invalid). If *last* is omitted, only register *first* is saved. If omitted, (14,12) is assumed.

SAVE=(,,DYNAM)

Specifies the new save area is to be obtained via GETMAIN and freed by the corresponding CORP. Valid for OS procs only.

SAVE=(,,NONE)

Specifies that no new OS save area is to be provided, but the registers indicated by the first two suboperands are to be saved in the old save area. Valid for OS procs only.

SAVE=(,,*savearea*) sym

Specifies the address of a user-provided new save area. Valid for OS procs only.

If the third suboperand of the SAVE= keyword is omitted (and SAVE=NONE is not coded) on OS procs, an in-line save area will be generated within the PROC macro as the new save area.

SAVE=*(,,,length)* dec dig, sym

Gives the length, in words, of the dynamic or in-line save area. If specified symbolically for an in-line save area, the symbol must be previously defined. If omitted, default is 18. Valid for OS procs only.

BASE=NONE

Indicates that no base register loading is to be performed.

BASE=*basereg* dec dig, sym

Code to load register *basereg* will be generated and a USING will be issued against it. The operand must be one of the registers 2 through 12.

BASE=(*baselist*) dec dig, sym

A list of base registers may be supplied. Each register in the list will be loaded 4096 bytes beyond the previous and USINGs will be issued for all registers in the list. If the first suboperand of the list is omitted (by coding "BASE=(,*reg*2,*reg* 3, . . . *regn*)"), the default base register will be assumed. (See below.) Only registers 2 through 12 may be specified.

If the BASE= operand is omitted, the defaults are:

For OS procs with an in-line save area— BASE=13
(May not be explicitly coded.)

For OS procs without an in-line save area— BASE=12

For non-OS procs— BASE=NONE
(If the first suboperand of baselist is omitted for non-OS procs, it defaults to 12.)

WORK=NONE

Indicates that any register (other than register 0) destroyed in the code generated is to be restored.

WORK=*workreg* dec dig, sym

Indicates that register *workreg* may be destroyed by the code generated and need not be restored. The work register may not be specified as a base register.

If the WORK-operand is omitted, defaults are:

For OS procs: WORK=2

For Non-OS procs: WORK=1

EXIT=*severity* dec dig or "*"

Specifies that the error message which is generated at the target of an EXIT is to have the indicated severity code. The value of severity must be between 0 and 4095 or be a "*". Once specified, it will remain in effect until specified on some other proc. Until first specified, the severity is 0.

DEBUG=*options-list*

Indicates those debugging options to be turned on or off during the duration of this proc.

The individual options may be turned on by specifying either the option or its abbreviation from the following list.

LISTBLOCKS[LB]— List block name, number, and nest level in comment at beginning and end of each block.

PROCNAMES[PN]— Each proc's name is to be generated as an in-line character constant.

BLOCKNAMES[BN]— Each block's name is to be generated as an in-line character constant.

PROCCOUNTS[PC]— Code is to be generated to count proc executions.

BLOCKCOUNTS[BC]—Code is to be generated to count all block executions.

PROCTRACE[PT]— Code is to be generated to keep track of the last 257 procs envoked.

CORPVALUES[CV]— Maintain save areas to hold values of registers at non-OS CORPs.

SAVETRACE[ST]— Statically link together all save areas in non-OS procs and dynamically insert entire chain in save area list on entry to first proc. For this option, first proc must be LINKAGE=OS and must enable the SAVETRACE option. The FINAL macro must also be coded following the last proc.

To turn off any of the options, prefix the name by NO- or the abbreviation by N- (*e.g.*, "NOPROCTRACE" or "NPT"). When the CORP is generated, options will revert to their status before the PROC macro. To avoid the restoring of the options' status at CORP time, include "GLOBAL" (or "GBL") in the list. "ALL" or "NONE" may be specified to turn on or off all options; either may be followed by exceptions. (*e.g.* , "DEBUG=(ALL,NST)" turns on all options except the save-trace.)

PROCEND--Terminate a Proc

The PROCEND macro is provided as an alias for CORP. See CORP for description.

[*label*]	PROCEND	[*proc-name*] [(RC={NONE / *value* / (*reg*)})] [RESTORE=(*first* [,*last*])] [RETURN=*reglist*] [LINK={NONE / *linkreg*}]

APPENDIX B

INTRODUCTION TO ABSTRACT SOURCE LISTING OF STRCMACS

OS macro assembly language is an insufficiently powerful language for doing structured programming. As a result, the programming of the STRCMACS was performed in an abstract programming language called SIMPL-M. This is an imaginary language which is a hybrid of SIMPL-X (a high-level structured programming language developed at the University of Maryland),[1] OS macro assembly language, and the STRCMACS themselves. After the code was written in SIMPL-M, it was translated by hand to OS macro assembly language. The SIMPL-M program is considered the "source" code and all updates are performed in it. It is much easier to read than the macro assembly language source. The SIMPL-M source for the macros is listed in Appendix C. In this appendix, we will give a brief description of the SIMPL-M language. In addition, a decision table for the DO macro formats is included in this appendix to complete the source documentation.

Introduction to SIMPL-M

SIMPL-M is a high-level language for the specifying of assembly language macros. In some ways it resembles ALGOL or PL/I; it provides for aribtrary nesting of control structures such as if, while...do, and docase. Two types of modules are allowed: macros and procs. The macros are not macros in the sense that they are expanded when the SIMPL-M source is "translated"; they are macros in the sense that the translated version defines and may be evoked as OS assembly language macros. The operands which are specified for macros closely parallel the allowable operands of OS macro prototype statements (that is, a name field operand and a list of positional and/or keyword operands). The procs are parameterless modules constructed during the stepwise refinement of each of the macros of the STRCMACS. They are expanded in-line in the translation to the assembly language macro definition. Both macros and procs are shown as being evoked by call instructions. The distinction is obvious since the macro calls always have argument lists (possibly empty as "call BLEND (;)"), and the proc calls never have argument lists. In addition, procs always have multi-word names whose first word indicates the macro of which the proc is a part. (For example, the proc "DOCASE_GENERAL_SETUP" is a part of the DOCASE macro.)

[1] Basili, Victor R., "SIMPL-X: A Language for Writing Structured Programs," University of Maryland Technical Report TR-233, January 1973.

The correspondence between the SIMPL-M macro statement and an OS assembly language macro prototype is illustrated by the following example:

SIMPL-M:

```
macro  CORP    (USER_NAME; PROC_NAME, RETURN=, LINK=14,
                RESTORE=, RC=)
```

OS MACRO:

```
               MACRO
&USRNAME   CORP    &PROCNAM,&RETURN=,&LINK=14,&RESTORE=,&RC=
```

Statements in SIMPL-M require neither terminators nor continuation indicators. Statement boundaries are unambiguously defined by the use of reserved keywords (which are shown in the listing as lower case underlined terms such as while and generate) and by a carefully chosen syntax.

The data types in SIMPL-M are taken directly from OS macro assembly language. They are:

int - Intergers

bit - Logical variables

char - Character strings

Such variables may be global to all macros and procs (defined before the first macro), local to a macro but global to its procs (defined at the beginning of a macro), or local to a proc and unknown to any macro (defined at the beginning of a proc). Int, bit, and char variables are initialized to 0, false, and '' (the null string) respectively. The globals are initialized at the beginning of the assembly program's execution; the macro locals, at the beginning of each macro expansion; the proc locals are not considered to be initialized. Automatic type conversion occurs as follows:

int to bit: 0 → false; all else to true

int to char: the absolute value of the integer is expressed as characters without leading sign or zeros

bit to int: false → 0; true → 1

bit to char: false → '0'; true → '1'

char to int: Value if numeric character string (with possibly leading "+" or "-"); else undefined

char to bit: '0' → false; '1' → true; else undefined

Character constants may be surrounded by either single or double quotes, but may not contain the delimiter character. One dimensional arrays are allowed. They are demensioned in their declarations as

```
int X(20)
```

and are referred to as

```
X(3) := Y
```

The first element of the array has the index 1.

Macro operands are either positional (determined by order) or keyword (determined by the fixed term preceeding the "="). The variables representing such operands are implicitly defined as char variables. If a list argument corresponds to the parameter X, the whole list may be referred to as "X"; the first item in the list may be referred to as "X(1)"; the second as "X(2)"; etc. If the argument is not a list, it may be referred to as either "X" or "X(1)"; "X(2)" will then have the null string as a value.

The assignment statement is indicated by the symbol ":=". For example:

```
I := 1
```

stores the value 1 into I. Multiple assignments may be made by specifying:

```
I, J := 1
```

Relations may include implied operands. For example:

```
if I = 1 or = 19          is the same as          if I = 1 or I = 19
```

Only as much of the conditional expression is evaluated as is necessary to establish the overall value. This allows such expressions as:

```
if J ≠ 0 and I / J = 4
```

to be evaluated without an underflow occurring.

The body of a macro is terminated by a mend instruction. The mexit instruction causes immediate exit from the macro definition. Character strings are concatenated by using the " || " operator.

```
X := 'ABC'

Y := X || 'DEF'
```

assigns 'ABCDEF' to Y. Brackets are used to select substrings.

```
X := 'ABC'

Y := X[2,1]
```

assigns 'B' to Y. The two expressions in brackets are the starting character position and the length.

The instruction "generate (string)" causes the operand string to be generated as an assembly language instruction at OS macro expansion time.

Three intrinsic functions are provided for testing macro operands. Their values are given below when applied to the macro operand ARG.

T'ARG — Has the char value 'O' (oh, not zero) if ARG was omitted by the user; has the value 'N' if ARG is a decimal self-defining term; has some other value if neither of these is true.

K'ARG — Has an int value equal to the number of characters in ARG considered as a character string.

N'ARG — Has an int value equal to the number of suboperands in ARG. (If ARG is "(A,,B)", N'ARG is 3.)

The special variable SYSLIST takes on the value at macro call of all the positional operands, considered as a list. N'SYSLIST is the number of positional operands to the macro. For example, in the prototype "macro (LAB; X, Y, Z)" SYSLIST(2) and Y may be used to refer to the same operand; SYSLIST(4) is the only way to reference a fourth operand; LAB is the only way to reference the label-field operand.

Comments are surrounded by "/*" and "*/" and may flow over any number of lines. By convention, comments which are inserted as part of a program proof are further nested in braces:

```
/*{...}*/
```

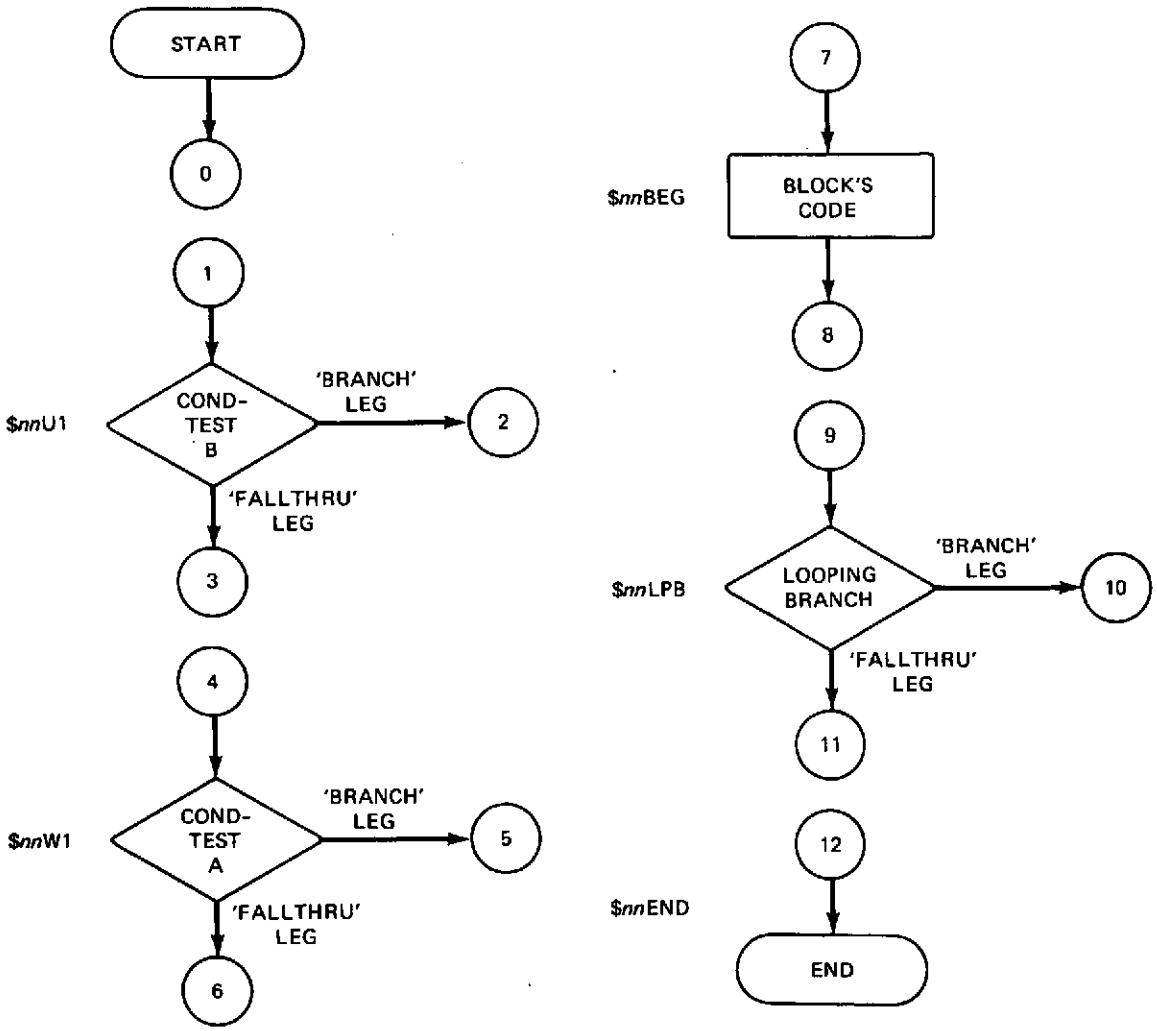
DO Macro Decision Table

A decision table was used to simplify the coding of DO operand processing. This decision table is included here for documentation.

The complete form of a DO macro is

DO WHILE,⟨looping-branch⟩,⟨and/or⟩,⟨cond-test-A⟩,⟨and/or⟩,
UNTIL,⟨looping-branch⟩,⟨and/or⟩,⟨cond-test-B⟩

The complete form of the code generated is given by the partial flow chart:



The following decision table shows the connections which must be made for the various formats. Those shown lightly shaded occur without branching (control falls through to the indicated node). Boxes shown cross-hatched do not occur for that operand combination. An example follows the table.

FOREVER			✓																						
WHILE				✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓					
	looping-branch											✓	✓	✓	✓	✓	✓	✓	✓	✓					
	AND/OR														A	O	A	A	O	O					
	cond-test A			✓	✓	✓	✓	✓	✓	✓	✓				✓	✓	✓	✓	✓	✓					
AND/OR				A	O	A	O	A	A	O	O		A	O			A	O	A	O					
UNTIL				✓	✓	✓	✓	✓	✓	✓	✓		✓	✓			✓	✓	✓	✓	✓	✓	✓	✓	
	looping-branch					✓	✓	✓	✓	✓	✓										✓		✓	✓	
	AND/OR							O	A	O	A												O	A	
	cond-test B			✓	✓			✓	✓	✓	✓		✓	✓			✓	✓	✓	✓		✓	✓	✓	
Entry point - 0			7	4	4	4	4	4	4	4	4	4	9	9	9	9	9	9	9	9	9	7	7	7	7
B	'Branch' leg 2	Condition			T	F			T	T	F	F		T	T			T	T	T	T		T	T	T
		To node			12	7			12	12	7	7		12	9			12	9	12	9		12	12	12
	'Fall thru' leg 3	Condition			F	T			F	F	T	T		F	F			F	F	F	F		F	F	F
		To node			4	4			4	4	4	4		9	7			9	7	9	7		7	7	7
A	'Branch' leg 5	Condition		F	F	F	F	F	F	F	F	F				F	F	F	F	F	F				
		To node		12	12	12	12	12	12	12	12	12				12	12	12	12	12	12				
	'Fall thru' leg 6	Condition		T	T	T	T	T	T	T	T	T				T	T	T	T	T	T				
		To node		7	7	7	7	7	7	7	7	7				7	7	7	7	7	7				
After Code - 8			7	4	1	1	9	9	9	9	9	9	9	1	1	9	9	1	1	1	1	9	1	9	9
Looping Branch	'Branch' leg 10	To node					4	7	1	4	1	7	7	7	7	4	7	4	4	7	7	7		1	7
	'Fall thru' leg 11	To node					12	4	12	1	4	1	12	12	12	12	4	12	12	4	4	12		12	1
OPERAND FORMAT			0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22

Example of the code generated for the macro:

```
DO WHILE,(LTR,3,3,P),AND,
     UNTIL,(BCT,5),OR,
          (CLI,SPOT,C'X',E)
```

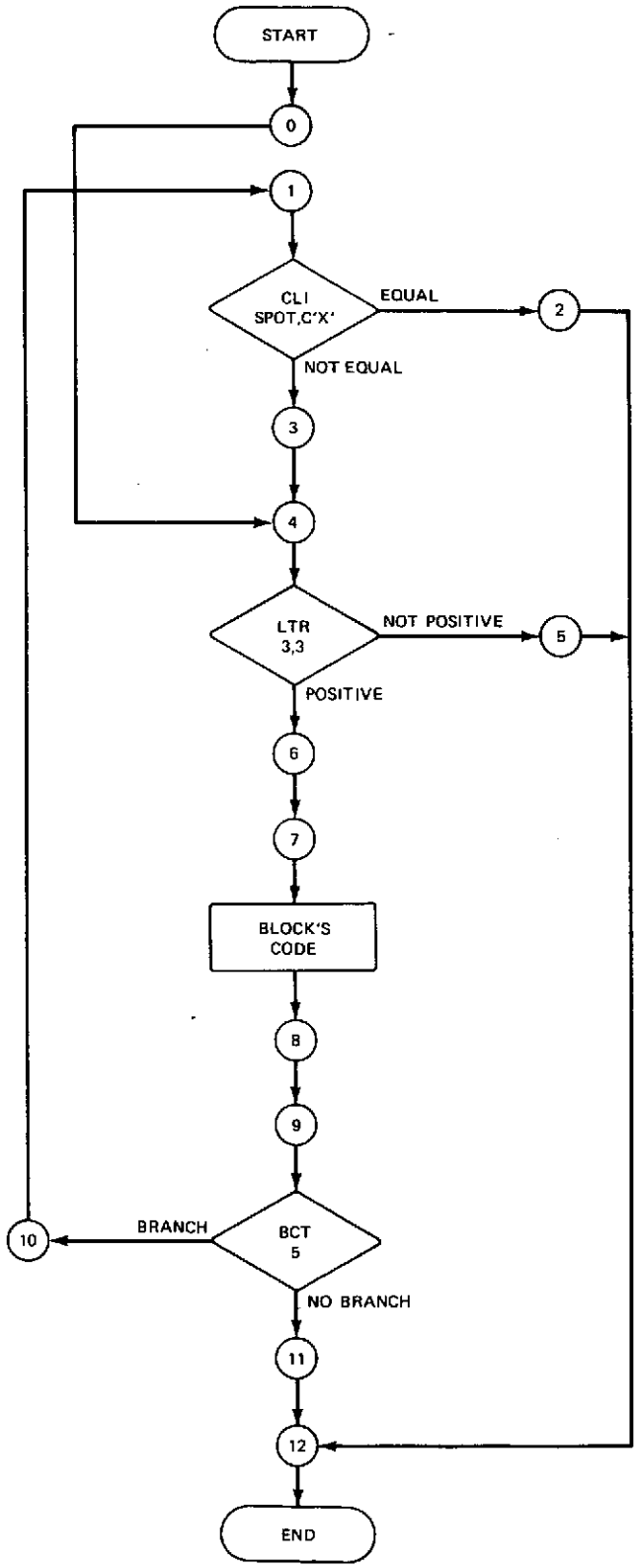
Operand format number 6.

Code generated:

```
         B       $18W1
$18U1    CLI     SPOT,C'X'
         BE      $18END
$18W1    LTR     3,3
         BNP     $18END

        <Block's code>

         BCT     5,$18U1
$18END   DS      OH
```



APPENDIX C

ABSTRACT SOURCE LISTING OF STRCMACS

```
135.       /* GENERAL PURPOSE GLOBALS. */
136.   int
137.       PROC_COUNTER, /* Number for last special proc label "$Ppp".  The
138.             pp is PROC_COUNTER in hex. */
139.       LAST_BLOCK_NUMBER,  /* Number used in labels of most recently generated
140.             block. */
141.       HEX_IN  /* Input value to XHEX macro. */

143.   bit
144.       ERROR_OCCURRED,        /* General purpose error flag used by various
145.             inner macros to report failure. */
146.       TDI_FALLTHRU_OCCURS,  /* Set true by TERMINATE_DO_LOOP if DO may
147.             fall through bottom in terminating (when a looping branch is
148.             present);  else set false. */
149.       NOT_FIRST_PROC,  /* false until processing begins on first PROC
150.             macro. */
151.       SAVETRACE_ON_FIRST_PROC, /* false unless first PROC macro included
152.             DEBUG=(***,SAVETRACE,***) operand. */
153.       TRACE_VECTOR_GENNED  /* false until the trace vector ($TRACE)
154.             is generated on the first PROC which includes the
155.             DEBUG=(***,PROCTRACE,***) operand. */

157.   char
158.       BLOCK_LABEL_PREFIX,        /* Unique character string for each block
159.             for use in generating labels. */
160.       EXIT_SEVERITY, /* Mnote severity for EXIT target message.  Can be
161.             set by PROC macros. */
162.       HEX(16),  /* Constants used in converting decimal to hex by XHEX. */
163.       HEX_OUT, /* Output value from XHEX macro. */
164.       PREV_SAVETRACE_AREA, /* Holds label generated on last local PROC
165.             save area to be used in producing the static chain for SAVETRACE. */
166.       PREV_SAVETRACE_PTR   /* Holds label generated as forward pointer
167.             within last local PROC save area for static chain for SAVETRACE. */


170.       /* DEBUG FLAGS. */

172.   bit
173.       DEBUG_BLOCKCOUNTS_REQD, /* Causes code and counters to keep execution
174.             counts on all blocks. */
175.       DEBUG_BLOCKNAMES_REQD,  /* Causes block names to be generated as
176.             inline character constants to aid in locating within dumps. */
177.       DEBUG_CORPVALUES_REQD, /*  Causes register values at end of procs (at
178.             CORP macro start) to be saved in inline save areas for reference. */
179.       DEBUG_DEBUGMACROS_REQD, /* Causes various intermediate values to be
180.             printed during macro processing for debugging the macros. */
181.       DEBUG_LISTBLOCKS_REQD, /* Causes mnotes to be generated at the start
182.             and end of all blocks listing their name, number, and static
183.             nesting depth. */
184.       DEBUG_MACRONAMES_REQD, /* Causes mnotes to be generated whenever any
185.             macros are entered (including inner macros) which list the macro's
186.             name;  for debugging the macros. */
187.       DEBUG_PROCCOUNTS_REQD, /*  Causes code and counters to keep execution
188.             counts on PROC blocks only. */
189.       DEBUG_PROCNAMES_REQD,   /* Causes PROC names to be generated as
190.             inline character constants to aid in locating within dumps. */
191.       DEBUG_PROCTRACE_REQD,   /* Causes a trace vector to be generated and
192.             code to be generated to keep track of the last 257 PROCs entered. */
193.       DEBUG_SAVETRACE_REQD   /* Causes all local save areas to be statically
194.             chained together and code to be generated to link the chain to the
195.             OS save area to provide OS formatting within ABEND dumps. */
```

```
197.        /* MAIN STACK. Dimensioned to 100. */

199.    int
200.        CURRENT_NEST_LEVEL,       /* Current depth of static nesting of
201.                blocks;  stack pointer. */
202.        NESTING_LIMIT, /* Holds dimension of main stack. */
203.        BLOCK_NUMBER(100)  /* Block number of the Ith block. */

205.    bit
206.        END_LABEL_REQD(100), /* Indicates whether Ith block needs an END label
207.                generated during POP_OLD_BLOCK. */
208.        EXIT_LABEL_REQD(100)  /* Indicates whether Ith block needs an XIT label
209.                generated during POP_OLD_BLOCK. */

211.    char
212.        BLOCK_NAME(100), /* Block name of Ith block, either USER_NAME specified
213.                in macro label field or generated name "BLKnnn" where nnn is the
214.                sequential block number. */
215.        BLOCK_TYPE(100),  /* Macro name which generated the Ith block
216.                (IF, DO, DOCASE, CASE, BLOCK, or PROC). */
217.        OPERAND1(100), OPERAND2(100), OPERAND3(100), OPERAND4(100),
218.           /* These hold various data which are needed to close the blocks
219.                generated.  Specific contents vary according to the type of
220.                block generated.  See individual macros. */
221.        INFORMATION(100) /* Similar to the OPERANDn stacks above, the
222.                INFORMATION stack holds information for the closing of the block.
223.                Often the individual characters within the variables are used for
224.                different values, packed together into INFORMATION. */


227.        /* GCASE STACK.  Holds data for general DOCASEs.  Dimensioned to 9. */

229.    int
230.        MAX_CASE_VALUE(9), /* Maximum branch vector value found. */
231.        NEXT_COMP_LABEL_NO(9), /* Case number for next comparison case label
232.                to be generated. */
233.        GCASE_NEST_LEVEL,   /* Current depth of stacking in the GCASE stack;
234.                number of nested DOCASEs with either GENERAL, SPARSE, or CHARCOMP
235.                operand formats. */
236.        GCASE_NEST_LIMIT   /* Maximum depth of nesting of GCASE stack;  must
237.                be equal to stack dimension. */

239.    bit
240.        CASE_OCCURS(2304)       /* Each group of 256 bits are used to note
241.                which branch vector cases occur. */


244.        /* CONDITIONAL_EXPRESSION_PROCESSOR PSEUDO-PARAMETERS. */

246.    int
247.        FIRST_INDEX,
248.        LAST_INDEX   /* Pseudo-parameters to CONDITIONAL_EXPRESSION_PROCESSOR.
249.                Indicates indexes within SYSLIST of first and last parameter to be
250.                processed. */

252.    bit
253.        ULTIMATE_FALLTHRU_CONDITION, /* Logical value upon which conditional
254.                expression is to pass control (or fall through) to the ULTIMATE_
255.                FALLTHRU_LABEL. */
256.        FALLTHRU_LABEL_USED  /* CEP sets this true if a branch is generated
257.                to the ULTIMATE_FALLTHRU_LABEL (else no change occurs). */

259.    char
260.        ULTIMATE_BRANCH_LABEL, /* Indicates label to be used as branch target
261.                when conditional test does not have logical value stored in
262.                ULTIMATE_FALLTHRU_CONDITION. */
263.        ULTIMATE_FALLTHRU_LABEL, /* Indicates label available as branch target
264.                when conditional test has logical value stored in ULTIMATE_
265.                FALLTHRU_CONDITION.  If used as a branch target, FALLTHRU_
266.                LABEL_USED must be set true (by CEP) to insure next sequential
267.                instruction following conditional expression receives label
268.                definition. */
269.        UNIQUE_LABEL_ID  /* One character unique to this call of CEP used to
270.                insure labels generated by this call will differ from all other
271.                labels, even others within the same macro (particularly for DO). */
```

"IF" Macro -- 21 June 1973

```
11001.   macro  IF  (USER_NAME;  REL=, MASK=, EXIT=, ELSE=)

11003.           /* Initiate a block in the structure.  Save any information needed
11004.               by ELSE or FI.  For ASYNCH type, generate branch around block.
11005.               For normal IF (EXIT= not specified), generate conditional expression
11006.               tests with branch around block (or to ELSE) for false and fall
11007.               through for true;  if EXIT= specified, then generate branch to
11008.               proper block end for true, fall through for false, and delete
11009.               IF block from structure.  Put USER_NAME on first executable
11010.               instruction if one specified. */
11011.       bit
11012.           VALID_EXIT
11013.           /* VALID_EXIT is true if EXIT= was specified and no errors have
11014.               been found to cause the EXIT to be ignored.  */
11015.       char
11016.           EXIT_LABEL,  /* Label for EXIT= branch, when deferred until
11017.                   after block count has been incremented. */
11018.           LABEL  /* Outstanding label, waiting to be generated. */

11020.       call TRACE_PRINTER (  ; 'IF')
11021.           /* Prints macro name "IF" in mnote if tracing on.  */
11022.       call PUSH_NEW_BLOCK (USER_NAME;
11023.               BLOCK_TYPE_VALUE='IF',
11024.               OPERAND1_VALUE=ELSE,
11025.               END_LABEL_VALUE=true)
11026.           /* Define new block;  add to stack.  Initialize block specifications.
11027.               Assume block will require an END label.  Note block type and save
11028.               name of ELSE block if one specified here.  Set up unique
11029.               BLOCK_LABEL_PREFIX for use in generating unique labels. */
11030.       if ERROR_OCCURRED /* during PUSH_NEW_BLOCK (viz., stack overflow) */
11031.         then
11032.           mexit
11033.         fi
11034.       if REL ¬= '' or MASK ¬= ''
11035.         then
11036.           mnote (8, 'STRC1102 REL= OR MASK= NOT IN PARENTHESES—IGNORED')
11037.         fi
11038.       LABEL := USER_NAME
11039.           /* Generate USER_NAME at first opportunity. */
11040.       VALID_EXIT := (EXIT ¬= '')
11041.           /* Set VALID_EXIT to the truth of whether EXIT= was specified. */
11042.       if SYSLIST(1,1) = 'ASYNCH'
11043.         then /* Either "IF ASYNCH" or "IF (ASYNCH)" was entered. */
11044.           call IF_ASYNCH_BRANCH  /* Generate branch around block. */
11045.         else
11046.           call IF_SET_CONDITIONAL_TEST_SPECS
11047.               /* Set all conditional test specifications in globals required to
11048.                   define the action to be performed by the conditional test
11049.                   generators. */
11050.           if VALID_EXIT  /* i.e., if EXIT specified and still valid... */
11051.             then
11052.               call IF_EXIT_SPECS
11053.                   /* Reset conditional test specs according to EXIT target provided
11054.                       no conflicting parameters exist (in that case, set VALID_EXIT to
11055.                       false). */
11056.             fi
11057.           call IF_CONDITIONAL_GENERATOR
11058.               /* Generate conditional tests according to IF macro operands and
11059.                   the current conditional test specs. */
11060.         fi
11061.       call IF_BLOCK_COUNT
11062.           /* Now that we're into the block's execution, do any debug counting
11063.               et. al. that is required. */
11064.       if LABEL ¬= ''
11065.         then
11066.           generate (LABEL || '  DS    0H')
11067.         fi
11068.       if VALID_EXIT
11069.         then
11070.               /* No IF block remains after completion of "IF  EXIT=..." macro;
11071.                   simulate presence of FI macro. */
11072.           call FI (  ;  )
11073.         fi
11074.   mend
```

```
11076.   proc IP_ASYNCH_BRANCH
11077.           /* Give error message if EXIT specified.  Generate branch to
11078.               end of IP block. */

11080.       if VALID_EXIT
11081.         then
11082.           mnote (8, 'STRC1101 EXIT= IGNORED WITH "ASYNCH"')
11083.           VALID_EXIT := false
11084.         fi
11085.       generate (LABEL || '  B     ' || BLOCK_LABEL_PREFIX || 'END')
11086.           /* Branch around asynchroneous IF block. */
11087.       LABEL := ''
11088.     corp
```

```
11090.   proc IF_SET_CONDITIONAL_TEST_SPECS
11091.           /* Set the conditional test specifications which, together with the
11092.               actual positional operands of the IF macro, define the conditions
11093.               to be generated.  The specs are:
11094.                   ULTIMATE_BRANCH_LABEL      label for target of overall test's branch
11095.                   ULTIMATE_FALLTHRU_LABEL    label to be appended to next sequential
11096.                           instruction following overall test;  will be generated
11097.                           if used in the test's branching structure
11098.                   ULTIMATE_FALLTHRU_CONDITION
11099.                           logical value which is the one upon which the overall test is
11100.                           to fall through
11101.                   FALLTHRU_LABEL_USED        false until a branch is required within
11102.                           the testing structure to the fall-through label.
11103.               All of the above are global variables. */

11105.           /* Set the normal conditional test specs. */
11106.       ULTIMATE_BRANCH_LABEL := BLOCK_LABEL_PREFIX || 'END'
11107.           /* Branch target for false result is END label——end of IF or
11108.               start of ELSE. */
11109.       ULTIMATE_FALLTHRU_LABEL := BLOCK_LABEL_PREFIX || 'BEG'
11110.           /* Fall-through label to be used is BEG. */
11111.       ULTIMATE_FALLTHRU_CONDITION := true
11112.           /* Fall through if conditional test yields true result. */
11113.       FALLTHRU_LABEL_USED := false  /* Assume not required. */
11114.     corp
```

```
11116.   proc IF_EXIT_SPECS
11117.           /* An EXIT= operand has been specified;  insure ELSE= not also
11118.               specified.  If valid, change conditional test specs to standard
11119.               for EXIT-type IF including the assigning of the branch target
11120.               as the XIT label of the block specified by the EXIT keyword.  */
11121.       char HOLD  /* Temporary. */

11123.       if ELSE ¬= '' /* If ELSE= was not omitted... */
11124.         then
11125.           mnote (8, 'STRC1103 EXIT= IGNORED WITH ELSE=')
11126.           VALID_EXIT := false
11127.         else
11128.           HOLD := ULTIMATE_BRANCH_LABEL
11129.               /* Save old branch label, we may need it yet. */
11130.           call EXIT_FIND ( ;  EXIT)
11131.               /* Sets ULTIMATE_BRANCH_LABEL to XIT label of block whose name
11132.                   is specified in the argument;  if none specified ("EXIT=*,"),
11133.                   use block surrounding IF macro;  if no such block, issue message,
11134.                   leave ULTIMATE_BRANCH_LABEL unmodified, and set ERROR_OCCURRED to
11135.                   true.  Mark target block as requiring XIT label. */
11136.           if DEBUG_BLOCKCOUNTS_REQD
11137.             then
11138.               EXIT_LABEL := ULTIMATE_BRANCH_LABEL
11139.               ULTIMATE_BRANCH_LABEL := HOLD
11140.                   /* Make EXIT-type IF act like regular IF (i.e., fall through on true)
11141.                       so we can count the number of times the exit is taken;  save the
11142.                       EXIT_LABEL for a branch after the count is made and make the
11143.                       ULTIMATE_BRANCH_LABEL whatever it would have been had this been
11144.                       a regular IF. */
11145.             else
11146.               ULTIMATE_FALLTHRU_LABEL := BLOCK_LABEL_PREFIX
11147.               ULTIMATE_FALLTHRU_CONDITION := false /* Fall through on false. */
11148.               END_LABEL_REQD(CURRENT_NEST_LEVEL) := false
11149.               if ERROR_OCCURRED /* on EXIT_FIND... */
11150.                 then
11151.                       /* Exit point not found and message has been issued.  Make branch
11152.                           point same as fall-through point and clear error (i.e., fix up
11153.                           and continue). */
11154.                   ULTIMATE_BRANCH_LABEL := BLOCK_LABEL_PREFIX
11155.                   FALLTHRU_LABEL_USED := true
11156.                       /* ...since it's also the branch label. */
11157.                   ERROR_OCCURRED := false
11158.                 fi
11159.             fi
11160.         fi
11161.     corp
```

```
11163.   proc IF_CONDITIONAL_GENERATOR
11164.           /* Generate code to pass control to the ULTIMATE_FALLTHRU_LABEL (or
11165.               to fall through to it) if the conditional test specified has the
11166.               logical value which is stored in ULTIMATE_FALLTHRU_CONDITION;
11167.               else to pass control to the ULTIMATE_BRANCH_LABEL.  Also generate
11168.               fall-through label definition if FALLTHRU_LABEL_USED was ever
11169.               turned on. */

11171.           /* Set up further specifications required by CONDITIONAL_EXPRESSION_
11172.               PROCESSOR. */
11173.       FIRST_INDEX := 1
11174.       LAST_INDEX  := N'SYSLIST
11175.           /* The operands of SYSLIST to be processed are operand 1 through the
11176.               last operand [SYSLIST(N'SYSLIST)], i.e., all of them. */
11177.       UNIQUE_LABEL_ID := 'I'
11178.           /* Used by CONDITIONAL_EXPRESSION_PROCESSOR to produce unique
11179.               labels. */
11180.       call CONDITIONAL_EXPRESSION_PROCESSOR (LABEL;  SYSLIST)
11181.           /* Generate code corresponding to the operands of the IF (refered
11182.               to collectively as SYSLIST).  Only the SYSLIST can be passed
11183.               directly as arguments;  the following variables are effectively
11184.               arguments but are passed in global variables:
11185.                   FIRST_INDEX,
11186.                   LAST_INDEX,
11187.                   ULTIMATE_BRANCH_LABEL,
11188.                   ULTIMATE_FALLTHRU_LABEL,
11189.                   ULTIMATE_FALLTHRU_CONDITION,
11190.                   UNIQUE_LABEL_ID,
11191.                   FALLTHRU_LABEL_USED.
11192.               Process operands of the SYSLIST beginning with SYSLIST(FIRST_INDEX)
11193.               through SYSLIST(LAST_INDEX) [for the IF macro, this is the entire
11194.               SYSLIST], generating the indicated tests to pass control as
11195.               indicated above.  The UNIQUE_LABEL_ID is used to insure unique
11196.               labels.  If a branch is made to the ULTIMATE_FALLTHRU_LABEL, then
11197.               FALLTHRU_LABEL_USED is set, else it is unaltered. */
11198.       if FALLTHRU_LABEL_USED
11199.         then
11200.           LABEL := ULTIMATE_FALLTHRU_LABEL
11201.         else
11202.           LABEL := ''
11203.         fi
11204.     corp
```

```
11206.   proc IF_BLOCK_COUNT
11207.           /* If debugging in progress, generate block name and/or count of
11208.              block execution.  Note that ASYNCH blocks cannot be counted. */
11209.       char TARGET  /* Temporary. */

11211.       if SYSLIST(1,1) = 'ASYNCH'
11212.         then
11213.           if DEBUG_BLOCKNAMES_REQD
11214.             then
11215.               generate ("       DC    C'" || BLOCK_NAME(CURRENT_NEST_LEVEL) ||
11216.                         "',OH'0'")
11217.                   /* Asynch branch has already occurred;  only name required. */
11218.             fi
11219.         else   /* Not ASYNCH. */
11220.           if DEBUG_BLOCKCOUNTS_REQD or DEBUG_BLOCKNAMES_REQD
11221.             then
11222.               if DEBUG_BLOCKCOUNTS_REQD
11223.                 then
11224.                     /* Generate code to increment block execution count. */
11225.                   generate (LABEL || '  LH    1,' || BLOCK_LABEL_PREFIX || 'IFC')
11226.                   LABEL := ''
11227.                   generate ('       LA    1,1(1)')
11228.                   generate ('       STH   1,' || BLOCK_LABEL_PREFIX || 'IFC')
11229.                 fi
11230.                   /* Generate branch around block name and/or block count. */
11231.               if EXIT_LABEL = ''
11232.                 then
11233.                   TARGET := BLOCK_LABEL_PREFIX || 'GO'
11234.                       /* Branch directly around block name/count. */
11235.                 else
11236.                   TARGET := EXIT_LABEL
11237.                       /* Branch to end of EXIT=  block, postponed to here so we could do the
11238.                          counting. */
11239.                 fi
11240.               generate (LABEL || '  B     ' || TARGET)
11241.               if EXIT_LABEL = ''
11242.                 then
11243.                   LABEL := TARGET
11244.                       /* Label for branch-around must be defined. */
11245.                 else
11246.                   LABEL := ''
11247.                 fi
11248.               if DEBUG_BLOCKNAMES_REQD
11249.                 then
11250.                   generate ("       DC    C'" || BLOCK_NAME(CURRENT_NEST_LEVEL) ||
11251.                             "',OH'0'")
11252.                 fi
11253.               if DEBUG_BLOCKCOUNTS_REQD
11254.                 then
11255.                   generate (BLOCK_LABEL_PREFIX || "IFC  DC    H'0' IF COUNT")
11256.                 fi
11257.             fi
11258.         fi
11259.     corp
```

"ELSE" Macro -- 26 June 1973

```
13001.   macro  ELSE  (USER_NAME;  BLEND=)
13002.          /* Initiates a block which is to be executed if the currently active
13003.              IF block was not executed, and to be skipped if the IF block was
13004.              executed.  The ELSE block terminates the IF block with a branch to
13005.              the end of the ELSE block and initiates the new ELSE
13006.              block. */
13007.       int
13008.           ELSE_BLOCK_NO  /* Block number which will be assigned to
13009.              upcoming ELSE block. */
13010.       char
13011.           IF_EXIT_LABEL  /* XIT label for IF block if one was to have been
13012.                  generated, else null. */

13014.       call TRACE_PRINTER (  ;  'ELSE')
13015.           /* Print macro name "ELSE" in mnote if tracing on. */
13016.       if CURRENT_NEST_LEVEL > NESTING_LIMIT
13017.         then
13018.           mexit
13019.         fi
13020.       call VERIFY_END(  ;  'IF', BLEND)
13021.           /* Verifies current block has the name specified by the BLEND=
13022.              operand on the ELSE macro (if any) and that it is an IF block.
13023.              Various errors receive messages and either intermediate blocks are
13024.              BLENDed as a fixup or ERROR_OCCURRED is set. */
13025.       if ERROR_OCCURRED
13026.         then
13027.           mexit
13028.         fi
13029.       if OPERAND2(CURRENT_NEST_LEVEL) = 'ELSE'
13030.         then
13031.           mnote (8, 'STRC1302 ELSE HAS ALREADY BEEN GENERATED FOR CURRENT IF')
13032.           mexit
13033.         fi
13034.       if OPERAND1(CURRENT_NEST_LEVEL) ≠ '' and ≠ USER_NAME
13035.         then
13036.               /* ELSE=elsename specified on IF macro but different (or no) label
13037.                  field on ELSE macro. */
13038.           mnote (8, 'STRC1301 ELSE=' || OPERAND1(CURRENT_NEST_LEVEL) ||
13039.                  ' SPECIFIED ON IF BLOCK ' || BLOCK_NAME(CURRENT_NEST_LEVEL))
13040.           mexit
13041.         fi
13042.       ELSE_BLOCK_NO := LAST_BLOCK_NUMBER + 1
13043.           /* Get block number of upcoming ELSE block. */
13044.       generate ('        B      $' || ELSE_BLOCK_NO || 'END')
13045.           /* Generate branch to end of ELSE block. */
13046.       if EXIT_LABEL_REQD(CURRENT_NEST_LEVEL)
13047.         then
13048.           IF_EXIT_LABEL := '$' || BLOCK_NUMBER(CURRENT_NEST_LEVEL) || 'XIT'
13049.           EXIT_LABEL_REQD(CURRENT_NEST_LEVEL) := false
13050.               /* Exit label has been postponed to FI. */
13051.         fi
13052.       call POP_OLD_BLOCK (  ;  )
13053.           /* Remove IF block from stack. */
13054.       call PUSH_NEW_BLOCK(USER_NAME;
13055.               BLOCK_TYPE_VALUE='IF',  /* Must be marked as "IF" block for FI's
13056.                                          call to VERIFY_END. */
13057.               END_LABEL_VALUE=true,
13058.               OPERAND2_VALUE='ELSE',
13059.               OPERAND3_VALUE=IF_EXIT_LABEL)
13060.           /* Generate new block in structure.  It is marked as an IF block
13061.              to simplify FI checking;  "ELSE" is stored in OPERAND2, however,
13062.              to indicate that an ELSE has been generated for this IF.  An END
13063.              label will be required for the block and, if the IF block was
13064.              marked as needing an XIT label, save it in OPERAND3. */
13065.       if USER_NAME ≠ ''
13066.         then
13067.           generate (USER_NAME || '  DS    0H')
13068.               /* Define outstanding label. */
13069.         fi
13070.     mend
```

```
15001.  macro  FI  (  ;  USER_NAME)
15002.           /* Generates end to match IF (or ELSE) block.  Standard block closing
15003.              occurs. */

15005.      call TRACE_PRINTER (  ;  'FI')
15006.           /* Print macro name "FI" in mnote if tracing on. */
15007.      if CURRENT_NEST_LEVEL > NESTING_LIMIT
15008.        then
15009.          call POP_OLD_BLOCK (  ;  )
15010.          mexit
15011.        fi
15012.      call VERIFY_END (  ;  'IF', USER_NAME)
15013.           /* Verifies current block has the name specified by the USER_NAME
15014.              operand on the FI macro (if any) and that it is an IF block.
15015.              Various errors receive messages and either intermediate blocks are
15016.              BLENDed as a fixup or ERROR_OCCURRED is set.
15017.           {Lemma:  If CURRENT_NEST_LEVEL > 0 and
15018.              [USER_NAME = '' or = BLOCK_NAME(CURRENT_NEST_LEVEL) ] and
15019.              BLOCK_TYPE(CURRENT_NEST_LEVEL) = 'IF',   then
15020.              ERROR_OCCURRED will be set false and CURRENT_NEST_LEVEL will be
15021.              unmodified.} */
15022.      if ERROR_OCCURRED
15023.        then
15024.          mexit
15025.        fi
15026.      if OPERAND1(CURRENT_NEST_LEVEL) ^= ''
15027.        then
15028.          mnote (8, 'STRC1501 ELSE BLOCK "' || OPERAND1(CURRENT_NEST_LEVEL)
15029.                       || '" NOT FOUND')
15030.        fi
15031.      call POP_OLD_BLOCK (  ;  OPERAND3(CURRENT_NEST_LEVEL))
15032.           /* Delete current block, generating END and XIT labels as required,
15033.              and popping stack.  {Lemma:  Execution of POP_OLD_BLOCK always
15034.              results in decrementing of CURRENT_NEST_LEVEL by exactly 1.} */
15035.    mend
15036.      /* {Lemma:  If CURRENT_NEST_LEVEL > 0 and
15037.          [USER_NAME = '' or = BLOCK_NAME(CURRENT_NEST_LEVEL) ] and
15038.          BLOCK_TYPE(CURRENT_NEST_LEVEL) = 'IF' at entry to FI, then
15039.          CURRENT_NEST_LEVEL will be decremented by exactly 1.} */
```

"DO" Macro -- 21 June 1973

```
21001.   macro  DO  (USER_NAME;  REL=, MASK=)
21002.           /* Create a block in the structure.  Save any information needed by
21003.              OD.  Generate conditional tests to loop according to UNTIL and WHILE
21004.              conditional expressions.  Insure UNTIL tests are not checked on first
21005.              execution.  Generate USER_NAME, if any, on first executable
21006.              instruction.  Order of generated code follows.  (Each section may be
21007.              omitted in certain cases;  see decision table in documentation for
21008.              details.)

21010.                  FLOW POINT 0:   Branch to entry point for total block.
21011.                  FLOW POINT 1:   Start of UNTIL tests, labeled $nnnU1.
21012.                  FLOW POINT 2:   Branch on success of UNTIL tests.
21013.                  FLOW POINT 3:   Fall-through on failure of UNTIL tests.
21014.                  FLOW POINT 4:   Start of WHILE tests, labeled $nnnW1.
21015.                  FLOW POINT 5:   Branch on failure of WHILE tests.
21016.                  FLOW POINT 6:   Fall-through on success of WHILE tests.
21017.                  FLOW POINT 7:   Start of internal looping code (user code between DO
21018.                                      and termination of DO loop by OD, ATEND, or
21019.                                      ONEXIT), labeled $nnnBBG.
21020.                  FLOW POINT 8:   End of internal DO code
21021.                  FLOW POINT 9:   Start of looping branch (BCT, BXH, or BXLE),
21022.                                      labeled $nnnLPB.
21023.                  FLOW POINT 10:  Branch of looping branch.
21024.                  FLOW POINT 11:  Fall-through of looping branch.
21025.                  FLOW POINT 12:  End of total block, labeled $nnnEND.

21027.              Flow points 1 through 7 are generated by DO;  8 through 12 by
21028.              TERMINATE_DO_LOOP. */

21030.       int
21031.           WHILE_INDEX, /* Index within SYSLIST of start of WHILE operands;
21032.               initially points to WHILE keyword but eventually points to start of
21033.               conditional test. */
21034.           WHILE_END_INDEX, /* Index within SYSLIST of end of WHILE operands. */
21035.           UNTIL_INDEX, UNTIL_END_INDEX, /* Same as WHILE counterparts. */
21036.           OPERAND_FORMAT, /* Column number of DO decision table (see documenta-
21037.               tion) which indicates what operands are present. */
21038.           I, /* Temporary work variable. */
21039.           LB, /* Index within SYSLIST of the looping branch (BCT, BXH, or
21040.               BXLE). */
21041.           LASTOP  /* Index of last positional operand of DO macro which is
21042.               considered valid. */

21044.       bit
21045.           WHILE_PRESENT, UNTIL_PRESENT,
21046.               /* Indicates a looping group (a looping branch and/or a
21047.                   conditional test) of the indicated type is present. */
21048.           LB_LABEL_REQ,
21049.               /* Indicates whether the looping branch requires a label. */
21050.           WHILE_COND_TEST, UNTIL_COND_TEST,
21051.               /* Indicates a conditional test of the given type is present. */
21052.           THIS_CONDITIONAL_REQD
21053.               /* Indicates the currently processed type (WHILE or UNTIL) includes
21054.                   a conditional test. */

21056.       char
21057.           LB_OPCODE_ID,
21058.               /* Opcode of looping branch:  BCT, BXH, or BXLE. */
21059.           LB_OPERAND1, LB_OPERAND2,
21060.               /* Operands of the looping branch, if present. */
21061.           LB_LOGIC_OP,
21062.               /* Logical operator ("AND" or "OR") which connects the looping branch
21063.               and the conditional test, if both are present. */
21064.           LOOPING_BRANCH_TYPE,
21065.               /* "NONE", "WHILE", or "UNTIL";  indicates position of looping
21066.                   branch. */
21067.           LABEL,
21068.               /* Any outstanding label waiting to be generated. */
21069.           FIRST_ID,
21070.               /* Label required at start of conditional test being processed. */
21071.           MAIN_OP
21072.               /* Logical operator ("AND" or "OR") which connects WHILE and UNTIL
21073.                   looping groups, if both are present. */
```

```
21075.       call TRACE_PRINTER (  ;  'DO')
21076.           /* Prints macro name "DO" in mnote if tracing on. */
21077.       call PUSH_NEW_BLOCK(USER_NAME;  BLOCK_TYPE_VALUE='DO')
21078.           /* Define new block;  add to stack.  Initialize block specifications.
21079.                Note block type and set up unique BLOCK_LABEL_PREFIX for use in
21080.                generating unique labels. */
21081.       if ERROR_OCCURRED
21082.         then
21083.           mexit
21084.         fi
21085.       if REL ≠ '' or MASK ≠ ''
21086.         then
21087.           mnote (8, 'STRC2113 REL= OR MASK= NOT IN PARENTHESES—IGNORED')
21088.         fi
21089.       LABEL := USER_NAME
21090.       call DO_SCAN_OPERANDS
21091.           /* Collect scaning information and looping branch (BCT, BXH, and
21092.                BXLE) information from operands.  Set OPERAND_FORMAT based on
21093.                these values. */
21094.       if OPERAND_FORMAT ≠ 0 and ≠ 10 and
21095.                ≠ 12 and ≠ 19
21096.         then
21097.           END_LABEL_REQD(CURRENT_NEST_LEVEL) := true
21098.         fi
21099.       if DEBUG_BLOCKCOUNTS_REQD
21100.         then /* Generate reset of current loop count. */
21101.           generate (LABEL || ' SR     1,1')
21102.           LABEL := ''
21103.           generate ('          STH    1,' || BLOCK_LABEL_PREFIX || 'DOL')
21104.         fi
21105.       if OPERAND_FORMAT ≠ 0
21106.         then /* Not infinite loop. */
21107.           call DO_BRANCH_FOR_LOOP_ENTRY
21108.                /* Generate flow point 0 branch, if required, to proper label to insure
21109.                     UNTIL tests are not made before first loop. */
21110.           call DO_GENERATE_ALL_CONDITIONAL_TESTS
21111.                /* Cause WHILE and UNTIL tests to be generated (flow points 1 through
21112.                     6) with proper labels. */
21113.         fi
21114.       call DO_LABEL_BLOCK
21115.           /* Store begin label (flow point 7) into IABEL. */
21116.       call DO_INFO_SAVE
21117.           /* Insert into stack all information required by TERMINATE_DO_LOOP to
21118.                close loop (flow points 8 through 12). */
21119.       call DO_TRACE_COUNTERS
21120.           /* Generate any debugging counters, etc. */
21121.       if LABEL ≠ ''
21122.         then
21123.           generate (LABEL || '  DS    0H')
21124.         fi
21125.     mend
```

```
21127.   proc DO_SCAN_OPERANDS
21128.           /* Collect WHILE_INDEX, WHILE_END_INDEX, UNTIL_INDEX, UNTIL_END_INDEX,
21129.              (limits of corresponding conditional test's operands within the
21130.              SYSLIST) and note in WHILE_COND_TEST and UNTIL_COND_TEST whether
21131.              the corresponding keywords include a conditional test to be
21132.              generated;  set looping branch information (LOOPING_BRANCH_TYPE,
21133.              LB_OPCODE_ID, LB_OPERAND1, LB_OPERAND2, LB_LABEL_REQ, and LB_LOGIC_OP)
21134.              which must be passed to TERMINATE_DO_LOOP to close loop;  and set
21135.              OPERAND_FORMAT (case number code from decision table). */

21137.       call DO_FIND_KEYWORDS_AND_PRESENCE
21138.           /* Put operand index of "WHILE" and "UNTIL" keywords into xxxxx_INDEX
21139.              (or set to 0 if omitted) and note in xxxxx_PRESENT whether these
21140.              looping groups exist.  Set LASTOP to index of last valid operand
21141.              in the SYSLIST. */
21142.       call DO_FIND_END_INDEXES_AND_MAIN_OP
21143.           /* For each type xxxxx (WHILE and UNTIL) which is present, put index
21144.              of the last operand of looping group for that type into xxxxx_END_
21145.              INDEX;  if both present, find logic operator which connects them
21146.              and put it into MAIN_OP, else put in null string. */
21147.       call DO_LOOPING_BRANCH_AND_FIRST_OPERAND
21148.           /* Collect looping branch information and step WHILE_INDEX and
21149.              UNTIL_INDEX to first operand of conditional test (not including
21150.              looping branch and following operator) or set to zero if not present.
21151.              Also set WHILE_ and UNTIL_COND_TEST to indicate presence of
21152.              conditional tests. */
21153.       call DO_SET_FORMAT
21154.           /* Set type of operands according to decision table, using
21155.              WHILE_PRESENT, UNTIL_PRESENT, MAIN_OP, WHILE_COND_TEST,
21156.              UNTIL_COND_TEST, LOOPING_BRANCH_TYPE, and LB_LOGIC_OP to make
21157.              decision. */
21158.     corp
```

```
21160.   proc DO_FIND_KEYWORDS_AND_PRESENCE
21161.         /* Find operand index of 'WHILE' and 'UNTIL' keywords;  set to zero
21162.            if omitted.  Put index of last valid operand in LASTOP.
21163.            Set WHILE_PRESENT and UNTIL_PRESENT true if corresponding looping
21164.            group is present, else set to false. */

21166.      WHILE_INDEX, UNTIL_INDEX := 0
21167.      if SYSLIST(1) ≠ 'WHILE' and ≠ 'UNTIL' and
21168.              ≠ 'FOREVER' and ≠ ''
21169.        then
21170.          mnote (8, 'STRC2108 FIRST OPERAND MUST BE "WHILE", "UNTIL", ' ||
21171.                  '"FOREVER", OR OMITTED')
21172.        fi
21173.      I := 1
21174.      LASTOP := N'SYSLIST  /* Assuming they're all valid. */
21175.      while I ≤ LASTOP
21176.        do /* Search for WHILE and UNTIL keywords. */
21177.          if SYSLIST(I) = 'WHILE'
21178.            then
21179.              if WHILE_INDEX = 0
21180.                then /* No WHILE found before. */
21181.                  WHILE_INDEX := I
21182.                else
21183.                  mnote (8, 'STRC2101 OPERANDS AFTER SECOND "WHILE" IGNORED')
21184.                  LASTOP := I - 1
21185.                fi
21186.            else /* Operand was not "WHILE". */
21187.              if SYSLIST(I) = 'UNTIL'
21188.                then
21189.                  if UNTIL_INDEX = 0
21190.                    then /* No UNTIL found before. */
21191.                      UNTIL_INDEX := I
21192.                    else
21193.                      mnote (8, 'STRC2102 OPERANDS AFTER SECOND "UNTIL" IGNORED')
21194.                      LASTOP := I - 1
21195.                    fi
21196.                fi
21197.              fi
21198.          I := I + 1
21199.        od /* {Termination:  I is incremented and N'SYSLIST is fixed during
21200.              loop;  LASTOP ≤ N'SYSLIST.  I would eventually exceed N'SYSLIST
21201.              so it must eventually exceed LASTOP.} */
21202.      if WHILE_INDEX > 1 and UNTIL_INDEX > 1
21203.        then /* Garbage operands are present. */
21204.          mnote (8, 'STRC2114 SUPERFLUOUS LOOPING GROUP IGNORED')
21205.        fi
21206.          /* Decide whether WHILE and UNTIL looping groups are present.  The
21207.             possible operand formats are:
21208.                  DO   UNTIL,<looping-group>
21209.                  DO   WHILE,<looping-group>
21210.                  DO   WHILE,<looping-group>,<and/or>,UNTIL,<looping-group>
21211.                  DO   UNTIL,<looping-group>,<and/or>,WHILE,<looping-group>
21212.                  DO      [No operand or single operand "FOREVER"
21213.                              means infinite loop.] */
21214.      UNTIL_PRESENT := (UNTIL_INDEX > 0)
21215.      WHILE_PRESENT := (WHILE_INDEX > 0 or
21216.             UNTIL_INDEX > 1 or
21217.             ((¬ UNTIL_PRESENT) and LASTOP > 0))
21218.          /* Last two alternatives are only to fix up when WHILE was
21219.             omitted. */
21220.    corp
```

```
proc DO_FIND_END_INDEXES_AND_MAIN_OP
        /* For each type xxxxx (WHILE and UNTIL), put index of last operand of
            looping groups for that type into xxxxx_END_INDEX;  if both are
            present, find logic operator which connects them and put it into
            MAIN_OP, else MAIN_OP := ''. WHILE_INDEX and UNTIL_INDEX currently
            point to the corresponding keyword or are zero if the corresponding
            keyword is omitted or implied (due to error). */

    MAIN_OP := ''
    WHILE_END_INDEX, UNTIL_END_INDEX := LASTOP /* As initial guess. */
    if LASTOP = 1 and SYSLIST(1) = 'FOREVER'
      then
        WHILE_PRESENT, UNTIL_PRESENT := false
      else
        if WHILE_PRESENT
          then
            if UNTIL_PRESENT
              then
                I := WHILE_INDEX - 1
                if UNTIL_INDEX < WHILE_INDEX
                  then /* UNTIL is first:  "DO  UNTIL,<test>,<and/or>,WHILE,<test>" */
                    UNTIL_END_INDEX := I - 1  /* Point at end of UNTIL. */
                    if SYSLIST(I) ≠ 'AND' and ≠ 'OR'
                      then
                        UNTIL_END_INDEX := I  /* Error message will be printed later. */
                      fi
                  else /* WHILE is first:  "DO  WHILE,<test>,<and/or>,UNTIL,<test>" */
                    I := UNTIL_INDEX - 1  /* Point I at <and/or>. */
                    WHILE_END_INDEX := I - 1
                    if SYSLIST(I) ≠ 'AND' and ≠ 'OR'
                      then
                        WHILE_END_INDEX := I  /* Error message will be printed later. */
                      fi
                  fi
                if WHILE_INDEX = WHILE_END_INDEX
                  then
                        /* One of the following was entered:
                            "DO  WHILE,UNTIL,•••" or
                            "DO  WHILE,AND,UNTIL,•••" or
                            "DO  AND,UNTIL,•••"     */
                    mnote (8, 'STRC2109 WHILE TEST IS VOID—IGNORED')
                    WHILE_PRESENT := false
                  else
                    if UNTIL_INDEX = UNTIL_END_INDEX
                      then
                            /* One of the following was entered:
                                "DO  UNTIL,WHILE,•••"  or  "DO  UNTIL,AND,WHILE,•••" */
                        mnote (8, 'STRC2111 UNTIL TEST IS VOID—IGNORED')
                        UNTIL_PRESENT := false
                      fi
                  fi
              else /* WHILE_PRESENT but not UNTIL_PRESENT. */
                if WHILE_INDEX = WHILE_END_INDEX /* Which is equal to LASTOP. */
                  then /* "DO  WHILE" with no other operands. */
                    WHILE_PRESENT := false  /* Ignore to get infinite DO. */
                  fi
              fi
          else
            if UNTIL_PRESENT and UNTIL_INDEX = UNTIL_END_INDEX
              then /* "DO  UNTIL" with no other operands. */
                UNTIL_PRESENT := false /* Ignore to get infinite DO. */
              fi
          fi
        if WHILE_PRESENT and UNTIL_PRESENT
          then
            MAIN_OP := 'AND' /* Assumed. */
            if SYSLIST(I) = 'OR'
              then
                MAIN_OP := 'OR'
              else
                if SYSLIST(I) ≠ 'AND'
                  then
                    mnote (8, 'STRC2110 LOGIC OPERATOR BETWEEN "WHILE" AND "UNTIL" ' ||
                            'OMITTED—"AND" ASSUMED')
                  fi
              fi
          fi
      fi
 corp
```

•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•

```
21302.   proc DO_LOOPING_BRANCH_AND_FIRST_OPERAND
21303.          /* Step WHILE_INDEX and UNTIL_INDEX to first operand of conditional
21304.              test or set to zero if not present.  Collect all looping branch
21305.              information.  Set WHILE_ and UNTIL_COND_TEST to true if appropriate
21306.              conditional test is present (as opposed to only a looping branch). */

21308.          /* Assume no looping branch. */
21309.        LB := 0
21310.        LOOPING_BRANCH_TYPE := 'NONE'
21311.        if UNTIL_PRESENT
21312.          then
21313.            I, UNTIL_INDEX := UNTIL_INDEX + 1
21314.                /* Move UNTIL_INDEX from pointing at "UNTIL" to pointing at first
21315.                    UNTIL operand. */
21316.            if SYSLIST(I,1) = 'BCT' or = 'BXLE' or = 'BXH'
21317.              then
21318.                LOOPING_BRANCH_TYPE := 'UNTIL'
21319.                UNTIL_COND_TEST := (UNTIL_END_INDEX > I)
21320.                LB := I
21321.                    /* UNTIL_INDEX is still pointing at the looping branch;  we aren't
21322.                        sure how far to advance it yet. */
21323.              else
21324.                UNTIL_COND_TEST := true
21325.              fi
21326.          else
21327.            UNTIL_INDEX, UNTIL_END_INDEX := 0
21328.            UNTIL_COND_TEST := false
21329.                /* Turn off all UNTIL stuff. */
21330.          fi
21331.        if WHILE_PRESENT
21332.          then
21333.            I, WHILE_INDEX := WHILE_INDEX + 1
21334.                /* Move WHILE_INDEX from pointing at "WHILE" to point at first
21335.                    WHILE operand. */
21336.            if SYSLIST(I,1) = 'BCT' or = 'BXLE' or = 'BXH'
21337.              then
21338.                if LOOPING_BRANCH_TYPE = 'NONE'
21339.                  then /* There is no UNTIL looping branch. */
21340.                    LOOPING_BRANCH_TYPE := 'WHILE'
21341.                    WHILE_COND_TEST := (WHILE_END_INDEX > I)
21342.                    LB := I
21343.                        /* WHILE_INDEX is still pointing at looping branch;  we aren't sure
21344.                            how far to advance it yet. */
21345.                    if SYSLIST(I,1) = 'BCT'
21346.                      then
21347.                        mnote (4, 'STRC2103 WARNING--"WHILE,(BCT,..." WILL LOOP ONE' ||
21348.                                  ' LESS TIME THAN VALUE IN REGISTER')
21349.                      fi
21350.                    if MAIN_OP = 'OR'
21351.                      then
21352.                        mnote (4, 'STRC2104 WARNING--LOOPING BRANCH MAY NOT BE ' ||
21353.                                  'EXECUTED ON EVERY ITERATION')
21354.                      fi
21355.                  else /* There is also an UNTIL looping branch. */
21356.                    mnote (8, 'STRC2105 TWO LOOPING BRANCHES INVALID IN "DO"--' ||
21357.                              '"WHILE" IGNORED')
21358.                    WHILE_PRESENT, WHILE_COND_TEST := false
21359.                    MAIN_OP := ''
21360.                    WHILE_INDEX, WHILE_END_INDEX := 0
21361.                  fi
21362.              else
21363.                WHILE_COND_TEST := true
21364.              fi
21365.          else
21366.            WHILE_INDEX, WHILE_END_INDEX := 0
21367.            WHILE_COND_TEST := false
21368.          fi
21369.        call DO_LOOPING_BRANCH_PROCESS
21370.            /* Collect looping branch information and advance WHILE_ or
21371.                UNTIL_INDEX over looping branch operands. */
21372.    corp
```

```
21374.   proc DO_LOOPING_BRANCH_PROCESS
21375.         /* Collect looping branch information:
21376.                LB_OPCODE_ID     'BCT', 'BXLE' or 'BXH'
21377.                LB_OPERAND1      First looping-branch operand
21378.                LB_OPERAND2      Second operand, null (or garbage) for BCT
21379.                LB_LOGIC_OP      Logic operand connecting looping branch to
21380.                                    rest of WHILE or UNTIL.
21381.                LB_LABEL_REQ     Indicates whether looping branch will need
21382.                                    a label.
21383.             Also step WHILE_ or UNTIL_INDEX over looping branch.  If any looping
21384.             branch is present, LB contains it's index;  else, LB = 0. */
21385.      int OP_COUNT  /* Number of operands looping branch needs. */

21387.      LB_OPCODE_ID, LB_OPERAND1, LB_OPERAND2, LB_LOGIC_OP := ''
21388.        /* Assume no looping branch is present. */
21389.      if LB ¬= 0
21390.        then
21391.          LB_OPCODE_ID := SYSLIST(LB,1)
21392.          if LB_OPCODE_ID = 'BCT'
21393.            then
21394.              OP_COUNT := 2
21395.            else
21396.              OP_COUNT := 3
21397.            fi
21398.          if N'SYSLIST(LB) = 1
21399.            then /* Not a sublist */
21400.              mnote (8, 'STRC2112 PARENTHESES OMITTED AROUND ' || SYSLIST(LB))
21401.            else /* Given as a sublist. */
21402.              LB_OPERAND1 := SYSLIST(LB,2)
21403.              LB_OPERAND2 := SYSLIST(LB,3)
21404.              if N'SYSLIST(LB) ¬= OP_COUNT
21405.                then
21406.                  mnote (8, 'STRC2106 INVALID NUMBER OF OPERANDS FOR ' || LB_OPCODE_ID)
21407.                fi
21408.              LB := LB + 1
21409.                  /* LB should now point to logical operator which connects looping
21410.                       branch to conditional test, if both are present. */
21411.              if (LOOPING_BRANCH_TYPE = 'WHILE' and WHILE_COND_TEST)
21412.                       or (LOOPING_BRANCH_TYPE = 'UNTIL' and UNTIL_COND_TEST)
21413.                then
21414.                  LB_LOGIC_OP := SYSLIST(LB)
21415.                  if LB_LOGIC_OP = 'AND' or = 'OR'
21416.                    then
21417.                      LB := LB + 1 /* Step LB past logic operator. */
21418.                    else
21419.                      mnote (8, 'STRC2107 ' || LB_LOGIC_OP ||
21420.                                ' INVALID AFTER LOOPING BRANCH--"AND" INSERTED')
21421.                      LB_LOGIC_OP := 'AND'
21422.                    fi
21423.                  if LOOPING_BRANCH_TYPE = 'WHILE'
21424.                    then
21425.                      WHILE_INDEX := LB
21426.                    else
21427.                      UNTIL_INDEX := LB
21428.                    fi
21429.                      /* Set xxxxx_INDEX to point at start of conditional test, if any. */
21430.                fi
21431.            fi
21432.        fi
21433.      LB_LABEL_REQ := (LOOPING_BRANCH_TYPE = 'WHILE')
21434.    corp
```

```
21436.   proc DO_SET_FORMAT
21437.           /* Set OPERAND_FORMAT according to decision table (see documen-
21438.              tation).  */

21440.       if WHILE_PRESENT
21441.         then
21442.           if LOOPING_BRANCH_TYPE = 'WHILE'
21443.             then
21444.               if WHILE_COND_TEST
21445.                 then
21446.                   if LB_LOGIC_OP = 'AND'
21447.                     then
21448.                       if UNTIL_PRESENT
21449.                         then
21450.                           if MAIN_OP = 'AND'
21451.                             then
21452.                               OPERAND_FORMAT := 15
21453.                             else
21454.                               OPERAND_FORMAT := 16
21455.                             fi
21456.                         else
21457.                           OPERAND_FORMAT := 13
21458.                         fi
21459.                     else
21460.                       if UNTIL_PRESENT
21461.                         then
21462.                           if MAIN_OP = 'AND'
21463.                             then
21464.                               OPERAND_FORMAT := 17
21465.                             else
21466.                               OPERAND_FORMAT := 18
21467.                             fi
21468.                         else
21469.                           OPERAND_FORMAT := 14
21470.                         fi
21471.                     fi
21472.                 else
21473.                   if UNTIL_PRESENT
21474.                     then
21475.                       if MAIN_OP = 'AND'
21476.                         then
21477.                           OPERAND_FORMAT := 11
21478.                         else
21479.                           OPERAND_FORMAT := 12
21480.                         fi
21481.                     else
21482.                       OPERAND_FORMAT := 10
21483.                     fi
21484.                 fi
```

```
21486.                  /* if WHILE_PRESENT then
21487.                        if LOOPING_BRANCH_TYPE = 'WHILE' then ***   */

21489.              else
21490.                if LOOPING_BRANCH_TYPE = 'UNTIL'
21491.                  then
21492.                    if UNTIL_COND_TEST
21493.                      then
21494.                        if LB_LOGIC_OP = 'AND'
21495.                          then
21496.                            if MAIN_OP = 'AND'
21497.                              then
21498.                                OPERAND_FORMAT := 7
21499.                              else
21500.                                OPERAND_FORMAT := 9
21501.                              fi
21502.                          else
21503.                            if MAIN_OP = 'AND'
21504.                              then
21505.                                OPERAND_FORMAT := 6
21506.                              else
21507.                                OPERAND_FORMAT := 8
21508.                              fi
21509.                        fi
21510.                      else
21511.                        if MAIN_OP = 'AND'
21512.                          then
21513.                            OPERAND_FORMAT := 4
21514.                          else
21515.                            OPERAND_FORMAT := 5
21516.                          fi
21517.                    fi
21518.                  else
21519.                    if UNTIL_PRESENT
21520.                      then
21521.                        if MAIN_OP = 'AND'
21522.                          then
21523.                            OPERAND_FORMAT := 2
21524.                          else
21525.                            OPERAND_FORMAT := 3
21526.                          fi
21527.                      else
21528.                        OPERAND_FORMAT := 1
21529.                      fi
21530.                  fi
21531.              fi
```

```
21533.              /* if WHILE_PRESENT then ••• */

21535.          else
21536.            if UNTIL_PRESENT
21537.              then
21538.                if LOOPING_BRANCH_TYPE = 'UNTIL'
21539.                  then
21540.                    if UNTIL_COND_TEST
21541.                      then
21542.                        if LB_LCGIC_OP = 'AND'
21543.                          then
21544.                            OPERAND_FORMAT := 22
21545.                          else
21546.                            OPERAND_FORMAT := 21
21547.                          fi
21548.                      else
21549.                        OPERAND_FCRMAT := 19
21550.                      fi
21551.                  else
21552.                    OPERAND_FORMAT := 20
21553.                  fi
21554.              else
21555.                OPERAND_FORMAT := 0
21556.              fi
21557.          fi
21558.      corp
```

```
21560.   proc DO_BRANCH_FOR_LOOP_ENTRY
21561.           /* Generate branch at flow point 0, if required, to proper label to
21562.              ensure UNTIL tests are not made before first loop. */

21564.       docase OPERAND_FORMAT ifany
21565.         of
21566.           case (2,3,6-9)
21567.                /* Branch around UNTIL conditional test to WHILE conditional test. */
21568.             generate (LABEL || ' B      ' || BLOCK_LABEL_PREFIX || 'W1')
21569.             LABEL := ''
21570.           esac
21571.           case (10-18)
21572.                /* Branch to WHILE looping branch first. */
21573.             generate (LABEL || ' B      ' || BLOCK_LABEL_PREFIX || 'LPB')
21574.             LABEL := ''
21575.           esac
21576.           case (20,21,22)
21577.                /* Branch around UNTIL conditional test to DO internal code. */
21578.             generate (LABEL || ' B      ' || BLOCK_LABEL_PREFIX || 'BEG')
21579.             LABEL := ''
21580.           esac
21581.         esacgd
21582.     corp
```

```
21584.   proc DO_GENERATE_ALL_CONDITIONAL_TESTS
21585.           /* Cause WHILE and UNTIL conditional tests to be generated with proper
21586.              labels. */
21587.       int PASS  /* Looping index. */

21589.       PASS := 1
21590.       while PASS ≤ 2
21591.         do
21592.           if PASS = 1
21593.             then
21594.               call DO_UNTIL_PREPROCESS
21595.             else
21596.               call DO_WHILE_PREPROCESS
21597.             fi
21598.               /* The DO_xxxxx_PREPROCESS proc must set:
21599.                    THIS_CONDITIONAL_REQD        from xxxxx_COND_TEST
21600.                    FIRST_INDEX                  from xxxxx_INDEX
21601.                    LAST_INDEX                   from xxxxx_END_INDEX
21602.                    UNIQUE_LABEL_ID              with the first letter of xxxxx
21603.                    ULTIMATE_BRANCH_LABEL        with the branch target
21604.                    ULTIMATE_FALLTHRU_LABEL      with the fallthru name
21605.                    ULTIMATE_FALLTHRU_CONDITION  with the proper value
21606.                    FIRST_ID                     with the first label
21607.               to insure proper test generation. */
21608.           if THIS_CONDITIONAL_REQD
21609.             then
21610.               call DO_GENERATE_CONDITIONAL_SET
21611.                    /* Generate code to pass control to the ULTIMATE_FALLTHRU_LABEL (or
21612.                       to fall through to it) if the conditional test specified by
21613.                       SYSLIST(FIRST_INDEX) through SYSLIST(LAST_INDEX) has the logical
21614.                       value stored in ULTIMATE_FALLTHRU_CONDITION;  else pass control to
21615.                       the ULTIMATE_BRANCH_LABEL.  If a branch is generated to the
21616.                       ULTIMATE_FALLTHRU_LABEL, set FALLTHRU_LABEL_USED to true;
21617.                       else set it false.  Include definition of any LABEL outstanding
21618.                       before generating code. */
21619.               if PASS = 1
21620.                 then
21621.                   call DO_UNTIL_POSTPROCESS
21622.                        /* For those cases where the ULTIMATE_FALLTHRU_LABEL was not to
21623.                           follow the conditional test as the next sequential instruction,
21624.                           generate an unconditional branch to the ULTIMATE_FALLTHRU_LABEL
21625.                           and clear FALLTHRU_LABEL_USED. */
21626.                 fi
21627.               if FALLTHRU_LABEL_USED
21628.                 then
21629.                   LABEL := ULTIMATE_FALLTHRU_LABEL
21630.                        /* Generate label at next opportunity. */
21631.                 fi
21632.             fi
21633.           PASS := PASS + 1
21634.         od /* (Termination:  PASS incremented only (not modifiable by called
21635.               procs), must eventually exceed 2.) */
21636.   corp
```

"DO" Macro -- 21 June 1973

```
21638.   proc DO_UNTIL_PREPROCESS
21639.           /* Must set up THIS_CONDITIONAL_REQD, FIRST_INDEX, LAST_INDEX,
21640.                UNIQUE_LABEL_ID, ULTIMATE_BRANCH_LABEL, ULTIMATE_FALLTHRU_LABEL,
21641.                ULTIMATE_FALLTHRU_CONDITION, and FIRST_ID. */

21643.       THIS_CONDITIONAL_REQD := UNTIL_COND_TEST
21644.       if UNTIL_COND_TEST
21645.         then
21646.           FIRST_INDEX := UNTIL_INDEX
21647.           LAST_INDEX  := UNTIL_END_INDEX
21648.           ULTIMATE_BRANCH_LABEL := BLOCK_LABEL_PREFIX || 'END'
21649.               /* Flow point 2 normally connects to flow point 12. */
21650.           ULTIMATE_FALLTHRU_LABEL := BLOCK_LABEL_PREFIX || 'W1'
21651.               /* Flow point 3 usually falls through to flow point 4. */
21652.           ULTIMATE_FALLTHRU_CONDITION := false
21653.           UNIQUE_LABEL_ID := 'U'
21654.           FIRST_ID := BLOCK_LABEL_PREFIX || 'U1'
21655.           docase OPERAND_FORMAT ifany
21656.             of
21657.               case (3,8,9) /* UNTIL test ORed with WHILE test. */
21658.                 ULTIMATE_FALLTHRU_CONDITION := true
21659.                 ULTIMATE_BRANCH_LABEL := BLOCK_LABEL_PREFIX || 'BEG'
21660.               esac
21661.               case (11,15,17) /* UNTIL test ANDed with WHILE looping branch. */
21662.                 ULTIMATE_FALLTHRU_LABEL := BLOCK_LABEL_PREFIX || 'LPB'
21663.               esac
21664.               case (12,16,18) /* UNTIL test ORed with WHILE looping branch. */
21665.                 ULTIMATE_BRANCH_LABEL := BLOCK_LABEL_PREFIX || 'LPB'
21666.                 ULTIMATE_FALLTHRU_LABEL := BLOCK_LABEL_PREFIX || 'BEG'
21667.               esac
21668.               case (20,21,22) /* UNTIL conditional test only. */
21669.                 ULTIMATE_FALLTHRU_LABEL := BLOCK_LABEL_PREFIX || 'BEG'
21670.               esac
21671.             esacod
21672.         fi
21673.     corp
```

```
21675.   proc DO_WHILE_PREPROCESS
21676.           /* Must set up THIS_CONDITIONAL_REQD, FIRST_INDEX, LAST_INDEX,
21677.                UNIQUE_LABEL_ID, ULTIMATE_BRANCH_LABEL, ULTIMATE_FALLTHRU_LABEL,
21678.                ULTIMATE_FALLTHRU_CONDITION, and FIRST_ID. */

21680.       THIS_CONDITIONAL_REQD := WHILE_COND_TEST
21681.       if WHILE_COND_TEST
21682.         then
21683.           FIRST_INDEX := WHILE_INDEX
21684.           LAST_INDEX  := WHILE_END_INDEX
21685.           ULTIMATE_BRANCH_LABEL  := BLOCK_LABEL_PREFIX || 'END'
21686.               /* Flow point 5 always branches to flow point 12. */
21687.           ULTIMATE_FALLTHRU_LABEL := BLOCK_LABEL_PREFIX || 'BEG'
21688.               /* Flow point 6 always falls through to flow point 7. */
21689.           ULTIMATE_FALLTHRU_CONDITION := true
21690.           UNIQUE_LABEL_ID := 'W'
21691.           FIRST_ID := BLOCK_LABEL_PREFIX || 'W1'
21692.         fi
21693.     corp
```

```
21695.   proc DO_GENERATE_CONDITIONAL_SET
21696.           /* Generate code to pass control to the ULTIMATE_FALLTHRU_LABEL (or
21697.               to fall through to it) if the conditional test specified by
21698.               SYSLIST(FIRST_INDEX) through SYSLIST(LAST_INDEX) has the
21699.               logical value which is stored in ULTIMATE_FALLTHRU_CONDITION;
21700.               else to pass control to the ULTIMATE_BRANCH_LABEL.  Also
21701.               see that FALLTHRU_LABEL_USED is set to true if branch to
21702.               ULTIMATE_FALLTHRU_LABEL is generated. */

21704.       FALLTHRU_LABEL_USED := false
21705.       if LABEL ¬= ''
21706.         then /* A label is waiting to be generated. */
21707.           if FIRST_ID = ''
21708.             then
21709.                  /* No special label is required at the beginning of this conditional
21710.                     test. */
21711.               FIRST_ID := LABEL  /* Put the label on the conditional test. */
21712.             else
21713.                  /* We also have a label waiting for the conditional test. */
21714.               generate (LABEL || '  DS    0H')
21715.                  /* Get the IABEL label out of the way. */
21716.             fi
21717.           LABEL := ''
21718.         fi
21719.       call CONDITIONAL_EXPRESSION_PROCESSOR (FIRST_ID;  SYSLIST)
21720.           /* Generate code corresponding to the operands of the current set
21721.               (WHILE or UNTIL) of the DO operands (referred to collectively as
21722.               SYSLIST).  Only the SYSLIST can be passed directly as arguments;
21723.               the following variables are effectively arguments but are passed
21724.               in global variables:
21725.                   FIRST_INDEX,
21726.                   LAST_INDEX,
21727.                   ULTIMATE_BRANCH_LABEL,
21728.                   ULTIMATE_FALLTHRU_LABEL,
21729.                   UNIQUE_LABEL_ID,
21730.                   FALLTHRU_LABEL_USED.
21731.               Process operands of the SYSLIST beginning with SYSLIST(FIRST_INDEX)
21732.               through SYSLIST(LAST_INDEX), generating the indicated tests to pass
21733.               control as indicated above.  If a branch is made to the
21734.               ULTIMATE_FALLTHRU_LABEL, then FALLTHRU_LABEL_USED is set, else
21735.               it is unaltered. */
21736.    corp
```

```
21738.   proc DO_UNTIL_POSTPROCESS
21739.           /* Generate where required a branch to follow the UNTIL tests to
21740.              transfer control to a non-sequential ULTIMATE_FALLTHRU_LABEL.
21741.              See decision table, flow point 3.  Insure FALLTHRU_LABEL_USED
21742.              is turned off so the label will not be generated on the next
21743.              sequential instruction. */

21745.       docase OPERAND_FORMAT ifany
21746.         of
21747.           case (11, 15-18)
21748.             generate ('       B     ' || ULTIMATE_FALLTHRU_LABEL)
21749.             FALLTHRU_LABEL_USED := false
21750.           esac
21751.         esacod
21752.     corp
```

```
21754.   proc DO_LABEL_BLOCK
21755.           /* If a begin label is required, generate it. */

21757.       docase OPERAND_FORMAT ifany
21758.         of
21759.           case (0,3,5,8-12,14,16-22)
21760.             if LABEL ≠ BLOCK_LABEL_PREFIX || 'BEG'
21761.               then
21762.                     /* Begin label must be generated. */
21763.                 if LABEL ≠ ''
21764.                   then
21765.                     generate (LABEL || '  DS    0H')
21766.                   fi
21767.                 LABEL := BLOCK_LABEL_PREFIX || 'BEG'
21768.               fi
21769.           esac
21770.         esacod
21771.     corp
```

```
21773.   proc DO_INFO_SAVE
21774.           /* Insert into stack all information required to close loop at
21775.                TERMINATE_DO_LOOP. */
21776.        char B8, B10, B11
21777.           /* One character codes indicating flow point to follow points 8,
21778.                10, and 11. */

21780.        B8 := 'W' /* Assume branch at point 8 is to WHILE group (point 4). */
21781.        B10, B11 := '0'
21782.           /* Assume no looping branch (thus no branches at 10 and 11). */

21784.           /* Set B8. */
21785.        docase OPERAND_FORMAT ifany
21786.          of
21787.            case (2,3,11,12,15-18,20)
21788.                 /* UNTIL conditional test but no UNTIL looping branch. */
21789.              B8 := 'U' /* To flow point 1. */
21790.            esac
21791.            case (4-10,13,14,19,21,22)
21792.                 /* UNTIL looping branch or no UNTIL but WHILE looping branch. */
21793.              B8 := 'L' /* Fall through to point 9. */
21794.            esac
21795.            case (0) /* Infinite loop. */
21796.              B8 := 'B' /* To flow point 7. */
21797.            esac
21798.          esacod
21799.        if LOOPING_BRANCH_TYPE ≠ 'NONE'
21800.          then
21801.                 /* Set B10. */
21802.            docase OPERAND_FORMAT only
21803.              of
21804.                case (4,7,13,15,16)
21805.                  B10 := 'W' /* To flow point 4. */
21806.                esac
21807.                case (5,9-12,14,17-19,22)
21808.                  B10 := 'B' /* To flow point 7. */
21809.                esac
21810.                case (6,8,21)
21811.                  B10 := 'U' /* To flow point 1. */
21812.                esac
21813.              esacod
21814.                /* Set B11. */
21815.            docase OPERAND_FORMAT only
21816.              of
21817.                case (4,6,10-13,15,16,19,21)
21818.                  B11 := 'N' /* Fall through to flow point 12 (end of DO block). */
21819.                esac
21820.                case (5,8,14,17,18)
21821.                  B11 := 'W' /* To flow point 8. */
21822.                esac
21823.                case (7,9,22)
21824.                  B11 := 'U' /* To flow point 1. */
21825.                esac
21826.              esacod
21827.          fi
21828.        INFORMATION(CURRENT_NEST_LEVEL) := B8 || B10 || B11 || LB_LABEL_REQ ||
21829.                     false || false || false || false
21830.           /* Byte 5 is set true when the loop is terminated (by ATEND, ONEXIT,
21831.                or OD).
21832.              Byte 6 is set true when an ATEND occurs for this DO.
21833.              Byte 7 is set true when an ONEXIT occurs for this DO.
21834.              Byte 8 is set true if a FIN label is required in the OD code. */
21835.        OPERAND1(CURRENT_NEST_LEVEL) := LB_OPERAND1
21836.        OPERAND2(CURRENT_NEST_LEVEL) := LB_OPERAND2
21837.        OPERAND3(CURRENT_NEST_LEVEL) := LB_OPCODE_ID
21838.     corp
```

```
21840.   proc DO_TRACE_COUNTERS
21841.           /* If debugging, generate block name and/or counters for block and
21842.              loop execution.  */

21844.       if DEBUG_BLOCKCOUNTS_REQD or DEBUG_BLOCKNAMES_REQD
21845.         then
21846.           if DEBUG_BLOCKCOUNTS_REQD
21847.             then
21848.               generate (LABEL || '  LH    1,' || BLOCK_LABEL_PREFIX || 'DOL')
21849.               LABEL := ''
21850.               generate ('           LA    1,1(1)')
21851.               generate ('           STH   1,' || BLOCK_LABEL_PREFIX || 'DOL')
21852.               generate ('           LH    1,' || BLOCK_LABEL_PREFIX || 'DTR')
21853.               generate ('           LA    1,1(1)')
21854.               generate ('           STH   1,' || BLOCK_LABEL_PREFIX ||  'DTR')
21855.             fi
21856.               /* Generate branch around block name and/or block counts. */
21857.           generate (LABEL || '  B     ' || BLOCK_LABEL_PREFIX || 'GO')
21858.           LABEL := BLOCK_LABEL_PREFIX || 'GO'
21859.           if DEBUG_BLOCKNAMES_REQD
21860.             then
21861.               generate ("           DC    C'" || BLOCK_NAME(CURRENT_NEST_LEVEL) ||
21862.                       "',OH'0'")
21863.             fi
21864.           if DEBUG_BLOCKCOUNTS_REQD
21865.             then
21866.               generate (BLOCK_LABEL_PREFIX || "DOL  DC    H'0'  CURRENT LOOP COUNT")
21867.               generate (BLOCK_LABEL_PREFIX || "DTR  DC    H'0'  OVERALL LOOP COUNT")
21868.             fi
21869.         fi
21870.     corp
```

```
23001.   macro  ATEND  (  ;  USER_NAME)
23002.            /* The ATEND macro causes the generation of the loop-terminating code
23003.                for the surrounding DO block if such code has not yet been generated.
23004.                The target for normal loop termination is then defined to allow the
23005.                code which follows to be executed upon normal loop termination.  If
23006.                the ATEND has been preceded by an ONEXIT macro, the branch is generate
23007.                to the OD for the ONEXIT block.  */
23008.         bit
23009.             ATEND_GENNED,  /* Indicates whether ATEND has been generated previously
23010.                               for this block. */
23011.             TDL_GENNED,    /* Indicates whether the TERMINATE_DO_LOOP macro has
23012.                               been evoked for this DO by a previous macro (properly, only
23013.                               by an ONEXIT). */
23014.             FIN_LABEL_REQD  /* Indicates a branch to the label "$nnnFIN" has
23015.                               been generated and must be defined at OD time. */
23016.         char
23017.             INFO    /* Holds copy of INFORMATION(CURRENT_NEST_LEVEL). */

23019.         call TRACE_PRINTER (  ;  'ATEND')
23020.             /* Prints macro name "ATEND" in mnote if tracing on. */
23021.         if CURRENT_NEST_LEVEL > NESTING_LIMIT
23022.           then
23023.             mexit
23024.           fi
23025.         call VERIFY_END (  ;  'DO', USER_NAME)
23026.             /* Verifies current block has the name specified by the USER_NAME
23027.                operand on the ATEND macro (if any) and that it is a DO block.
23028.                Various errors receive messages and either intermediate blocks are
23029.                BLENDed as a fixup or ERROR_OCCURRED is set. */
23030.         if ERROR_OCCURRED
23031.           then
23032.             mexit
23033.           fi
23034.         INFO := INFORMATION(CURRENT_NEST_LEVEL)
23035.         ATEND_GENNED := INFO[6,1]
23036.             /* See if we've already generated an ATEND. */
23037.         if ATEND_GENNED
23038.           then
23039.             mnote (8, 'STRC2301 MORE THAN ONE "ATEND" IN BLOCK')
23040.             mexit
23041.           fi
23042.         BLOCK_LABEL_PREFIX := '$' || BLOCK_NUMBER(CURRENT_NEST_LEVEL)
23043.         TDL_GENNED := INFO[5,1]
23044.             /* See if we've already generated the loop-terminating code. */
23045.         FIN_LABEL_REQD := INFO[8,1]
23046.             /* Note whether a FIN label has already been referenced. */
23047.         if ¬ TDL_GENNED
23048.           then
23049.             call TERMINATE_DO_LOOP (  ;  )
23050.                 /* Terminate the loop by generating any necessary back branches. */
23051.           else /* TERMINATE_DO_LOOP must have been done by previous ONEXIT. */
23052.             generate ('         B     ' || BLOCK_LABEL_PREFIX || 'FIN')
23053.             FIN_LABEL_REQD := true
23054.                 /* Terminate the ONEXIT block. */
23055.           fi
23056.         if END_LABEL_REQD(CURRENT_NEST_LEVEL)
23057.           then
23058.             generate (BLOCK_LABEL_PREFIX || 'END  DS    0H')
23059.             END_LABEL_REQD(CURRENT_NEST_LEVEL) := false
23060.                 /* If normal block termination required an END label, provide it and
23061.                    note that we no longer require it. */
23062.           fi
23063.         INFORMATION(CURRENT_NEST_LEVEL) := INFO[1,4] ||
23064.                     true /* TDL has now been generated. */  ||
23065.                     true /* ATEND has now been generated. */  ||
23066.                     INFO[7,1]  ||
23067.                     FIN_LABEL_REQD  /* Forward FIN_LABEL_REQD to OD. */
23068.     mend
```

```
25001.  macro ONEXIT  (  ;  USER_NAME)
25002.          /* The ONEXIT macro causes the generation of the loop-terminating code
25003.              for the surrounding DO block if such code has not yet been generated.
25004.              The target for abnormal loop termination (EXIT macros) is then defined
25005.              to allow the code which follows to be executed upon abnormal loop
25006.              termination.  If the ONEXIT has been preceded by an ATEND macro,
25007.              the branch is generated to the OD for the ATEND block. */
25008.      bit
25009.          ONEXIT_GENNED,  /* Indicates whether ONEXIT has been generated
25010.                  previously for this block. */
25011.          TDL_GENNED,     /* Indicates whether the TERMINATE_DO_LOOP macro has
25012.                  been evoked for this DO by a previous macro (properly, only
25013.                  by an ATEND). */
25014.          FIN_LABEL_REQD  /* Indicates a branch to the label "$nnnFIN" has
25015.                  been generated and must be defined at OD time. */
25016.      char
25017.          INFO     /* Holds copy of INFORMATION(CURRENT_NEST_LEVEL). */

25019.      call TRACE_PRINTER (  ;  'ONEXIT')
25020.          /* Prints macro name "ONEXIT" in mnote if tracing on. */
25021.      if CURRENT_NEST_LEVEL > NESTING_LIMIT
25022.        then
25023.          mexit
25024.        fi
25025.      call VERIFY_END (  ;  'DO' USER_NAME)
25026.          /* Verifies current block has the name specified by the USER_NAME
25027.              operand of the ONEXIT macro (if any)  and that it is a DO block.
25028.              Various errors receive messages and either intermediate blocks are
25029.              BLENDed as a fixup or ERROR_OCCURRED is set. */
25030.      if ERROR_OCCURRED
25031.        then
25032.          mexit
25033.        fi
25034.      INFO := INFORMATION(CURRENT_NEST_LEVEL)
25035.      ONEXIT_GENNED := INFO[7,1]
25036.          /* See if we've already generated an ONEXIT. */
25037.      if ONEXIT_GENNED
25038.        then
25039.          mnote (8, 'STRC2501 MORE THAN ONE "ONEXIT" IN BLOCK')
25040.          mexit
25041.        fi
25042.      if ¬ EXIT_LABEL_REQD(CURRENT_NEST_LEVEL)
25043.        then
25044.          mnote (8, 'STRC2502 NO EXIT FOR THIS "DO"')
25045.          mexit
25046.        fi
25047.      FIN_LABEL_REQD := INFO[8,1]
25048.          /* Note whether a FIN label has already been referenced. */
25049.      BLOCK_LABEL_PREFIX := '$' || BLOCK_NUMBER(CURRENT_NEST_LEVEL)
25050.      TDL_GENNED := INFO[5,1]
25051.          /* See if we've already generated the loop-terminating code. */
25052.      if ¬ TDL_GENNED
25053.        then
25054.          call TERMINATE_DO_LOOP (  ;  )
25055.          if TDL_FALLTHRU_OCCURS
25056.            then /* Looping branch expects to fall through to END label. */
25057.              generate ('         B      ' || BLOCK_LABEL_PREFIX || 'END')
25058.            fi
25059.        else /* TERMINATE_DO_LOOP must have been done by previous ATEND. */
25060.          generate ('         B      ' || BLOCK_LABEL_PREFIX || 'FIN')
25061.          FIN_LABEL_REQD := true
25062.              /* Provide branch to FIN for ATEND block. */
25063.        fi
25064.      generate (BLOCK_LABEL_PREFIX || 'EXIT  DS    0H')
25065.      EXIT_LABEL_REQD(CURRENT_NEST_LEVEL) := false
25066.          /* Provide target for EXIT branch and note that it is no longer
25067.              needed. */
25068.      INFORMATION(CURRENT_NEST_LEVEL) := INFO[1,4] ||
25069.                  true /* TDL has now been generated. */  ||
25070.                  INFO[6,1] ||
25071.                  true /* ONEXIT has now been generated. */  ||
25072.                  FIN_LABEL_REQD  /* Forward FIN_LABEL_REQD to OD. */
25073.  mend
```

```
27001.  macro  OD  (  ;  USER_NAME)
27002.           /* Terminate DO loop if ATEND or ONEXIT have not done so and do
27003.              standard block closing. */
27004.      bit
27005.          TDL_GENNED,  /*  Indicates whether the looping code has been generated
27006.                  yet. */
27007.          FIN_LABEL_REQD  /* Indicates whether a "$nnnFIN" label is
27008.                  required.  (It is used at the end of the ATEND or ONEXIT code.) */
27009.      char
27010.          INFO   /* Holds a copy of INFORMATION(CURRENT_NEST_LEVEL). */

27012.      call TRACE_PRINTER (  ;  'OD')
27013.          /* Prints macro name "OD" in mnote if tracing on. */
27014.      if CURRENT_NEST_LEVEL ≤ NESTING_LIMIT
27015.        then
27016.          call VERIFY_END (  ;  'DO', USER_NAME)
27017.              /* Verifies current block has the name specified by the USER_NAME
27018.                  operand of the OD macro (if any) and that it is a DO block.
27019.                  Various errors receive messages and either intermediate blocks are
27020.                  BLENDed as a fixup or ERROR_OCCURRED is set.
27021.                  {Lemma:  If CURRENT_NEST_LEVEL > 0 and
27022.                      [USER_NAME = '' or = BLOCK_NAME(CURRENT_NEST_LEVEL)] and
27023.                      BLOCK_TYPE(CURRENT_NEST_LEVEL) = 'DO', then
27024.                      ERROR_OCCURRED will be set false and CURRENT_NEST_LEVEL will
27025.                      not be modified.} */
27026.          if ERROR_OCCURRED
27027.            then
27028.              mexit
27029.            fi
27030.          INFO := INFORMATION(CURRENT_NEST_LEVEL)
27031.          TDL_GENNED := INFO[5,1]
27032.          FIN_LABEL_REQD := INFO[8,1]
27033.          if ¬ TDL_GENNED
27034.            then
27035.              call TERMINATE_DO_LOOP (  ;  )
27036.                  /* Call separate macro to generate loop-terminating branches.
27037.                      {Lemma:  TERMINATE_DO_LOOP does not modify CURRENT_NEST_LEVEL.} */
27038.              else /* ATEND or ONEXIT occurred;  we may need FIN label. */
27039.                if FIN_LABEL_REQD
27040.                  then
27041.                    generate ('$' || BLOCK_NUMBER(CURRENT_NEST_LEVEL) || 'FIN  DS    0H')
27042.                  fi
27043.              fi
27044.        fi
27045.      call POP_OLD_BLOCK (  ;  )
27046.          /* Delete current block from the stack.
27047.              {Lemma:  POP_OLD_BLOCK decrements CURRENT_NEST_LEVEL by exactly
27048.                  one.} */
27049.    mend
27050.      /* {Lemma:  If CURRENT_NEST_LEVEL > 0 and
27051.              [ USER_NAME = '' or = BLOCK_NAME(CURRENT_NEST_LEVEL) ] and
27052.              BLOCK_NAME(CURRENT_NEST_LEVEL) = 'DO' at entry to OD, then
27053.              CURRENT_NEST_LEVEL will be decremented by exactly one.} */
```

"DOCASE" Macro -- 26 June 1973

```
31001.   macro  DOCASE  (USER_NAME;  INDEX, OPTION, RANGE)
31002.          /* The DOCASE macro is used to select one of its immediate subblocks
31003.              defined by CASE macros for execution.  The operands are scanned to
31004.              determine the type of case specification provided.  Depending on the
31005.              format indicated, some instructions may be generated at this time and
31006.              various data are stored in the stack to direct code generation at
31007.              the CASE and ESACCD macros. */
31008.       bit
31009.           BRANCH_TO_CASE1,  /* Initially false;  to be set true at any
31010.               time a branch is generated which would require the first CASE to
31011.               be labeled (as opposed to falling through to the first CASE). */
31012.           INDEX_RANGE_ASSURED  /* Set to true if "ONLY" option is specified
31013.                   to indicate index will take on only values represented by the
31014.                   following CASE blocks. */
31015.       char
31016.           INDEX_REG,  /* Name of register containing DOCASE index, if
31017.               in a register. */
31018.           INDEX_LENGTH, /* Length (or symbol indicating length) of index for
31019.               CHARCOMP operand. */
31020.           INDEX_TYPE,  /* Type of DOCASE index:  "R" register, "W" word (or no
31021.               index--CONDTEST type DOCASE), "H" halfword, or "B" byte (or
31022.               character string--CHARCOMP type DOCASE). */
31023.           CASE_FORMAT, /* Format of CASE macros to follow:  "GENERAL" (branch
31024.               vector and/or symbolic compares with index), "SPARSE" (symbolic
31025.               compares only), "CHARCOMP" (character string compares), "SIMPLE"
31026.               (short sequence of integers in order 1, 2, 3, ...), or
31027.               "CONDTEST" (no index on DOCASE, conditional test on each CASE
31028.               macro). */
31029.           LABEL,  /* Any outstanding label, to be generated on next executable
31030.               instruction. */
31031.           INDEX_ADDR  /* Symbolic address of byte-type or CHARCOMP index, if
31032.                   any;  else null. */

31034.           /* Ground rules:  LABEL is to be generated on the first of any
31035.               executable instruction sequence and then cleared to null;  any label
31036.               which needs to be so generated may replace a null LABEL.  BRANCH_TO_
31037.               CASE1 must be set by any branch directly or indirectly to the first
31038.               CASE (i.e., by all but falling through to the first CASE). */
```

```
31040.       call TRACE_PRINTER ( ;  'DOCASE')
31041.            /* Print macro name "DOCASE" in mnote if tracing on. */
31042.       call PUSH_NEW_BLOCK(USER_NAME;
31043.               BLOCK_TYPE_VALUE='DOCASE',
31044.               END_LABEL_VALUE=true)
31045.           /* Define new block;  add to stack.  Initialize block specifications.
31046.               Note that block will need an END label.  Set up unique
31047.               BLOCK_LABEL_PREFIX for generating unique labels. */
31048.       if ERROR_OCCURRED
31049.         then
31050.           mexit
31051.         fi
31052.       LABEL := USER_NAME
31053.           /* Generate macro's label at first opportunity. */
31054.       call DOCASE_EXTRACT_OPERANDS
31055.           /* Validate operands and issue any error-messages;  set INDEX_REG,
31056.               INDEX_TYPE, INDEX_RANGE_ASSURED, INDEX_LENGTH and CASE_FORMAT. */
31057.       if CASE_FORMAT ^= 'CONDTEST'
31058.         then
31059.           call DOCASE_INDEX_TO_REG1
31060.               /* If case format is GENERAL, SPARSE, or CHARCOMP and the index is a
31061.                   byte, save symbolic address of the index in INDEX_ADDR, otherwise
31062.                   set INDEX_ADDR to null and generate code to put index into GPR1. */
31063.           if CASE_FORMAT ^= 'SIMPLE'
31064.             then
31065.               call DOCASE_GENERAL_SETUP
31066.                   /* Generate branch to general handler for GENERAL format.  In any
31067.                       case (GENERAL, SPARSE, or CHARCOMP), advance GCASE_NEST_LEVEL for the
31068.                       GCASE stack and initialize the GCASE globals. */
31069.             fi
31070.         fi
31071.       call DOCASE_DEBUG_STUFF
31072.           /* Generates last-case variable and block-name constant if required. */
31073.       call DOCASE_INFO_SAVE
31074.           /* Store in stack all data needed by CASE and ESACOD to complete
31075.               case processing. */
31076.       if LABEL ^= ''
31077.         then
31078.           generate (LABEL || '  DS    0H')
31079.         fi
31080.     mend
```

```
31082.   proc DOCASE_EXTRACT_OPERANDS
31083.           /* Validate operands and issue any error messages;  set INDEX_REG,
31084.               INDEX_TYPE, INDEX_RANGE_ASSURED, INDEX_LENGTH, and CASE_FORMAT. */

31086.       if OPTION = 'SIMPLE' or = 'SPARSE'
31087.         then
31088.           CASE_FORMAT := OPTION
31089.         else
31090.           if OPTION = 'ONLY'
31091.             then /* Allow range specification as second operand of macro, also. */
31092.               INDEX_RANGE_ASSURED := true
31093.             else
31094.               if OPTION ≠ '' and ≠ 'IFANY'
31095.                 then
31096.                   mnote (8, 'STRC3102 ' || OPTION ||
31097.                             ' INVALID SECOND OPERAND--IGNORED')
31098.                 fi
31099.             fi
31100.           if INDEX = ''
31101.             then
31102.               CASE_FORMAT := 'CONDTEST'
31103.             else
31104.               CASE_FORMAT := 'GENERAL'
31105.               if INDEX(1) = 'IFANY' or = 'ONLY'
31106.                 then
31107.                   mnote (4, 'STRC3101 WARNING--"' || INDEX(1) ||
31108.                             '" ASSUMED AS INDEX;  USE "DOCASE ,' || INDEX(1) ||
31109.                             '" FOR RANGE SPEC')
31110.                 fi
31111.             fi
31112.         fi
31113.       if RANGE = 'ONLY'
31114.         then
31115.           INDEX_RANGE_ASSURED := true
31116.         else
31117.           if RANGE ≠ '' and ≠ 'IFANY'
31118.             then
31119.               mnote (8, 'STRC3103 "' || RANGE ||
31120.                         '" INVALID THIRD OPERAND--IGNORED')
31121.             fi
31122.         fi
31123.       INDEX_LENGTH := '0' /* Assume not CHARCOMP. */
31124.       if N'INDEX = 1 and INDEX[1,1] = '('
31125.         then
31126.             /* A one-element sublist was specified;  we take it to be a
31127.                register. */
31128.           INDEX_REG := INDEX(1)
31129.           INDEX_TYPE := 'R'  /* Index is specified as a register. */
31130.         else
31131.           INDEX_REG := ''
31132.           if N'INDEX > 1
31133.             then
31134.               INDEX_TYPE := INDEX(2)
31135.                   /* Get index type specified;  should be "W" (word), "H" (halfword),
31136.                      "B" (byte), or length of CHARCOMP index. */
31137.               if INDEX_TYPE ≠ 'W' and ≠ 'H' and ≠ 'B'
31138.                 then
31139.                   INDEX_LENGTH := INDEX_TYPE /* Operand two is length specification. */
31140.                   INDEX_TYPE := 'B'
31141.                   CASE_FORMAT := 'CHARCOMP' /* Change format to CHARCOMP. */
31142.                 fi
31143.             else
31144.               INDEX_TYPE := 'W' /* No type specified;  "W" is default. */
31145.             fi
31146.         fi
31147.     corp
```

```
31149.   proc DOCASE_INDEX_TO_REG1
31150.           /* If case format is GENERAL, SPARSE, or CHARCOMP and the index is a
31151.              byte, save symbolic address of the index in INDEX_ADDR, otherwise
31152.              set INDEX_ADDR to null and generate code to put index into GPR1.
31153.              Given:  This proc is not called for CONDTEST format. */

31155.       INDEX_ADDR := ''  /* Assume index will be stored in GPR1. */
31156.       docase INDEX_TYPE only
31157.         of
31158.           case ('R') /* Register index. */
31159.             if INDEX_REG ≠ '1'
31160.               then
31161.                 generate (LABEL || '  LR    1,' || INDEX_REG)
31162.                 LABEL := ''
31163.               fi
31164.           esac
31165.           case ('W') /* Word index. */
31166.             generate (LABEL || '  L     1,' || INDEX(1))
31167.             LABEL := ''
31168.           esac
31169.           case ('H') /* Halfword index. */
31170.             generate (LABEL || '  LH    1,' || INDEX(1))
31171.             LABEL := ''
31172.           esac
31173.           case ('B') /* Byte index. */
31174.             if CASE_FORMAT = 'SIMPLE'
31175.               then
31176.                 generate (LABEL || ' SR    1,1')
31177.                 LABEL := ''
31178.                 generate ('        IC    1,' || INDEX(1))
31179.               else
31180.                 INDEX_ADDR := INDEX(1)
31181.                     /* Postpone loading of index into register;  we may want to do CLI's.*/
31182.               fi
31183.           esac
31184.         esacod
31185.     corp
```

```
31187.   proc DOCASE_GENERAL_SETUP
31188.           /* Generate branch to beginning of general handler for general format
31189.              DOCASE.  In any case, advance GCASE_NEST_LEVEL for the GCASE stack
31190.              and initialize the GCASE globals.  It is assumed that this proc is
31191.              called only for GENERAL, SPARSE, and CHARCOMP case formats. */
31192.       int I, J  /* Temporaries. */

31194.       if CASE_FORMAT = 'GENERAL'
31195.         then
31196.           generate (LABEL || '  B     ' || BLOCK_LABEL_PREFIX || 'BEG')
31197.               /* Generate branch to general handler which is defined at ESACOD. */
31198.           LABEL := ''
31199.           BRANCH_TO_CASE1 := true /* Albeit indirectly. */
31200.         fi
31201.       GCASE_NEST_LEVEL := GCASE_NEST_LEVEL + 1 /* Advance GCASE stack. */
31202.       if GCASE_NEST_LEVEL ≤ GCASE_NEST_LIMIT
31203.         then /* Clear GCASE globals. */
31204.           MAX_CASE_VALUE(GCASE_NEST_LEVEL) := -1
31205.               /* Maximum branch vector value found. */
31206.           NEXT_COMP_LABEL_NO(GCASE_NEST_LEVEL) := 1
31207.               /* Case number for next comparison case label to be generated. */
31208.           J := GCASE_NEST_LEVEL * 256
31209.           I := J - 255
31210.           if CASE_FORMAT = 'GENERAL'
31211.             then /* Clear CASE_OCCURS bits. */
31212.               while I ≤ J
31213.                 do
31214.                   CASE_OCCURS(I) := false
31215.                   I := I + 1
31216.                 od /* {Termination:  I is incremented, J is fixed during loop;  I
31217.                           must eventually exceed J.} */
31218.             fi
31219.         else /* GCASE stack overflow. */
31220.           mnote (12, 'STRC3104 GENERAL/SPARSE/CHARCOMP DOCASE NESTING LEVEL ' ||
31221.                   GCASE_NEST_LEVEL || ' EXCEEDS MAXIMUM OF ' ||
31222.                   GCASE_NEST_LIMIT || '--MACROS MUST BE MODIFIED')
31223.         fi
31224.     corp
```

```
31226.   proc DOCASE_DEBUG_STUFF
31227.          /* Generates last-case variable and block-name constant if required. */
31228.       char X  /* Temporary. */

31230.       if DEBUG_BLOCKCOUNTS_REQD or DEBUG_BLOCKNAMES_REQD
31231.         then
31232.           if ¬ BRANCH_TO_CASE1
31233.             then
31234.                  /* Branch must be generated around the last-case variable and/or
31235.                     block-name constant.  Put target suffix into X. */
31236.               if CASE_FORMAT = 'GENERAL'
31237.                 then
31238.                       /* This case should not occur since DOCASE_GENERAL_SETUP generates
31239.                          the branch for GENERAL cases;  we include the code here for
31240.                          completeness. */
31241.                   X := 'BEG'
31242.                 else
31243.                   X := 'C1'
31244.                 fi
31245.               generate (LABEL || '  B      ' || BLOCK_LABEL_PREFIX || X)
31246.               BRANCH_TO_CASE1 := true
31247.               LABEL := ''
31248.             fi
31249.           if DEBUG_BLOCKNAMES_REQD
31250.             then
31251.               generate ("       DC    C'" || BLOCK_NAME(CURRENT_NEST_LEVEL) || "'")
31252.             fi
31253.           if DEBUG_BLOCKCOUNTS_REQD
31254.             then
31255.               generate (BLOCK_LABEL_PREFIX || "LSC  DC    X'00'  LAST CASE NUMBER")
31256.             fi
31257.           generate ('       DS    0H')
31258.         fi
31259.     corp
```

```
31261.   proc DOCASE_INFO_SAVE
31262.           /* Store the case counter initial value (0) in OPERAND1;  INDEX_ADDR
31263.               in OPERAND2;  CASE_FORMAT in OPERAND3;  INDEX_LENGTH in OPERAND4;  and
31264.               various switches in INFORMATION. */

31266.       OPERAND1(CURRENT_NEST_LEVEL) := '0' /* Case counter. */
31267.       OPERAND2(CURRENT_NEST_LEVEL) := INDEX_ADDR /* Byte index address, */
31268.       OPERAND3(CURRENT_NEST_LEVEL) := CASE_FORMAT
31269.       OPERAND4(CURRENT_NEST_LEVEL) := INDEX_LENGTH
31270.       INFORMATION(CURRENT_NEST_LEVEL) :=
31271.           BRANCH_TO_CASE1 || false || true || false || INDEX_RANGE_ASSURED)
31272.           /* Information:
31273.               Byte 1:  Indicates whether first CASE requires a label.
31274.               Byte 2:  Indicates whether a MISC CASE has been found.
31275.               Byte 3:  Indicates whether all self-defined operands are divisible
31276.                        by 4.
31277.               Byte 4:  Indicates whether any unexpected operands were found
31278.                        for general case processing (i.e., any operands which were not
31279.                        equal to their own sequential CASE number).
31280.               Byte 5:  Indicates whether index test for out-of-range value may be
31281.                        omitted.       */
31282.    corp
```

```
33001.  macro  CASE  (USER_NAME;  REL=, MASK=)
33002.          /* The CASE macro is used to specify a block of code which is one
33003.              of the alternatives for the immediately surrounding DOCASE macro.  If
33004.              CASE macro is not the immediate daughter of a DOCASE and no fixup is
33005.              possible, a BLOCK macro is substituted.  Otherwise, the information
33006.              stored by the DOCASE is extracted and the operands of the CASE are
33007.              processed to produce the necessary code for the selecting of this
33008.              block in the indicated case.  Finally, any debugging code required
33009.              is generated. */
33010.      int
33011.          CASE_COUNTER,    /* Case number for this CASE maintained in
33012.              mother info. */
33013.          COMP_LABEL_NO, /* Number to be used in next compare label to be
33014.              defined. */
33015.          I    .   /* SYSLIST index. */
33016.      bit
33017.          CASE_LABEL_REQD, /* true unless DOCASE is falling through into
33018.                  first CASE. */
33019.          INDEX_RANGE_ASSURED,  /* true if we have been assured (by
33020.                  "DOCASE ***,ONLY") that no values other than those specified
33021.                  by CASE operands will occur. */
33022.          EQUAL_TEST_OUTSTANDING,
33023.              /* Indicates that a compare for the current operand has been generated
33024.                  but the "BE" to the beginning of the block (or "BNE" around the
33025.                  block) has not been generated yet. */
33026.          RANGE_TEST_OUTSTANDING,
33027.              /* Indicates that a compare for the current range operand has been
33028.                  generated as well as the branch if below the range;  the branch
33029.                  if within the range to the beginning of the block (or "BH" around
33030.                  the block) has not been generated yet. */
33031.          MISC_FOUND, /* Indicates whether MISC has been found yet. */
33032.          MULTIPLESOF4, /* Indicates whether all the self-defining operands
33033.              of the CASE macros processed so far are multiples of 4. */
33034.          UNEXPECTED_OPERANDS_FOUND /* Indicates whteher any operands have
33035.              been found so far in the CASE macros' operands which either were
33036.              symbolic or were self-definers not equal to their own case number. */
33037.      char
33038.          CASE_FORMAT, /* Type of CASE operands expected:  GENERAL,
33039.              SPARSE, SIMPLE, CHARCOMP, or CONDTEST. */
33040.          MAMA_BLOCK_PREFIX, /* BLOCK_LABEL_PREFIX from mother DOCASE block. */
33041.          INDEX_ADDR, /* Symbolic address of byte or CHARCOMP operand. */
33042.          LABEL, /* Outstanding label waiting to be generated. */
33043.          COMP_LABEL,  /* The next label for a compare test. */
33044.          NEXT_CASE,  /* Label to be generated on next SIMPLE or CONDTEST
33045.                  CASE macro. */
33046.          INDEX_LENGTH  /* Length of CHARCOMP index. */
```

```
33048.       call TRACE_PRINTER (  ;  'CASE')
33049.           /* Print macro name "CASE" in mnote if tracing on. */
33050.       call CASE_POSITION_CHECK
33051.           /* Verifies mother block is a DOCASE or attempts fixup with up to 2
33052.                BLENDs.  Indicates whether un-fixup-able ERROR_OCCURRED. */
33053.       if ERROR_OCCURRED
33054.         then
33055.           mnote (8, 'STRC3304 "CASE" TREATED AS "BLOCK" MACRO')
33056.           call BLOCK (USER_NAME; )
33057.           mexit
33058.         fi
33059.       call PUSH_NEW_BLOCK (USER_NAME;  BLOCK_TYPE_VALUE='CASE')
33060.           /* Define new block;  add to stack.  Initialize block specifications.
33061.                Note block type.  Set up unique BLOCK_LABEL_PREFIX for use in
33062.                generating unique labels. */
33063.       if ERROR_OCCURRED /* during PUSH_NEW_BLOCK (viz., stack overflow) */
33064.         then
33065.           mexit
33066.         fi
33067.       if REL ≠ '' or MASK ≠ ''
33068.         then
33069.           mnote (8, 'STRC3310 REL= OR MASK= NOT IN PARENTHESES—IGNORED')
33070.         fi
33071.       call CASE_GET_DOCASE_INFO
33072.           /* Extract CASE_FORMAT, CASE_LABEL_REQD, CASE_COUNTER, MISC_FOUND,
33073.                NAMA_BLOCK_PREFIX, INDEX_RANGE_ASSURED, INDEX_ADDR, and INDEX_LENGTH
33074.                from mother DOCASE block. */
33075.       if USER_NAME ≠ ''
33076.         then
33077.           generate (USER_NAME || '  DS    0H')
33078.               /* Any USER_NAME on a CASE macro is just a comment since a branch to
33079.                    it will produce unpredictable results.  If one was specified, get it
33080.                    out of the way now. */
33081.         fi
33082.       if SYSLIST(1) = 'MISC'
33083.         then
33084.           call CASE_MISC_PROCESS
33085.               /* Completely process miscellaneous CASE block. */
33086.         else
33087.           if CASE_FORMAT = 'GENERAL' or = 'SPARSE' or = 'CHARCOMP'
33088.             then
33089.               if GCASE_NEST_LEVEL ≤ GCASE_NEST_LIMIT
33090.                 then
33091.                   call CASE_PROCESS_COMPARE_OPERANDS
33092.                       /* Generate code to handle all "symbolic" operands
33093.                            (i.e., all those which cannot be handled with the branch vector),
33094.                            or for all operands in the SPARSE or CHARCOMP format.  These are
33095.                            all handled by generating compare-and-branch sequences. */
33096.                   if CASE_FORMAT = 'GENERAL'
33097.                     then
33098.                       call CASE_PROCESS_VECTOR_OPERANDS
33099.                           /* Generate labels and save information about any operands which
33100.                                are to be handled via branch vector. */
33101.                     fi
33102.                 fi
33103.             else /* Must be SIMPLE or CONDTEST type. */
33104.               call CASE_SET_NAMES
33105.                   /* Set LABEL if label required on first of code (usually is;  only
33106.                        exception is when DOCASE is falling through to first CASE macro).
33107.                        Set NEXT_CASE with label to be used on next case. */
33108.               if CASE_FORMAT = 'SIMPLE'
33109.                 then
33110.                   call CASE_BCT_GEN
33111.                       /* Generate BCT instruction for this case. */
33112.                 else
33113.                   call CASE_CONDTEST_GEN
33114.                       /* Generate conditional test specified on CASE macro. */
33115.                 fi
33116.             fi
33117.         fi
33118.       call CASE_TRACE_COUNTER
33119.           /* Generate code to count this block, note last case number, and/or
33120.                display block name if appropriate debugging requested. */
33121.       call CASE_UPDATE_INFO
33122.           /* Update the information stored in mother DOCASE block. */
33123.       if LABEL ≠ ''
33124.         then
33125.           generate (LABEL || '  DS    0H')
33126.         fi
33127.     mend
```

```
proc CASE_POSITION_CHECK
        /* Varifies mother is DOCASE macro or attempts fixup by inserting up
            to two BLENDs (if that will get us to a DOCASE mother).  Indicates if
            no fixup possible in ERROR_OCCURRED. */

    ERROR_OCCURRED := false /* Assumed. */
    if BLOCK_TYPE(CURRENT_NEST_LEVEL) ≠ 'DOCASE'
      then
        mnote (8, 'STRC3301 "CASE" NOT IMMEDIATE DAUGHTER OF "DOCASE"')
        ERROR_OCCURRED := true  /* Assume no fixup possible. */
        if CURRENT_NEST_LEVEL > 1 and
                BLOCK_TYPE(CURRENT_NEST_LEVEL-1) = 'DOCASE'
          then
            mnote (8, 'STRC3302 ASSUMING "BLEND" OMITTED--INSERTED')
            call BLEND (  ;  )
            ERROR_OCCURRED := false /* Note patch up. */
          else
            if CURRENT_NEST_LEVEL > 2 and
                    BLOCK_TYPE(CURRENT_NEST_LEVEL-2) = 'DOCASE'
              then
                mnote (8, 'STRC3303 ASSUMING TWO "BLENDS" OMITTED--INSERTED')
                call BLEND (  ;  )
                call BLEND (  ;  )
                ERROR_OCCURRED := false
              fi
          fi
      fi
    corp
```

```
33158.   proc CASE_GET_DOCASE_INFO
33159.           /* Extract DOCASE information being maintained in mother's stack
33160.              position. */
33161.       char X
33162.       int MOM   /* Temporaries. */

33164.       MOM := CURRENT_NEST_LEVEL - 1
33165.       CASE_COUNTER := OPERAND1(MOM)
33166.       if SYSLIST(1) ≠ 'MISC'
33167.         then
33168.           CASE_COUNTER := CASE_COUNTER + 1
33169.         fi
33170.       INDEX_ADDR    := OPERAND2(MOM)
33171.       CASE_FORMAT   := OPERAND3(MOM)
33172.       INDEX_LENGTH  := OPERAND4(MOM)
33173.       X             := INFORMATION(MOM)
33174.       CASE_LABEL_REQD := X[1,1]
33175.       MISC_FOUND    := X[2,1]
33176.       MULTIPLESOF4  := X[3,1]
33177.       UNEXPECTED_OPERANDS_FOUND := X[4,1]
33178.       INDEX_RANGE_ASSUMED := X[5,1]
33179.       MAMA_BLOCK_PREFIX := '$' || BLOCK_NUMBER(MOM)
33180.     corp
```

```
33182.   proc CASE_PROCESS_COMPARE_OPERANDS
33183.           /* Generate compare-and-branch sequences for all "symbolic" operands
33184.              (i.e., those which cannot be handled by the branch vector:  all non-
33185.              self-defining-terms, all self-defining operands which are not in
33186.              the range 0-255 inclusive, and all "range" operands (m,n)
33187.              where either m or n is either non-self-defining or outside the
33188.              range 0-255) or for all operands if LOCASE was flagged as SPARSE or
33189.              CHARCOMP. */

33191.       I := 1 /* Start search with first operand. */
33192.       COMP_LABEL_NO := NEXT_COMP_LABEL_NO(GCASE_NEST_LEVEL)
33193.           /* Note the next compare label number. */
33194.       EQUAL_TEST_OUTSTANDING, RANGE_TEST_OUTSTANDING := false
33195.       while I ≤ N'SYSLIST
33196.         do
33197.           if (CASE_FORMAT = 'SPARSE' or = 'CHARCOMP' or
33198.                   (N'SYSLIST(I) ≤ 1 and (T'SYSLIST(I) ≠ 'N'
33199.                           or SYSLIST(I) < 0 or > 255))
33200.                   or (N'SYSLIST(I) > 1 and (T'SYSLIST(I,1) ≠ 'N' or
33201.                           SYSLIST(I,1) < 0 or > 255 or
33202.                           T'SYSLIST(I,2) ≠ 'N' or
33203.                           SYSLIST(I,2) < 0 or > 255)))
33204.             then
33205.               if EQUAL_TEST_OUTSTANDING
33206.                 then
33207.                   LABEL := BLOCK_LABEL_PREFIX || 'BEG'
33208.                   generate ('       BE    ' || LABEL)
33209.                       /* After leaving this proc, someone will generate the BEG label
33210.                          at the beginning of the block. */
33211.                   EQUAL_TEST_OUTSTANDING := false
33212.                 else
33213.                   if RANGE_TEST_OUTSTANDING
33214.                     then
33215.                       LABEL := BLOCK_LABEL_PREFIX || 'BEG'
33216.                       generate ('       BNH   ' || LABEL)
33217.                           /* Again, by leaving BEG label in LABEL, it will be generated
33218.                              after leaving this proc. */
33219.                       RANGE_TEST_OUTSTANDING := false
33220.                     else /* Must be first time through. */
33221.                       if CASE_LABEL_REQD
33222.                         then
33223.                           COMP_LABEL := MAMA_BLOCK_PREFIX || 'C' || COMP_LABEL_NO
33224.                               /* Generate label name to be attached to first instruction. */
33225.                       fi
33226.                       UNEXPECTED_OPERANDS_FOUND := true
33227.                     fi
33228.                 fi
33229.               call CASE_GEN_COMPARE
33230.                   /* Generate compare for the single compare operand at SYSLIST(I)--
33231.                      either general case non-self-definer or any SPARSE or CHARCOMP
33232.                      operand.  DOCASE index is at INDEX_ADDR unless that's null, then in
33233.                      GPR1.  Length is in INDEX_LENGTH for CHARCOMP type.  Any label to be
33234.                      generated is in COMP_LABEL;  once defined, COMP_LABEL_NO must be
33235.                      increased.  Any branch target outstanding at exit is to be
33236.                      put into COMP_LABEL.  Also on exit, EQUAL_TEST_OUTSTANDING or
33237.                      RANGE_TEST_OUTSTANDING should be set to indicate which type of
33238.                      operand was processed. */
33239.             fi
33240.           I := I + 1 /* Advance to next operand of CASE. */
33241.         od /* {Termination:  I is incremented above and not modified by
33242.                   called procs;  N'SYSLIST is fixed;  I must eventually exceed
33243.                   N'SYSLIST.} */
33244.         /* All compare operands have now been processed. */
33245.       if EQUAL_TEST_OUTSTANDING
33246.         then
33247.               /* Generate branch to next symbolic case. */
33248.           generate ('       BNE   ' || MAMA_BLOCK_PREFIX || 'C' ||
33249.                   COMP_LABEL_NO)
33250.         else
33251.           if RANGE_TEST_OUTSTANDING
33252.             then
33253.               generate ('       BH    ' || COMP_LABEL)
33254.                   /* Generate branch to next compare case.  Label was left in COMP_LABEL
33255.                      when we branched on lower end of range. */
33256.             fi
33257.         fi
33258.       NEXT_COMP_LABEL_NO(GCASE_NEST_LEVEL) := COMP_LABEL_NO
33259.           /* Store case number of next symbolic case to be defined. */
33260.     corp
```

```
33262.   proc CASE_GEN_COMPARE                                          *
33263.           /* Generate compare for the single compare operand at SYSLIST(I) —
33264.              either general case non-self-definer or any SPARSE or CHARCOMP
33265.              operand.  DOCASE index is at INDEX_ADDR unless that's null, then in
33266.              GPR1.  Length is in INDEX_LENGTH for CHARCCMP type.  Any label to be
33267.              generated is in COMP_LABEL;  any branch target at exit is to be
33268.              put into COMP_LABEL.  Also on exit, EQUAL_TEST_OUTSTANDING or
33269.              RANGE_TEST_OUTSTANDING should be set tc indicate which type of
33270.              operand was processed.  Operands may be of the form m or (m, n),
33271.              the latter implying the range from m to n.  m and n may be
33272.              self-defining terms or symbols EQUated to absolute expressions for
33273.              GENERAL or SPARSE format;  for CHARCCMP, they may be absolute or
33274.              symbolic addresses of character strings or may be literals (with the
33275.              leading "=" and, for character literals, "C" possibly omitted). */
33276.        char INSERT  /* Temporary . */
```

```
33278.      if INDEX_ADDR = ''
33279.        then /* Index is in GPR1. */
33280.          generate (COMP_LABEL || '  LA    0,' || SYSLIST(I,1))
33281.          generate ('       CH    1,0')
33282.        else /* Index is at INDEX_ADDR. */
33283.          if CASE_FORMAT = 'CHARCOMP'
33284.            then
33285.              INSERT := ''
33286.              if SYSLIST(I,1)[1,1] ≠ '='
33287.                then
33288.                  if SYSLIST(I,1)[1,1] = "'"
33289.                    then /* Character string. */
33290.                      INSERT := '=C'
33291.                    else
33292.                      if SYSLIST(I,1)[K'SYSLIST(I,1),1] = "'"
33293.                        then /* Literal without the "=" (operand ends with "'"). */
33294.                          INSERT := '='
33295.                        fi
33296.                    fi
33297.                fi
33298.              generate (COMP_LABEL || '  CLC   ' || INDEX_ADDR || '(' ||
33299.                    INDEX_LENGTH || '),' || INSERT || SYSLIST(I,1))
33300.            else
33301.              generate (COMP_LABEL || '  CLI   ' || INDEX_ADDR || ',' ||
33302.                    SYSLIST(I,1))
33303.            fi
33304.        fi
33305.      if COMP_LABEL ≠ ''
33306.        then
33307.          COMP_LABEL := ''
33308.          COMP_LABEL_NO := COMP_LABEL_NO + 1
33309.        fi
33310.      if N'SYSLIST(I) ≤ 1
33311.        then /* Operand is not a range. */
33312.          EQUAL_TEST_OUTSTANDING := true
33313.        else /* A range has been specified:  (m,n) */
33314.          if N'SYSLIST(I) > 2
33315.            then
33316.              mnote (8, 'STRC3312 ' || SYSLIST(I) ||
33317.                        ' INVALID--ONLY FIRST TWO SUBOPERANDS PROCESSED')
33318.            fi
33319.          RANGE_TEST_OUTSTANDING := true
33320.              /* Generate another label. */
33321.          COMP_LABEL := MAMA_BLOCK_PREFIX || 'C' || COMP_LABEL_NO
33322.          generate ('       BL    ' || COMP_LABEL)
33323.              /* Generate branch on out of range to next compare test label (in
33324.                 either this CASE macro or some other CASE in this DOCASE). */
33325.          if INDEX_ADDR = ''
33326.            then
33327.              generate ('       LA    0,' || SYSLIST(I,2))
33328.              generate ('       CH    1,0')
33329.            else
33330.              if CASE_FORMAT = 'CHARCOMP'
33331.                then
33332.                  INSERT := ''
33333.                      /* Go through the same business figuring out the insert for n as
33334.                         we did for m. */
33335.                  if SYSLIST(I,2)[1,1] ≠ '='
33336.                    then
33337.                      if SYSLIST(I,2)[1,1] = "'"
33338.                        then
33339.                          INSERT := '=C'
33340.                        else
33341.                          if SYSLIST(I,2)[K'SYSLIST(I,2),1] = "'"
33342.                            then
33343.                              INSERT := '='
33344.                            fi
33345.                        fi
33346.                    fi
33347.                  generate ('       CLC   ' || INDEX_ADDR || '(' || INDEX_LENGTH ||
33348.                        '),' || INSERT || SYSLIST(I,2))
33349.                else
33350.                  generate ('       CLI   ' || INDEX_ADDR || ',' || SYSLIST(I,2))
33351.                fi
33352.            fi
33353.        fi
33354.    corp
```

```
proc CASE_PROCESS_VECTOR_OPERANDS
        /* Generate labels and note that CASE_OCCURRS for any operands which
           can be handled via branch vector: viz., any of the form m or
           (m,n) where m and n are self-defining terms in the range
           0-255 inclusive.  This procedure assumes the CASE_FORMAT is general
           (not sparse). */
    int
        BASE, /* Array position in CASE_OCCURS of the case for zero. */
        OP, /* Case value currently being considered. */
        LIMIT /* High limit in range operands. */

    BASE := ((GCASE_NEST_LEVEL - 1) * 256) + 1
        /* Calculate offset in CASE_OCCURRS array for this DOCASE. */
    if N'SYSLIST > 0
      then /* One or more operands were specified. */
        I := 1 /* Start with first operand. */
        while I ≤ N'SYSLIST
          do
            if T'SYSLIST(I,1) = 'N'
              then /* m is a self-defining term. */
                OP := SYSLIST(I,1)
                if OP ≤ 255 and OP ≥ 0
                  then /* It's in the range. */
                    if T'SYSLIST(I,2) = 'O' or = 'N'
                      then /* n is self-defining or not present. */
                        if N'SYSLIST(I) > 2
                          then
                            mnote (8, 'STRC3312 ' || SYSLIST(I) ||
                                      ' INVALID--ONLY FIRST TWO SUBOPERANDS PROCESSED')
                        fi
                        if T'SYSLIST(I,2) = 'N'
                          then
                            LIMIT := SYSLIST(I,2)
                            if LIMIT ≤ 255 and ≥ 0
                              then /* n is in the right range also. */
                                if LIMIT < OP
                                  then
                                    mnote (8, 'STRC3305 ' || SYSLIST(I) || ' INVALID--' ||
                                              OP || ' ASSUMED'
                                    LIMIT := OP
                                fi
                              else
                                LIMIT := OP
                              fi
                            if LIMIT > MAX_CASE_VALUE(GCASE_NEST_LEVEL)
                              then /* We have found a new maximum case number. */
                                MAX_CASE_VALUE(GCASE_NEST_LEVEL) := LIMIT
                              fi
                            while OP ≤ LIMIT
                              do
                                CASE_OCCURS(BASE+OP) := true
                                generate (MAMA_BLOCK_PREFIX || 'G' || OP || '   DS    0H')
                                if OP ≠ CASE_COUNTER
                                  then
                                    UNEXPECTED_OPERANDS_FOUND := true
                                  fi
                                if OP/4*4 ≠ OP
                                  then
                                    MULTIPLESOF4 := false
                                  fi
                                OP := OP + 1
                              od /* (Termination:  OP is incremented, LIMIT is fixed
                                        during loop;  OP must eventually exceed LIMIT.) */
                        fi
                    fi
                fi
            fi
            I := I + 1 /* Go do next operand. */
          od /* (Termination:  I is incremented, N'SYSLIST is fixed during
                    loop;  I must eventually exceed N'SYSLIST.) */
      else /* No operands present. */
        call CASE_ASSUMED_VECTOR_CASE
          /* If all the CASE macros so far have had no operands or only
             "expected" ones (integers which match the case counter), assume
             this one matches too and generate the single operand. */
    fi
corp
```

```
33434.   proc CASE_ASSUMED_VECTOR_CASE
33435.           /* Generate label for branch vector cases with no operands.  Value used
33436.              is the next higher value than the maximum used so far.  CASE_OCCURRS
33437.              is noted, and the SELFDEP_COUNT and MAX_CASE_VALUE are updated.  If
33438.              any previous operands have occured which were not expected, a
33439.              message is printed. */
33440.       int GUESS  /* Assumed operand. */

33442.       GUESS := MAX_CASE_VALUE(GCASE_NEST_LEVEL) + 1
33443.           /* Guess at what omitted operand was intended. */
33444.       if GUESS ≤ 0
33445.         then /* First guess. */
33446.           GUESS := 1
33447.         fi
33448.       MAX_CASE_VALUE(GCASE_NEST_LEVEL) := GUESS
33449.       CASE_OCCURS(BASE+GUESS) := true
33450.       generate (MAMA_BLOCK_PREFIX || 'G' || GUESS || '  DS    0H')
33451.       if UNEXPECTED_OPERANDS_FOUND
33452.         then
33453.           mnote (4,
33454.                'STRC3306 EARLIER UNEXPECTED OPERAND IMPLIES THIS TO BE CASE ' ||
33455.                GUESS)
33456.         fi
33457.     corp
```

"CASE" Macro -- 27 June 1973

```
33459.   proc CASE_SET_NAMES
33460.           /* Set LABEL if one will be required on this SIMPLE or CONDTEST case
33461.               code (usually is;  only exception involves when DOCASE falls through
33462.               to first case).  Also set NEXT_CASE with label of next case to be
33463.               generated.  LABEL is always null at entry. */

33465.       if CASE_LABEL_REQD
33466.         then
33467.           LABEL := MAMA_BLOCK_PREFIX || 'C' || CASE_COUNTER
33468.         fi
33469.       I := CASE_COUNTER + 1
33470.       NEXT_CASE := MAMA_BLOCK_PREFIX || 'C' || I
33471.     corp
```

```
33473.   proc CASE_BCT_GEN
33474.         /* Generate BCT for this simple case.  Verify operand, if any. */

33476.      generate (LABEL || '  BCT   1,' || NEXT_CASE)
33477.      LABEL := ''
33478.      if T'SYSLIST(1) ≠ '0'
33479.        then /* An operand was specified. */
33480.          if T'SYSLIST(1) = 'N'
33481.            then /* Operand is a self-defining term. */
33482.              if SYSLIST(1) ≠ CASE_COUNTER
33483.                then
33484.                  mnote (8, 'STRC3307 OPERAND INVALID VALUE ON SIMPLE CASE ' ||
33485.                      CASE_COUNTER)
33486.                fi
33487.            else /* Operand is not self-defining term. */
33488.              mnote (8, 'STRC3309 OPERAND MUST BE SELF-DEFINING TERM OR OMITTED ' ||
33489.                  'ON SIMPLE CASE ' || CASE_COUNTER)
33490.            fi
33491.        fi
33492.    corp
```

```
33494.   proc CASE_CONDTEST_GEN
33495.           /* Generate conditional test indicated by operands. */
33496.       int OP_COUNT /* Number of operands for instruction being passed
33497.               to SIMPLE_CONDITIONAL. */

33499.       ULTIMATE_BRANCH_LABEL := NEXT_CASE
33500.       ULTIMATE_FALLTHRU_LABEL := BLOCK_LABEL_PREFIX || 'BEG'
33501.       ULTIMATE_FALLTHRU_CONDITION := true
33502.       FALLTHRU_LABEL_USED := false
33503.       FIRST_INDEX := 1
33504.       LAST_INDEX  := N'SYSLIST
33505.           /* Process entire operand list as a single conditional expression. */
33506.       UNIQUE_LABEL_ID := 'T'
33507.       call CONDITIONAL_EXPRESSION_PROCESSOR (LABEL;  SYSLIST)
33508.           /* Generate code corresponding to the operands of the CASE macro
33509.               (referred to collectively as SYSLIST).  Only the SYSLIST can be passed
33510.               directly as arguments;  the following variables are effectively
33511.               arguments but are passed in global variables:
33512.                   FIRST_INDEX,
33513.                   LAST_INDEX,
33514.                   ULTIMATE_BRANCH_LABEL,
33515.                   ULTIMATE_FALLTHRU_LABEL,
33516.                   ULTIMATE_FALLTHRU_CONDITION,
33517.                   UNIQUE_LABEL_ID,
33518.                   FALLTHRU_LABEL_USED.
33519.               Process operands of the SYSLIST beginning with SYSLIST(FIRST_INDEX)
33520.               through SYSLIST(LAST_INDEX)  (for the CASE macro, this is the entire
33521.               SYSLIST), generating the indicated test to pass control to the
33522.               ULTIMATE_FALLTHRU_LABEL if the test succeeds, else to the
33523.               ULTIMATE_BRANCH_LABEL.  The UNIQUE_LABEL_ID is used to insure
33524.               unique labels where needed.  If a branch is made to the
33525.               ULTIMATE_FALLTHRU_LABEL, then set FALLTHRU_LABEL_USED;  else
33526.               it is unaltered. */
33527.       if FALLTHRU_LABEL_USED
33528.         then
33529.           LABEL := BLOCK_LABEL_PREFIX || 'BEG'
33530.         else
33531.           LABEL := ''
33532.         fi
33533.     corp
```

```
33535.   proc  CASE_MISC_PROCESS
33536.           /* Generate label for miscellaneous block, including a branch around
33537.              this block in case we were falling through into it.  Note that a
33538.              miscellaneous block has been found and varify that no other
33539.              miscellaneous case has occurred for this DOCASE. */

33541.       if ¬ CASE_LABEL_REQD
33542.         then /* We are falling through into this block. */
33543.           generate ('       B     ' ||
33544.                   MAMA_BLOCK_PREPIX || 'C' || CASE_COUNTER)
33545.               /* Generate branch to next case number (probably C1). */
33546.         fi
33547.       if MISC_FOUND
33548.         then
33549.           mnote (8, 'STRC3311 MULTIPLE MISC CASES IN DOCASE--THIS BLOCK ' ||
33550.                   'IS DEAD CODE')
33551.         else
33552.           LABEL := MAMA_BLOCK_PREPIX || 'MSC'
33553.               /* Make MSC label outstanding (generate on next instruction).
33554.                  It is assumed that no LABEL can be outstanding when CASE_MISC_PROCESS
33555.                  is called. */
33556.           MISC_FOUND := true
33557.         fi
33558.       if INDEX_RANGE_ASSURED
33559.         then
33560.           mnote (8, 'STRC3308 "DOCASE ...,ONLY" INVALID WITH MISC')
33561.           INDEX_RANGE_ASSURED := false
33562.         fi
33563.     corp
```

```
33565.   proc  CASE_TRACE_COUNTER
33566.           /* Generate any debugging counters and/or labels requested. */

33568.       if DEBUG_BLOCKCOUNTS_REQD or DEBUG_BLOCKNAMES_REQD
33569.         then
33570.           if DEBUG_BLOCKCOUNTS_REQD
33571.             then
33572.                   /* Generate code to advance this case's counter. */
33573.               generate (LABEL || '  LH    1,' || BLOCK_LABEL_PREFIX || 'CTR')
33574.               LABEL := ''
33575.               generate ('           LA    1,1(1)')
33576.               generate ('           STH   1,' || BLOCK_LABEL_PREFIX || 'CTR')
33577.               if SYSLIST(1) = 'MISC' or CASE_COUNTER > 255
33578.                 then
33579.                   generate ('       MVI   ' || MAMA_BLOCK_PREFIX || "LSC,X'FF'")
33580.                 else
33581.                   HEX_IN := CASE_COUNTER
33582.                   call XHEX ( ; )
33583.                   gnote (*, "STRC3313 CASE DEBUG ID=X'" || HEX_OUT || "'")
33584.                   generate ('       MVI   ' || MAMA_BLOCK_PREFIX || "LSC,X'" ||
33585.                           HEX_OUT || "' CASE NUMBER FOR TRACING')
33586.               fi
33587.           fi
33588.           generate (LABEL || '  B     ' || BLOCK_LABEL_PREFIX || 'GO')
33589.           LABEL := BLOCK_LABEL_PREFIX || 'GO'
33590.               /* Branch around count and/or block name and set up label to be
33591.                   defined eventually. */
33592.           if DEBUG_BLOCKNAMES_REQD
33593.             then
33594.               generate ("       DC    C'" || BLOCK_NAME(CURRENT_NEST_LEVEL) ||
33595.                       "',0H'0'")
33596.             fi
33597.           if DEBUG_BLOCKCOUNTS_REQD
33598.             then
33599.               generate (BLOCK_LABEL_PREFIX || "CTR  DC    H'0' CASE COUNT")
33600.             fi
33601.         fi
33602.     corp
```

```
33604.   proc CASE_UPDATE_INFO
33605.           /* Returns to mother DOCASE level possibly updated information which
33606.                was extracted by CASE_GET_DOCASE_INFO. */
33607.       int MOM  /* Index level of DOCASE block. */

33609.       MOM := CURRENT_NEST_LEVEL - 1
33610.       INFORMATION(MOM) := true || MISC_FOUND || MULTIPLESOP4 ||
33611.                UNEXPECTED_OPERANDS_FOUND || INDEX_RANGE_ASSURED
33612.           /* First byte indicates case label is required on next case. */
33613.       OPERAND1(MOM) := CASE_COUNTER
33614.           /* No need to update OPERAND2 (INDEX_ADDR) or OPERAND3 (CASE_FORMAT)
33615.                or OPERAND4 (INDEX_LENGTH).  None ever change. */
33616.     corp
```

"ESAC" Macro -- 3 July 1973

```
35001.   macro  ESAC  (  ;  USER_NAME)
35002.            /* Generate end to match CASE block.  Do standard block closing, then
35003.                generate branch to end of mother DOCASE block. */

35005.       call TRACE_PRINTER (  ;  'ESAC')
35006.            /* Print macro name "ESAC" in mnote if tracing on. */
35007.       if CURRENT_NEST_LEVEL ≤ NESTING_LIMIT
35008.         then
35009.           call VERIFY_END (  ;  'CASE', USER_NAME)
35010.                /* Verifies current block has the name specified by the USER_NAME
35011.                   operand of the ESAC macro (if any) and that it is a CASE block.
35012.                   Various errors receive messages and either intermediate blocks are
35013.                   BLENDed as a fixup or ERROR_OCCURRED is set.
35014.                   {Lemma:  If CURRENT_NEST_LEVEL > 0 and
35015.                       [USER_NAME = '' or = BLOCK_NAME(CURRENT_NEST_LEVEL) ] and
35016.                       BLOCK_TYPE(CURRENT_NEST_LEVEL) = 'CASE', then
35017.                       ERROR_OCCURRED will be set false and CURRENT_NEST_LEVEL will not
35018.                       be modified.} */
35019.           if ERROR_OCCURRED
35020.             then
35021.               mexit
35022.             fi
35023.         fi
35024.       call POP_OLD_BLOCK(  ;  )
35025.            /* Delete current block, generating END and XIT labels as required, and
35026.                popping stack.  {Lemma:  POP_OLD_BLOCK decrements CURRENT_NEST_LEVEL
35027.                by exactly one.} */
35028.       if CURRENT_NEST_LEVEL ≤ NESTING_LIMIT
35029.         then
35030.           generate ('       B     $' || BLOCK_NUMBER(CURRENT_NEST_LEVEL) ||
35031.                     'END')
35032.                /* Generate branch to end of DOCASE. */
35033.           fi
35034.     mend
35035.       /* {Lemma:  If CURRENT_NEST_LEVEL > 0 and
35036.                [ USER_NAME = '' or = BLOCK_NAME(CURRENT_NEST_LEVEL) ] and
35037.                BLOCK_TYPE(CURRENT_NEST_LEVEL) = 'CASE' at entry to ESAC, then
35038.                CURRENT_NEST_LEVEL will be decremented by exactly one.} */
```

```
37001.  macro  ESACOD  (  ;  USER_NAME)
37002.          /* Generates final part of DOCASE processing:  for SIMPLE, CONDTEST,
37003.              or SPARSE type DOCASE, the EQU for the MISC block (or END of DOCASE)
37004.              to the last generated branch target is required;  for GENERAL type
37005.              DOCASE, the branch vector and the transfer to any symbolic
37006.              compares or MISC block must be generated.  Finally, the block is
37007.              popped. */
37008.      int
37009.          CASE_COUNTER, /* Holds number of last case generated. */
37010.          T,    /* Temporary. */
37011.          COMP_LABEL_NO, /* Label number of outstanding compare case. */
37012.          MAX_SD_VALUE, /* Maximum self-defined operand. */
37013.          BASE  /* Index within CASE_OCCURS array for CASE 0. */
37014.      bit
37015.          MISC_FOUND, /* Indicates whether a MISC CASE was found. */
37016.          MULTIPLESOF4, /* Indicates whether all branch-vector operands were
37017.                  multiples of 4. */
37018.          INDEX_RANGE_ASSURED,  /* true if we have been assured (by
37019.                  "DOCASE ***,ONLY") that no values other than those specified
37020.                  by CASE operands will occur. */
37021.          ANY_COMP_CASES,  /* Indicates whether any "compare" cases were
37022.                  generated (either CHARCOMP or symbolic general case operands). */
37023.          ANY_SELFDEF_CASES,  /* Indicates whether any "self-defining" cases (to
37024.                  be handled by branch vector) were generated. */
37025.          RANGE_TEST_REQD  /* Indicates that both branch vector and compare
37026.                  operands were present. */
37027.      char
37028.          CASE_FORMAT, /* Type of CASEs present:  GENERAL, SPARSE,
37029.                  CHARCOMP, SIMPLE, or CONDTEST. */
37030.          INDEX_ADDR, /* Address of DOCASE index. */
37031.          NOCASE,  /* Label for branch vector processing used for unspecified
37032.                  cases. */
37033.          LABEL   /* Any outstanding label waiting to be generated. */

37035.          /* (Ground rules:  No ESACOD proc modifies CURRENT_NEST_LEVEL.
37036.              This can be shown by referring to the cross-reference index.) */
```

```
37038.       call TRACE_PRINTER (  ;  'ESACOD')
37039.           /* Print macro name "ESACOD" in mnote if tracing on. */
37040.       if CURRENT_NEST_LEVEL ≤ NESTING_LIMIT
37041.         then
37042.           call VERIFY_END (  ;  'DOCASE', USER_NAME)
37043.               /* Verifies current block has the name specified by the USER_NAME
37044.                  operand of the ESACOD macro (if any) and that it is a DOCASE block.
37045.                  Various errors receive messages and either intermediate blocks are
37046.                  BLENDed as a fixup or ERROR_OCCURRED is set.
37047.                  {Lemma:  If CURRENT_NEST_LEVEL > 0 and
37048.                      [USER_NAME = '' or = BLOCK_NAME(CURRENT_NEST_LEVEL)] and
37049.                      BLOCK_TYPE = 'DOCASE', then
37050.                      ERROR_OCCURRED will be set false and CURRENT_NEST_LEVEL will
37051.                      not be modified.} */
37052.           if ERROR_OCCURRED
37053.             then
37054.               mexit
37055.             fi
37056.           call ESACOD_INFO_UNPACK
37057.               /* Extracts CASE_FORMAT, CASE_COUNTER, INDEX_ADDR, MISC_FOUND,
37058.                  BLOCK_LABEL_PREFIX, INDEX_RANGE_ASSURED, and MULTIPLESOF4 from
37059.                  stack. */
37060.           if CASE_FORMAT = 'GENERAL'
37061.             then
37062.               call ESACOD_GENERAL_CASE_CHOICE
37063.                   /* Generate all code to complete processing of general case. */
37064.             else
37065.               if CASE_FORMAT = 'SPARSE' or = 'CHARCOMP'
37066.                 then
37067.                   T := NEXT_COMP_LABEL_NO(GCASE_NEST_LEVEL)
37068.                       /* We need to define last compare case target. */
37069.                   GCASE_NEST_LEVEL := GCASE_NEST_LEVEL - 1
37070.                       /* Pop GCASE stack. */
37071.                 else /* CONDTEST or SIMPLE. */
37072.                   T := CASE_COUNTER + 1
37073.                       /* We need to define last conditional test target. */
37074.                 fi
37075.               if MISC_FOUND
37076.                 then
37077.                   generate (BLOCK_LABEL_PREFIX || 'C' || T || '  EQU   ' ||
37078.                             BLOCK_LABEL_PREFIX || 'MSC')
37079.                 else
37080.                   generate (BLOCK_LABEL_PREFIX || 'C' || T || '  DS    0H')
37081.                 fi
37082.           fi
37083.       fi
37084.       call POP_OLD_BLOCK (  ;  )
37085.           /* {Lemma:  POP_OLD_BLOCK decrements CURRENT_NEST_LEVEL by exactly
37086.              one.} */
37087.     mend
37088.       /* {Lemma:  If CURRENT_NEST_LEVEL > 0 and
37089.             [USER_NAME = '' or = BLOCK_NAME(CURRENT_NEST_LEVEL)] and
37090.             BLOCK_TYPE(CURRENT_NEST_LEVEL) = 'DOCASE' at entry to ESACOD, then
37091.             CURRENT_NEST_LEVEL will be decremented by exactly one.} */
```

```
37093.   proc ESACOD_INFO_UNPACK
37094.           /* Extract the following information from the stack: */

37096.       CASE_COUNTER := OPERAND1(CURRENT_NEST_LEVEL)
37097.       INDEX_ADDR   := OPERAND2(CURRENT_NEST_LEVEL)
37098.       CASE_FORMAT  := OPERAND3(CURRENT_NEST_LEVEL)
37099.       MISC_FOUND   := INFORMATION(CURRENT_NEST_LEVEL)[2,1]
37100.       MULTIPLESOP4 := INFORMATION(CURRENT_NEST_LEVEL)[3,1]
37101.       INDEX_RANGE_ASSURED := INFORMATION(CURRENT_NEST_LEVEL)[5,1]
37102.       BLOCK_LABEL_PREFIX := '$' || BLOCK_NUMBER(CURRENT_NEST_LEVEL)
37103.     corp
```

```
37105.   proc ESACOD_GENERAL_CASE_CHOICE
37106.           /* Generate all code to complete processing of general case.
37107.              Includes the generation of a branch vector, if required. */

37109.       call ESACOD_GENERAL_CASE_INFO
37110.           /* Pops MAX_SD_VALUE, COMP_LABEL_NO,
37111.              and BASE (of CASE_OCCURS array) out of GCASE stack. */
37112.       if ¬ ERROR_OCCURRED
37113.         then
37114.           if ANY_SELFDEF_CASES
37115.             then
37116.               call ESACOD_SELFDEF_GEN
37117.                   /* Handles branch vector-type implementation for all cases which
37118.                      contain self-defining terms (of value < 256).  Also generates
37119.                      linkage for any other terms and/or MISC case which were used
37120.                      with the self-definers. */
37121.             else
37122.                   /* No self-definers were present. */
37123.               if ANY_COMP_CASES
37124.                 then
37125.                   call ESACOD_GENERAL_SYMB_ONLY
37126.                       /* Generate linkage to process symbolic operands and MISC in the
37127.                          absence of self-definers. */
37128.                 else
37129.                   mnote (8, 'STRC3701 DOCASE CONTAINS NO VALID CASES')
37130.                 fi
37131.             fi
37132.         fi
37133.     corp
```

```
37135.   proc ESACOD_GENERAL_SYMB_ONLY
37136.           /* Generate linkage to process symbolic operands and MISC in the
37137.               absence of self-definers (self-defined terms of value < 256). */

37140.       generate (BLOCK_LABEL_PREFIX || 'BEG   EQU    * ||
37141.               BLOCK_LABEL_PREFIX || 'C1')
37142.       if MISC_FOUND
37143.         then
37144.           generate (BLOCK_LABEL_PREFIX || 'C' || COMP_LABEL_NO || '  EQU    ' ||
37145.                   BLOCK_LABEL_PREFIX || 'MSC')
37146.         else
37147.           generate (BLOCK_LABEL_PREFIX || 'C' || COMP_LABEL_NO || '  DS    OH')
37148.         fi
37149.     corp
```

```
37151.   proc ESACOD_SELFDEF_GEN
37152.           /* Handles branch vector generation for processing cases defined by
37153.              self-defining terms (of value < 256).  Also generates linkage for
37154.              symbolic terms and/or MISC case following self definers. */

37156.        LABEL := BLOCK_LABEL_PREFIX || 'BEG'
37157.           /* Note that BEG label must be generated on first instruction. */
37158.        docase ifany
37159.          of
37160.            case ANY_COMP_CASES
37161.              NOCASE := BLOCK_LABEL_PREFIX || 'C1'
37162.            esac
37163.            case MISC_FOUND
37164.              NOCASE := BLOCK_LABEL_PREFIX || 'MSC'
37165.            esac
37166.            case misc
37167.              NOCASE := BLOCK_LABEL_PREFIX || 'END'
37168.            esac
37169.          esacod
37170.        RANGE_TEST_REQD := ((¬ INDEX_RANGE_ASSURED) or ANY_COMP_CASES)
37171.        if RANGE_TEST_REQD
37172.          then
37173.            call ESACOD_OUT_OF_RANGE_CHECK
37174.                 /* Generate check for index out of the range 0 through
37175.                    MAX_SD_VALUE. */
37176.          fi
37177.        call ESACOD_BRVCT_GEN
37178.           /* Generate branch vector and all final constants and equates
37179.              required. */
37180.     corp
```

```
37182.   proc ESACOD_GENERAL_CASE_INFO
37183.           /* Pops following information out of GCASE stack.  Indicates success
37184.               (or lack thereof) in ERROR_OCCURRED. */
37185.       int I

37187.       I                  := GCASE_NEST_LEVEL
37188.       GCASE_NEST_LEVEL   := I - 1
37189.       if I > GCASE_NEST_LIMIT
37190.         then
37191.           ERROR_OCCURRED    := true
37192.         else
37193.           MAX_SD_VALUE      := MAX_CASE_VALUE(I)
37194.           ANY_SELFDEF_CASES := (MAX_SD_VALUE ≥ 0)
37195.           COMP_LABEL_NO     := NEXT_COMP_LABEL_NO(I)
37196.           ANY_COMP_CASES    := (COMP_LABEL_NO > 1)
37197.           BASE              := ((I-1) * 256) + 1
37198.           ERROR_OCCURRED    := false
37199.         fi
37200.     corp
```

```
37202.   proc ESACOD_OUT_OF_RANGE_CHECK
37203.           /* Generate check for index out of the range 0 through MAX_SD_VALUE to
37204.              branch to the NOCASE label.  In addition, if all cases are multiples
37205.              of 4, branch if index is not. */

37207.       if INDEX_ADDR = ''
37208.         then /* Index is in GPR1. */
37209.           generate (LABEL || '  LTR   1,1')
37210.           if CASE_OCCURS(BASE)
37211.             then /* CASE 0 occurs. */
37212.               generate ('       BM    ' || NOCASE)
37213.             else
37214.               generate ('       BNP   ' || NOCASE)
37215.             fi
37216.           generate ('       C     1,' || BLOCK_LABEL_PREFIX || 'SIZ')
37217.           generate ('       BH    ' || NOCASE)
37218.           if MULTIPLESOF4
37219.             then
37220.               generate ('       LA    0,3')
37221.               generate ('       NR    0,1')
37222.               generate ('       BNZ   ' || NOCASE)
37223.             fi
37224.         else
37225.           generate (LABEL || '  CLI   ' || INDEX_ADDR || ',' || MAX_SD_VALUE)
37226.           generate ('       BH    ' || NOCASE)
37227.           if MULTIPLESOF4
37228.             then
37229.               generate ('       TM    ' || INDEX_ADDR || ",B'00000011'")
37230.               generate ('       BNZ   ' || NOCASE)
37231.             fi
37232.         fi
37233.       LABEL := ''
37234.     corp
```

```
37236.   proc BSACOD_BRVCT_GEN
37237.         /* Generate branch vector proper. */
37238.      int I, INCR

37240.      if INDEX_ADDR ≠ ''
37241.        then /* Generate code to put byte index into GPR1. */
37242.          generate (LABEL || '  SR    1,1')
37243.          LABEL := ''
37244.          generate ('       IC    1,' || INDEX_ADDR)
37245.        fi
37246.      if MULTIPLESOF4
37247.        then
37248.          INCR := 4
37249.        else
37250.          INCR := 1
37251.          generate (LABEL || '  SLA   1,2')
37252.          LABEL := ''
37253.        fi
37254.      if CASE_OCCURS(BASE) or INDEX_ADDR ≠ ''
37255.        then /* Zero case must be included in branch vector. */
37256.          generate (LABEL || '  B     **4(1)')
37257.          I := 0
37258.        else
37259.          generate (LABEL || '  B     *(1)')
37260.          I := INCR
37261.            /* Skip the zero case and start with case 1 (or case 4). */
37262.        fi
37263.      LABEL := ''
37264.      while I ≤ MAX_SD_VALUE
37265.        do
37266.          if CASE_OCCURS(BASE+I)
37267.            then
37268.              generate ('       B     ' || BLOCK_LABEL_PREFIX || 'G' || I)
37269.            else
37270.              generate ('       B     ' || NOCASE)
37271.            fi
37272.          I := I + INCR
37273.        od /* {Termination:  INCR > 0, so I is incremented in loop;
37274.                  MAX_SD_VALUE is fixed;  therefore I must eventually exceed
37275.                  MAX_SD_VALUE.} */
37276.      if RANGE_TEST_REQD and INDEX_ADDR = ''
37277.        then
37278.          generate (BLOCK_LABEL_PREFIX || "SIZ  DC    F'" || MAX_SD_VALUE ||
37279.                "'")
37280.        fi
37281.      if ANY_COMP_CASES
37282.        then
37283.          if MISC_FOUND
37284.            then
37285.              generate (BLOCK_LABEL_PREFIX || 'C' || COMP_LABEL_NO || '  EQU   ' ||
37286.                    BLOCK_LABEL_PREFIX || 'MSC')
37287.            else
37288.              generate (BLOCK_LABEL_PREFIX || 'C' || COMP_LABEL_NO || '  DS    0H')
37289.            fi
37290.        fi
37291.    corp
```

"BLOCK" Macro -- 15 June 1973

```
41001.   macro  BLOCK  (USER_NAME; )
41002.           /* Generate simple one-in-one-out block in structure with name
41003.               specified. */
41004.       char LABEL
41005.           /* Contains any outstanding label waiting to be generated. */

41007.       call TRACE_PRINTER (  ;  'BLOCK')
41008.           /* Prints macro name "BLOCK" in mnote if tracing on. */
41009.       call PUSH_NEW_BLOCK(USER_NAME;  BLOCK_TYPE_VALUE='BLOCK')
41010.           /* Define new block;  add to stack.  Initialize block specifications.
41011.               Note block type and set up a unique BLOCK_LABEL_PREFIX for use in
41012.               generating labels. */
41013.       if ERROR_OCCURRED
41014.         then
41015.           mexit
41016.         fi
41017.       LABEL := USER_NAME
41018.       call BLOCK_TRACE_COUNTERS
41019.           /* If block counts were requested, generate counters and incrementing
41020.               instructions.  Any label waiting to be defined is returned in
41021.               LABEL. */
41022.       if LABEL ^= ''
41023.         then
41024.           generate (LABEL || '  DS    0H')
41025.               /* Define label if one required and not yet defined. */
41026.         fi
41027.    mend
```

```
41029.   proc BLOCK_TRACE_COUNTERS
41030.           /* Generate debugging information required---block name constant
41031.              and/or block counters. */

41033.       if DEBUG_BLOCKCOUNTS_REQD or DEBUG_BLOCKNAMES_REQD
41034.         then
41035.           if DEBUG_BLOCKCOUNTS_REQD
41036.             then
41037.                   /* Generate block count incrementing instructions. */
41038.               generate (LABEL || '  LH    1,' || BLOCK_LABEL_PREFIX || 'BLC')
41039.               LABEL := ''  /* Clear LABEL to show it has been generated. */
41040.               generate ('       LA    1,1(1)')
41041.               generate ('       STH   1,' || BLOCK_LABEL_PREFIX || 'BLC')
41042.             fi
41043.           generate (LABEL || '  B     ' || BLOCK_LABEL_PREFIX || 'GO')
41044.               /* Generate branch around block name and/or count. */
41045.           LABEL := BLOCK_LABEL_PREFIX || 'GO'
41046.               /* Establish GO label as requiring definition. */
41047.           if DEBUG_BLOCKNAMES_REQD
41048.             then
41049.               generate ("       DC    C'" || BLOCK_NAME(CURRENT_NEST_LEVEL) ||
41050.                         "',0H'0'")
41051.             fi
41052.           if DEBUG_BLOCKCOUNTS_REQD
41053.             then
41054.               generate (BLOCK_LABEL_PREFIX || "BLC  DC    H'0' BLOCK COUNT")
41055.             fi
41056.         fi
41057.     corp
```

```
43001.   macro  BLEND  (  ;  USER_NAME, RETURN=, LINK=14, RESTORE=, RC=)
43002.            /* The BLEND macro acts as a generic name for FI, OD, ESAC, ESACOD, and
43003.                PROC and as a terminating macro for BLOCK.  For any
43004.                of the former, the proper macro is invoked depending on the
43005.                block type being terminated.  For BLOCK blocks, the block is
43006.                simply terminated. */
43007.         int I  /* Temporary. */

43009.         call TRACE_PRINTER (  ;  'BLEND')
43010.             /* Prints macro name "BLEND" in mnote if tracing on. */
43011.         if CURRENT_NEST_LEVEL > NESTING_LIMIT
43012.           then
43013.             call POP_OLD_BLOCK (  ;  )
43014.           else
43015.             if CURRENT_NEST_LEVEL = 0
43016.               then
43017.                 mnote (8, 'STRC4301 NO BLOCKS ACTIVE--"BLEND" IGNORED')
43018.               else
43019.                 I := CURRENT_NEST_LEVEL
43020.                 if USER_NAME ¬= ''
43021.                   then
43022.                     while I > 0 and BLOCK_NAME(I) ¬= USER_NAME
43023.                       do
43024.                         I := I - 1
43025.                       od /* {Termination:  I is decremented--must eventually become
43026.                                 ≤ zero.} */
43027.                     if I = 0
43028.                       then
43029.                         mnote (8, 'STRC4302 NO BLOCK ACTIVE NAMED ' || USER_NAME ||
43030.                                   '--"BLEND" IGNORED')
43031.                         mexit
43032.                       fi
43033.                   fi
43034.                 docase BLOCK_TYPE(I) only
43035.                   of
43036.                     case 'IF'
43037.                       call FI (  ;  USER_NAME)
43038.                     esac
43039.                     case 'DO'
43040.                       call OD (  ;  USER_NAME)
43041.                     esac
43042.                     case 'CASE'
43043.                       call ESAC (  ; USER_NAME)
43044.                     esac
43045.                     case 'DOCASE'
43046.                       call ESACOD (  ; USER_NAME)
43047.                     esac
43048.                     case 'PROC'
43049.                       call CORP (  ; USER_NAME, RETURN=RETURN, LINK=LINK, RESTORE=RESTORE,
43050.                                  RC=RC)
43051.                     esac
43052.                     case 'BLOCK'
43053.                       call POP_OLD_BLOCK (  ;  )
43054.                     esac
43055.                   esacod
43056.               fi
43057.           fi
43058.     mend
43059.       /* {Lemma:  If CURRENT_NEST_LEVEL > 0 and
43060.              [ USER_NAME = '' or = BLOCK_NAME(CURRENT_NEST_LEVEL) ] at entry to
43061.              BLEND, then CURRENT_NEST_LEVEL will be decremented by exactly
43062.              one.} */
```

```
53001.   macro  FINAL  (  ;  )
53002.            /* Insure all blocks are closed.  Then if SAVETRACE_ON_FIRST_PROC,
53003.               define label $LASTSAV to be PREV_SAVETRACE_AREA and EQUate
53004.               PREV_SAVETRACE_PTR to 0. */

53006.       call TRACE_PRINTER (  ; 'FINAL')
53007.            /* Print macro name "FINAL" in mnote if tracing on. */
53008.       while CURRENT_NEST_LEVEL > 0
53009.         do
53010.           if CURRENT_NEST_LEVEL > NESTING_LIMIT
53011.             then
53012.               mnote (8, 'STRC5301 BLEND OF OUTSTANDING BLOCK ASSUMED')
53013.             else
53014.               mnote (8, 'STRC5301 BLEND OF ' || BLOCK_NAME(CURRENT_NEST_LEVEL) ||
53015.                         ' ASSUMED')
53016.             fi
53017.           call BLEND (  ;  )
53018.               /* {Lemma:  If CURRENT_NEST_LEVEL > 0 and no BLEND operands are
53019.                           specified, CURRENT_NEST_LEVEL will be decremented by exactly
53020.                           one.} */
53021.         od /* {Termination:  CURRENT_NEST_LEVEL decreases monotonically
53022.                   and therefore must eventually become ≤ 0.} */
53023.       if SAVETRACE_ON_FIRST_PROC
53024.         then
53025.           if PREV_SAVETRACE_PTR = '$FIRSTSV'
53026.             then /* No non-OS procs occurred;  generate dummy area. */
53027.               generate ("$FIRSTSV DC    OF'0',X'FFFFFFFF',A(0,0) DUMMY SAVEAREA")
53028.               generate ('$LASTSAV EQU   $FIRSTSV')
53029.             else
53030.               generate ('$LASTSAV  EQU   ' || PREV_SAVETRACE_AREA)
53031.               generate (PREV_SAVETRACE_PTR || '  EQU   0')
53032.             fi
53033.         fi
53034.     mend
```

"EXIT" Macro -- 10 July 1973

```
55001.   macro  EXIT  (USER_NAME;  EXIT_TARGET)
55002.           /* Find exit point.  Generate branch. */

55004.       call TRACE_PRINTER ( ;  'EXIT')
55005.            /* Print macro name "EXIT" in mnote if tracing on. */
55006.       if CURRENT_NEST_LEVEL > NESTING_LIMIT
55007.         then
55008.           mexit
55009.         fi
55010.       call EXIT_FIND (  ;  EXIT_TARGET)
55011.            /* Set ULTIMATE_BRANCH_LABEL to point to end of block whose name
55012.                is the argument and ,arl as meedomg an XIT label;  if no such block,
55013.                issue message and set ERROR_OCCURRED. */
55014.       if ¬ ERROR_OCCURRED
55015.         then
55016.           generate (USER_NAME || ' B      ' || ULTIMATE_BRANCH_LABEL)
55017.         fi
55018.     mend
```

```
81001.  macro  PROC  (USER_NAME;  LINKAGE=, ID=, BASE=, WORK=, SAVE=,
81002.                DEBUG=, EXIT=)
81003.         /* Defines a procedure block.  If LINKAGE=OS is specified, standard
81004.            OS save area conventions are followed;  otherwise a simple non-
81005.            linked save area is provided.  A base register is established
81006.            (unless BASE=NONE is specified under OS_LINKAGE or BASE= is
81007.            omitted on local PROCs).  Register values upon entry are
81008.            saved to allow restoring at CORP time. */
81009.      bit
81010.         FIRST_PROC,  /* Indicates whether this is the first PROC macro coded
81011.                         in this assembly. */
81012.         FIRST_VALUE_KNOWN,  /* Indicates whether the first SAVE= operand was
81013.                         a self-defining term (or omitted) or if it was symbolic. */
81014.         OS_LINKAGE,  /* Indicates whether LINKAGE=(OS,***) was
81015.                         entered. */
81016.         SPECIAL_PREFIX, /* Indicates whether the BLOCK_LABEL_PREFIX was
81017.                         changed to the special debugging form "$Pnn". */
81018.         USING13,  /* Indicates whether the base register is GPR13. */
81019.         MULTIBASE,  /* Indicates more than one base register was
81020.                         requested, but adcons for loading have not yet been generated. */
81021.         WORKREG_USED   /* Indicates whether the value in WORKREG was
81022.                         modified and its contents saved in register 0. */
81023.      char
81024.         COMMA2, MULT,  /* Contain "," and "M" respectively if a range of
81025.                         registers is to be saved, or the null string if a single register
81026.                         to be saved.  Used to generate "STM" or "ST" instruction. */
81027.         FIRST, LAST,  /* First and last register in range to be saved. */
81028.         LABEL,   /* Any outstanding label waiting to be generated. */
81029.         LOCAL_POINTER, OS_POINTER,  /* Instruction segments to generate
81030.                         store instruction for proper save area. */
81031.         PREVIOUS_DEBUG_VECTOR,  /* Holds value of debug switches on entry to
81032.                         PROC macro for restoring on exit from CORP. */
81033.         PROC_ID_BYTE,  /* Value of hex proc number (PROC_COUNTER in hex)
81034.                         used in various debugging instructions.  */
81035.         SAVE_LENGTH, /* Length of save area (in words), except length of
81036.                         register part only for local PROCs. */
81037.         SAVE_TYPE,  /* Type of save area generated:  FULL (savetrace), OSSAVE,
81038.                         NORML, NORMLHDR, TRUNC, TRUNCHDR, or NONE. */
81039.         SAVEREG,  /* Register (work or base) which is pointing at new
81040.                         save areabefore chaining. */
81041.         WORKREG   /* Register used for setting up linkage, etc. */
81042.      int
81043.         OFFSET,  /* Offset (in words) to either FIRST (if FIRST_VALUE_KNOWN),
81044.                         or to GPR0 within save area. */
81045.         OFFSET_TO_GPR0,  /* Offset in words to GPR0 within save area. */
81046.         SAF, SAL  /* Register number to go intc first register word of
81047.                         save area;  this, for example, could be 14 even though FIRST
81048.                         is a symbolic register of unknown value at macro expansion time.
81049.                         SAL is similar but for last register. */
```

```
        call TRACE_PRINTER ( ; 'PROC')
            /* Print macro name "PROC" in mnote if tracing on. */
        call PUSH_NEW_BLOCK(USER_NAME; BLOCK_TYPE_VALUE='PROC')
            /* Define new block; add to stack. Initialize block specifications.
                Note block type and set up unique BLOCK_LABEL_PREFIX for use
                in generating unique labels. */
        if ERROR_OCCURRED /* durring PUSH_NEW_BLOCK (viz., stack overflow) */
          then
            mexit
          fi
        LABEL := USER_NAME
            /* Generate PROC's name at first opportunity. */
        call PROC_SCAN_OPTIONS
            /* Validate LINKAGE= and WORK= keywords; issue error messages and set
                OS_LINKAGE and WORKREG. Process completely DEBUG and EXIT keywords.
                Change BLOCK_LABEL_PREFIX if necessary to special PROC form
                (indicating change in SPECIAL_PREFIX) and set value of FIRST_PROC. */
        call PROC_HEADER
            /* Generate "CSECT" and "USING *,15" if required. Handle in-line
                ID (a la IBM SAVE macro). */
        call PROC_REG_SAVE
            /* Set SAVE_TYPE and SAVE_LENGTH to indicate type of save area
                required. Save contents of general purpose registers, if required. */
        call PROC_ESTABLISH_BASE
            /* Set up base register where required and issue USING. Set USING13 if
                base register to be loaded into 13 was put temporarily into
                WORKREG. If multiple base registers, set MULTIBASE. */
        call PROC_GEN_SAVEAREA
            /* Generate proper save area depending on the variables SAVE_TYPE and
                SAVE_LENGTH set by PROC_REG_SAVE and depending on the SAVE macro
                operands. */
        call PROC_MULTIBASE_GEN
            /* Generate definition of adcons for multiple base registers. */
        call PROC_DEBUG_STUFF
            /* Generate trace and count code for debugging, if requested. */
        if WORK = 'NONE' and WORKREG_USED
          then
            generate (LABEL || '  LR     ' || WORKREG || ',0')
                /* Restore WORKREG. */
            LABEL := ''
          fi
        if LABEL ≠ ''
          then
            generate (LABEL || '  DS    0H')
          fi
        if LINKAGE(2) = 'CSECT'
          then
            generate ('        DROP  15')
                /* DROP for USING generated by PROC_HEADER. */
          fi
        call PROC_INFO_SAVE
            /* Save any information necessary to generate CORP macro. */
      mend
```

```
81105.   proc PROC_SCAN_OPTIONS
81106.           /* Validate LINKAGE= and WORK= keywords;  issue error messages and set
81107.              OS_LINKAGE and WORKREG (the latter receiving either the register
81108.              specified by the WORK= operand or some default).  Process completely
81109.              the DEBUG= and EXIT= keywords.  Change BLOCK_LABEL_PREFIX if
81110.              necessary from the normal "$nnn" to the special PROC prefix
81111.              "$Ppp".  Set FIRST_PROC to indicate whether this is the first
81112.              PROC macro coded in this assembly. */

81114.       OS_LINKAGE := (LINKAGE(1) ≠ '')
81115.       if LINKAGE(1) ≠ 'OS' and ≠ ''
81116.         then
81117.           mnote (8, 'STRC8101 LINKAGE=' || LINKAGE(1) ||
81118.                     ' INVALID--"OS" ASSUMED')
81119.         fi
81120.       if LINKAGE(2) ≠ 'CSECT' and ≠ ''
81121.         then
81122.           mnote (8, 'STRC8102 SECOND LINKAGE OPERAND IGNORED')
81123.         fi
81124.       WORKREG := WORK
81125.       if WORKREG = 'NONE' or = ''
81126.         then
81127.              /* Pick default WORKREG.  We will restore it later if WORK=NONE and
81128.                 it gets clobbered. */
81129.           if OS_LINKAGE
81130.             then
81131.               WORKREG := '2'
81132.             else
81133.               WORKREG := '1'
81134.             fi
81135.         fi
81136.       EXIT_SEVERITY := EXIT
81137.           /* Save specified severity for EXIT target mnotes. */
81138.       FIRST_PROC := ¬ NOT_FIRST_PROC /* Brilliant, eh? */
81139.       NOT_FIRST_PROC := true
81140.           /* Note whether this is the first proc and set global NOT_FIRST_PROC
81141.              so we will be able to tell on later PROCs.  Note that we are making
81142.              use here of the fact that global bit variables are initialized to
81143.              false. */
81144.       call PROC_DEBUG_SET
81145.           /* Set debugging switches according to DEBUG= operand, saving prior
81146.              values for restoring at CORP time in PREVIOUS_DEBUG_VECTOR. */
81147.       SPECIAL_PREFIX := (DEBUG_SAVETRACE_REQD or
81148.              DEBUG_PROCTRACE_REQD or
81149.              DEBUG_PROCCOUNTS_REQD)  and
81150.                  PROC_COUNTER < 254
81151.       if SPECIAL_PREFIX
81152.         then /* We want to label this proc with a hex proc number. */
81153.           PROC_COUNTER := PROC_COUNTER + 1
81154.               /* Advance counter only on those procs which use it. */
81155.           HEX_IN := PROC_COUNTER
81156.           call XHEX ( ; )
81157.           PROC_ID_BYTE := HEX_OUT
81158.           BLOCK_LABEL_PREFIX := '$P' || PROC_ID_BYTE
81159.               /* Change labels on PROC to ease understanding of debug information. */
81160.           mnote (*, 'STRC8108 PROC ' || BLOCK_NAME(CURRENT_NEST_LEVEL) ||
81161.                     ", DEBUG ID=X'" || PROC_ID_BYTE || "'")
81162.         else
81163.           PROC_ID_BYTE := '00'
81164.         fi
81165.     corp
```

```
81167.   proc PROC_DEBUG_SET
81168.           /* Save the previous DEBUG specifications so they can be restored at
81169.              CORP time.  Scan the DEBUG= suboperands setting the debug flags
81170.              indicated.  If GLOBAL is specified send null restore value to
81171.              suppress it. */
81172.       bit
81173.           GLOBAL,  /* Indicates whether "GLOBAL" has been found as an
81174.                       operand of DEBUG=. */
81175.           SAVETRACE_VALUE,  /* Set true if SAVETRACE is to be turned on;  set
81176.                       false if SAVETRACE is to be turned off;  else not set. */
81177.           SAVETRACE_CHECK  /* Set true if SAVETRACE to be set either on or
81178.                       off. */
81179.       int
81180.           I  /* List suboperand index. */

81182.       GLOBAL := false  /* Has not yet been found. */
81183.       PREVIOUS_DEBUG_VECTOR :=
81184.               DEBUG_BLOCKNAMES_REQD  ||
81185.               DEBUG_PROCNAMES_REQD   ||
81186.               DEBUG_LISTBLOCKS_REQD  ||
81187.               DEBUG_BLOCKCOUNTS_REQD ||
81188.               DEBUG_PROCCOUNTS_REQD  ||
81189.               DEBUG_PROCTRACE_REQD   ||
81190.               DEBUG_CORPVALUES_REQD  ||
81191.               DEBUG_SAVETRACE_REQD
81192.           /* Save current value of debug switches. */
81193.       I := 1
81194.       while I ≤ N'DEBUG
81195.         do /* Scan all operands . */
81196.           docase DEBUG(I)
81197.             of
81198.               case ('GLOBAL', 'GBL')
81199.                 GLOBAL := true
81200.               esac
81201.               case ('BLOCKNAMES', 'BN')
81202.                 DEBUG_BLOCKNAMES_REQD := true
81203.                     /* BLOCKNAMES causes the name of each block to be generated as an
81204.                        in-line character constant at the start of each block (of any type,
81205.                        not just BLOCK macros) for ease of locating code in dumps. */
81206.               esac
81207.               case ('NOBLOCKNAMES', 'NBN')
81208.                 DEBUG_BLOCKNAMES_REQD := false
81209.               esac
81210.               case ('PROCNAMES', 'PN')
81211.                 DEBUG_PROCNAMES_REQD := true
81212.                     /* PROCNAMES causes the name of each PROC to be generated as an
81213.                        in-line character constant at the start of the PROC for ease of
81214.                        locating code in dumps. */
81215.               esac
81216.               case ('NOPROCNAMES', 'NPN')
81217.                 DEBUG_PROCNAMES_REQD := false
81218.               esac
81219.               case ('LISTBLOCKS', 'LB')
81220.                 DEBUG_LISTBLOCKS_REQD := true
81221.                     /* LISTBLOCKS causes the name, number, and depth of each block to be
81222.                        generated in an mnote at the start and end of the block. */
81223.               esac
81224.               case ('NOLISTBLOCKS', 'NLB')
81225.                 DEBUG_LISTBLOCKS_REQD := false
81226.               esac
81227.               case ('BLOCKCOUNTS', 'BC')
81228.                 DEBUG_BLOCKCOUNTS_REQD := true
81229.                     /* BLOCKCOUNTS causes counters to be kept on the number of executions
81230.                        of all blocks. */
81231.               esac
81232.               case ('NOBLOCKCOUNTS', 'NBC')
81233.                 DEBUG_BLOCKCOUNTS_REQD := false
81234.               esac
81235.               case ('PROCCOUNTS', 'PC')
81236.                 DEBUG_PROCCOUNTS_REQD := true
81237.                     /* PROCCOUNTS causes counters to be kept on the number of executions
81238.                        of all PROC blocks. */
81239.               esac
81240.               case ('NOPROCCOUNTS', 'NPC')
81241.                 DEBUG_PROCCOUNTS_REQD := false
81242.               esac
```

```
81244.              case ('PROCTRACE', 'PT')
81245.                DEBUG_PROCTRACE_REQD := true
81246.                    /* PROCTRACE causes a trace vector to be generated and instructions to
81247.                       move the hex PROC number into the vector to show the order of PROC
81248.                       calls. */
81249.              esac
81250.              case ('NOPROCTRACE', 'NPT')
81251.                DEBUG_PROCTRACE_REQD := false
81252.              esac
81253.              case ('CORPVALUES', 'CV')
81254.                DEBUG_CORPVALUES_REQD := true
81255.                    /* CORPVALUES causes the value of the registers at CORP time (before
81256.                       restoring those saved at PROC entry) to be stored into an area for
81257.                       reference. */
81258.              esac
81259.              case ('NOCORPVALUES', 'NCV')
81260.                DEBUG_CORPVALUES_REQD := false
81261.              esac
81262.              case ('SAVETRACE', 'ST')
81263.                    /* SAVETRACE causes all save areas for non-OS register saving to be
81264.                       linked into a static chain from the OS save area to provide
81265.                       formatting in the save area trace portion of ABEND/SNAP dumps. */
81266.                SAVETRACE_VALUE := true
81267.                SAVETRACE_CHECK := true
81268.                    /* Note that savetrace has been specified. */
81269.              esac
81270.              case ('NOSAVETRACE', 'NST')
81271.                SAVETRACE_VALUE := false
81272.                SAVETRACE_CHECK := true
81273.              esac
81274.              case ('ALL', 'NONE')
81275.                DEBUG_BLOCKNAMES_REQD, DEBUG_PROCNAMES_REQD, DEBUG_LISTBLOCKS_REQD,
81276.                    DEBUG_BLOCKCOUNTS_REQD, DEBUG_PROCCOUNTS_REQD, DEBUG_PROCTRACE_REQD,
81277.                    DEBUG_CORPVALUES_REQD, SAVETRACE_VALUE
81278.                        := (DEBUG(I) = 'ALL')
81279.                    /* Set (or reset) all main debug switches. */
81280.                SAVETRACE_CHECK := true
81281.              esac

81283.                  /* The following operands are provided for the debugging of the
81284.                      structured macros themselves.  They are not automatically restored
81285.                      at CORP time. */
81286.              case ('MACRONAMES', 'MN')
81287.                DEBUG_MACRONAMES_REQD := true
81288.                    /* MACRONAMES causes the name of each structured macro (including inner
81289.                       macros) to be printed in an mnote whenever evoked. */
81290.              esac
81291.              case ('NOMACRONAMES', 'NMN')
81292.                DEBUG_MACRONAMES_REQD := false
81293.              esac
81294.              case ('DEBUGMACROS', 'DM')
81295.                DEBUG_DEBUGMACROS_REQD := true
81296.                    /* DEBUGMACROS causes various intermediate values within the macros
81297.                       to be printed in mnotes for use in debugging the macros. */
81298.              esac
81299.              case ('NODEBUGMACROS', 'NDM')
81300.                DEBUG_DEBUGMACROS_REQD := false
81301.              esac
81302.              case misc
81303.                mnote (8, 'STRC8104 DEBUG=' || DEBUG(I) || ' INVALID--IGNORED')
81304.              esac
81305.            esacod
81306.          I := I + 1 /* Go on to next suboperand. */
81307.        od /* {Termination:  I is incremented, N'DEBUG is fixed in loop;
81308.                  I must eventually exceed N'DEBUG.} */
```

```
81310.       if SAVETRACE_CHECK
81311.         then
81312.           if SAVETRACE_VALUE
81313.             then
81314.               if FIRST_PROC
81315.                 then
81316.                   if OS_LINKAGE
81317.                     then
81318.                       SAVETRACE_ON_FIRST_PROC := true
81319.                       DEBUG_SAVETRACE_REQD := true
81320.                       mnote (4, 'STRC8103 WARNING--SAVETRACE REQUIRES "FINAL" MACRO')
81321.                     else
81322.                       mnote (8, 'STRC8106 SAVETRACE REQUIRES FIRST PROC TO BE LINKAGE=OS')
81323.                     fi
81324.                 else /* Not first PROC. */
81325.                   if SAVETRACE_ON_FIRST_PROC
81326.                     then /* SAVETRACE is being resumed. */
81327.                       DEBUG_SAVETRACE_REQD := true
81328.                     else
81329.                       mnote (8, 'STRC8105 SAVETRACE MUST BE SPECIFIED ON FIRST PROC')
81330.                     fi
81331.                 fi
81332.             else
81333.               DEBUG_SAVETRACE_REQD := false
81334.             fi
81335.         fi
81336.       if GLOBAL
81337.         then
81338.           PREVIOUS_DEBUG_VECTOR := ''
81339.             /* Null value suppresses restore by CORP. */
81340.         fi
81341.     corp
```

```
81343.   proc PROC_HEADER
81344.          /* Generates a "CSECT" and "USING *,15" if LINKAGE=(***,CSECT)
81345.              specified.  If LINKAGE=(OS,***) specified, generate inline ID
81346.              similar to IBM SAVE macro;  for non-OS linkage, do the same if
81347.              DEBUG_PROCNAMES_REQD or ID= specified, but omit ID-length-count field.
81348.              Return any label generated as branch target in LABEL.  */
81349.        char
81350.            SECT, /* PROC name (or "$PRIVATE") for default ID constant. */
81351.            QUOTE, /* "'" or null, for generating id. */
81352.            TARGET   /* Temporary. */
81353.        int
81354.            LENGTH  /* Length of character identifier. */

81356.        if LINKAGE(2) = 'CSECT'
81357.          then
81358.            generate (LABEL || '  CSECT')
81359.            LABEL := ''
81360.            generate ('        USING *,15')
81361.          fi
81362.        if ((OS_LINKAGE or ID ≠ '') and ID ≠ 'NONE') or
81363.                DEBUG_PROCNAMES_REQD
81364.          then
81365.            TARGET := BLOCK_LABEL_PREFIX || 'AA'
81366.            generate (LABEL || '  B      ' || TARGET)
81367.                /* Branch around in-line ID. */
81368.            LABEL := TARGET
81369.            if ID = '' or = '*' or = 'NONE'
81370.              then /* No ID was specified on PROC macro. */
81371.                if USER_NAME = ''
81372.                  then
81373.                    if OS_LINKAGE
81374.                      then
81375.                        SECT := '$PRIVATE' /* Default name if none specified. */
81376.                      else
81377.                        SECT := BLOCK_NAME(CURRENT_NEST_LEVEL)
81378.                    fi
81379.                    LENGTH := 8
81380.                  else
81381.                    SECT := USER_NAME
81382.                    LENGTH := K'USER_NAME
81383.                  fi
81384.                if OS_LINKAGE
81385.                  then /* Generate SAVE-type ID. */
81386.                    LENGTH := ((LENGTH/2)*2) + 1 /* Round up to odd number. */
81387.                    generate ('        DC    AL1(' || LENGTH || '),CL' || LENGTH ||
81388                            "'" || SECT || "'")
81389.                  else /* Generate PROC-name constant only. */
81390.                    LENGTH := ((LENGTH + 1)/2)*2 /* Round up to even number. */
81391.                    generate ('        DC    CL' || LENGTH || "'" || SECT || "'")
81392.                  fi
81393.              else /* ID= specified. */
81394.                QUOTE := "'"
81395.                LENGTH := 0
81396.                if ID[1,1] = "'"
81397.                  then /* ID already contains surrounding quotes. */
81398.                    QUOTE := ''
81399.                    LENGTH := -2  /* Subtract 2 for the quotes. */
81400.                  fi
81401.                if OS_LINKAGE
81402.                  then
81403.                    LENGTH := ((K'ID/2)*2) + 1 + LENGTH
81404.                        /* Round up to odd number. */
81405.                    generate ('        DC    AL1(' || LENGTH || '),CL' || LENGTH ||
81406.                            QUOTE || ID || QUOTE)
81407.                  else
81408.                    LENGTH := ((K'ID + 1)/2)*2 + LENGTH
81409.                        /* Round up to even number. */
81410.                    generate ('        DC    CL' || LENGTH || QUOTE || ID || QUOTE)
81411.                  fi
81412.              fi
81413.          fi
81414.    corp
```

```
81416.    proc PROC_REG_SAVE
81417.           /* If SAVE=NONE and doing SAVETRACE generate store of all registers.
81418.               If SAVE=NONE and no SAVETRACE, do nothing.  For other than
81419.               SAVE=NONE, extract from SAVE= operands register list to be saved.
81420.               Decide type and size of save area and put into SAVE_TYPE and
81421.               SAVE_LENGTH.  Generate instruction to store registers.  */
81422.       int I  /* Temporary. */

81424.       FIRST, LAST := ''
81425.       OFFSET, SAF, SAL, OFFSET_TO_GPRO := 0
81426.           /* Initialize save request information variables. */
81427.       if SAVE = 'NONE'
81428.         then
81429.           if DEBUG_SAVETRACE_REQD
81430.             then
81431.               SAVE_TYPE := 'FULL'
81432.               SAVE_LENGTH := 15
81433.               generate (LABEL || '  STM   14,12,' || BLOCK_LABEL_PREFIX || 'SV+12')
81434.               LABEL := ''
81435.           fi
81436.         else
81437.           call PROC_SET_SAVE_INFO
81438.               /* Collect following save request information.  Put character string
81439.                   name of first register to be saved in FIRST, last in LAST, 'M' into
81440.                   MULT, and ',' into COMMA2;  if only a single register to be saved,
81441.                   LAST, MULT, and COMMA2 get null strings.  Register number which could
81442.                   go into first word of register part of save area goes into SAF, last
81443.                   into SAL.  (These, for example, could be 14 and 12 while FIRST and
81444.                   LAST are symbolic register designations of unknown value at macro
81445.                   expansion time.)  Set FIRST_VALUE_KNOWN if FIRST is not symbolic. */
81446.           call PROC_DECIDE_SAVE_TYPE
81447.               /* Put type of save area to be generated into SAVE_TYPE.  Set
81448.                   SAVE_LENGTH with length (in words) of save area (except only length
81449.                   of register part for non-OS_LINKAGE areas).  Offset in save area
81450.                   (in words) of register 0 is put into OFFSET_TO_GPRO.  Offset of
81451.                   either FIRST (if FIRST_VALUE_KNOWN) or else of register 0 is
81452.                   put into OFFSET.  Also set LOCAL_POINTER to '' and OS_POINTER to
81453.                   '(13)' if OS_LINKAGE, else LOCAL_POINTER to
81454.                   (BLOCK_LABEL_PREFIX || 'SV+') and OS_POINTER to ''.  Thus
81455.                   LOCAL_POINTER || OFFSET || '*4' || OS_POINTER refers to the given
81456.                   offset in the proper area. */
81457.           if SAVE_TYPE = 'FULL'
81458.             then
81459.               generate (LABEL || '  STM   14,12,' || BLOCK_LABEL_PREFIX || 'SV+12')
81460.                   /* Save all registers on FULL (savetrace-required) save area. */
81461.             else
81462.               if FIRST_VALUE_KNOWN
81463.                 then
81464.                   I := OFFSET * 4  /* Calculate offset in bytes. */
81465.                   generate (LABEL || '  ST' || MULT || '   ' || FIRST || ',' ||
81466.                           LAST || COMMA2 || LOCAL_POINTER || I || OS_POINTER)
81467.                 else
81468.                   generate (LABEL || '  ST' || MULT || '   ' || FIRST || ',' ||
81469.                           LAST || COMMA2 || LOCAL_POINTER || '(' || FIRST || '+' || OFFSET ||
81470.                           '-((' || FIRST || '+2)/16*16))*4' || OS_POINTER)
81471.                 fi
81472.               fi
81473.             LABEL := ''
81474.         fi
81475.     corp
```

```
81477.   PROC PROC_SET_SAVE_INFO
81478.         /* Collect following save request information.  Put character string
81479.             name of first register to be saved in FIRST, last in LAST, 'M' into
81480.             MULT, and ',' into COMMA2;  if only a single register to be saved,
81481.             LAST, MULT, and COMMA2 get null strings.  Register number which could
81482.             go into first word of register part of save area goes into SAF, last
81483.             into SAL.  (These, for example, could be 14 and 12 while FIRST and
81484.             LAST are symbolic register designations of unknown value at macro
81485.             expansion time.)  Set FIRST_VALUE_KNOWN if FIRST is not symbolic. */

81487.       FIRST_VALUE_KNOWN := true
81488.       MULT := 'M'
81489.       COMMA2 := ','
81490.          /* Assumed values.  */
81491.       if T'SAVE(1) = '0' /* At least first suboperand is omitted. */
81492.         then
81493.           FIRST := '14'  SAF := 14
81494.           LAST  := '12'  SAL := 12
81495.              /* Default is to save all registers 14 through 12. */
81496.         else
81497.           if T'SAVE(1) = 'N' /* Self-defining term. */
81498.             then
81499.               SAF := SAVE(1) /* Store it as a number. */
81500.               FIRST := SAF
81501.                   /* Convert it back to a string (done for non-decimal
81502.                       self-defining terms). */
81503.             else /* It must be symbolic. */
81504.               FIRST := SAVE(1) /* Store it as a character string. */
81505.               SAF := 14        /* Just say first of save area is register 14. */
81506.               FIRST_VALUE_KNOWN := false
81507.             fi
81508.           if T'SAVE(2) = '0' /* Second suboperand is omitted. */
81509.             then
81510.               LAST, MULT, COMMA2 := ''
81511.               if FIRST_VALUE_KNOWN
81512.                 then
81513.                   SAL := SAF     /* Last register is same as first. */
81514.                 else
81515.                   SAL := 12      /* Last register is 12. */
81516.                 fi
81517.             else
81518.               if T'SAVE(2) = 'N' /* Self-defined. */
81519.                 then
81520.                   SAL := SAVE(2) /* Store it as a number. */
81521.                   LAST := SAL    /* Convert it back to a string. */
81522.                 else
81523.                   LAST := SAVE(2) /* Store it as a character string. */
81524.                   SAL := 12       /* Just say last of save area is register 12. */
81525.                 fi
81526.             fi
81527.         fi
81528.     corp
```

```
proc PROC_DECIDE_SAVE_TYPE
            /* Set SAVE_TYPE with type of save area to be generated:  NONE, OSSAVE,
               TRUNC, TRUNCHDR, NORML, NORMLHDR, or FULL.  Set SAVE_LENGTH with
               length (in words) of save area (except only length of register part
               for non-OS_LINKAGE areas).  Offset in save area (in words) of either
               FIRST (if FIRST_VALUE_KNOWN) or else of register 0 is put into OFFSET;
               the latter always is stored into OFFSET_TO_GPR0 for CORP's reference.
               Also set LOCAL_POINTER to '' and OS_POINTER to '(13)' if OS_LINKAGE,
               else LOCAL_POINTER to (BLOCK_LABEL_PREFIX || 'SV+') and OS_POINTER to
               ''. */
      int I  /* Temporary. */

      if OS_LINKAGE
        then
          OS_POINTER := '(13)'
          LOCAL_POINTER := ''
          if SAVE(3) = 'NONE'
            then
              SAVE_TYPE := 'NONE'
            else
              SAVE_TYPE := 'OSSAVE'  /* Standard OS save area. */
              OFFSET_TO_GPR0, OFFSET := 5 /* Put offset to reg 0 in both. */
              if SAVE(4) = ''
                then
                  SAVE_LENGTH := '18' /* Standard OS save area is 18 words. */
                else
                  SAVE_LENGTH := SAVE(4) /* Length specified. */
                fi
            fi
        else /* Not OS_LINKAGE. */
          OS_POINTER := ''
          LOCAL_POINTER := BLOCK_LABEL_PREFIX || 'SV+'
          if DEBUG_SAVETRACE_REQD
            then
              SAVE_TYPE := 'FULL'  /* Full 18 word pseudo-OS save area. */
              SAVE_LENGTH := '15'
                  /* Length of register part of full save area is 15 words. */
              OFFSET_TO_GPR0, OFFSET := 5
            else
              I := SAL - ((SAL+2)/16*16) - SAF + ((SAF+2)/16*16) + 1
              SAVE_LENGTH := I
                  /* Convert calculated length to charter string. */
              if SAF = 14
                then /* SAVE(1) was omitted, specified as 14, or symbolic. */
                  if DEBUG_PROCCOUNTS_REQD
                    then /* Header included for count. */
                      SAVE_TYPE := 'NORMLHDR'  /* First register word is 14. */
                      OFFSET_TO_GPR0, OFFSET := 3
                    else
                      SAVE_TYPE := 'NORML'
                      OFFSET_TO_GPR0, OFFSET := 2
                    fi
                else /* Save area is to start after 14:  a truncated area. */
                  if DEBUG_PROCCOUNTS_REQD
                    then /* header included for count. */
                      SAVE_TYPE := 'TRUNCHDR'
                      OFFSET := 1 /* To SAVE(1). */
                    else
                      SAVE_TYPE := 'TRUNC'
                      OFFSET := 0 /* To SAVE(1). */
                    fi
                  OFFSET_TO_GPR0 := OFFSET - SAF
                  if SAF > 13
                    then
                      OFFSET_TO_GPR0 := OFFSET_TO_GPR0 + 16
                    fi
                fi
            fi
        fi
      if FIRST_VALUE_KNOWN /* SAVE(1) was not symbolic */ and
              SAVE_TYPE(1,5) ≠ 'TRUNC'
        then
              /* Adjust OFFSET from giving offset to GPR0 to give offset to SAF. */
          OFFSET := OFFSET_TO_GPR0 + SAF
          if SAF > 13
            then
              OFFSET := OFFSET - 16
            fi
        fi
    corp
```

```
81611.  proc PROC_ESTABLISH_BASE
81612.          /* Set up base register and issue USING where required. */
81613.      bit
81614.          INLINE_SAVEAREA  /* Indicates whether an inline save area is to
81615.                              be generated. */
81616.      char
81617.          BASEREG   /* Name of register loaded with base value. */
81618.      int
81619.          I, J  /* Temporaries. */
```

```
81621.       if BASE ≠ 'NONE' and (OS_LINKAGE or BASE ≠ '')
81622.         then /* Generate a base register. */
81623.           BASEREG := BASE(1)
81624.           INLINE_SAVEAREA := (SAVE ≠ 'NONE' and SAVE(3) = '')
81625.           if BASEREG = '13' and ¬(INLINE_SAVEAREA and OS_LINKAGE)
81626.             then
81627.               mnote (8, 'STRC8109 REGISTER 13 INVALID--IGNORED')
81628.               BASEREG := ''
81629.             fi
81630.           if BASEREG = ''
81631.             then /* No base register specified. */
81632.               if INLINE_SAVEAREA and OS_LINKAGE
81633.                 then
81634.                   BASEREG := WORKREG
81635.                       /* We will load the base value first into the work register, then
81636.                          copy the value to register 13 after we finish all linkage. */
81637.                   USING13 := true
81638.                   if WORK = 'NONE' and ¬ WORKREG_USED
81639.                     then
81640.                       generate (LABEL || '  LR    0,' || WORKREG)
81641.                       LABEL := ''
81642.                     fi
81643.                   WORKREG_USED := true
81644.                 else
81645.                   BASEREG := '12'
81646.                 fi
81647.             fi
81648.           J := 0
81649.           I := 2
81650.           while I ≤ N'BASE
81651.             do
81652.               if BASE(I) = '13'
81653.                 then
81654.                   mnote (8, 'STRC8109 REGISTER 13 INVALID--IGNORED')
81655.                 else
81656.                   generate (LABEL || '  L     ' || BASE(I) || ',' ||
81657.                             BLOCK_LABEL_PREFIX || 'BB+' || J)
81658.                   LABEL := ''
81659.                 fi
81660.               I := I + 1
81661.               J := J + 4
81662.             od /* (Termination:  I is incremented in loop, N'BASE is fixed;
81663.                     I must eventually exceed N'BASE.) */
81664.           if INLINE_SAVEAREA
81665.             then
81666.               if OS_LINKAGE
81667.                 then
81668.                   generate ('       CNOP  0,4')
81669.                       /* Advance to fullword boundary;  outstanding label can wait for next
81670.                          instruction. */
81671.                 fi
81672.               TARGET := BLOCK_LABEL_PREFIX || 'BB'
81673.               generate (LABEL || '  BAL   ' || BASEREG || ',' || TARGET)
81674.               LABEL := TARGET
81675.             else /* No inline save area. */
81676.               generate (LABEL || '  BALR  ' || BASEREG || ',0')
81677.               LABEL := ''
81678.             fi
81679.           if ¬ USING13
81680.             then
81681.               generate ('       USING *,' || BASEREG)
81682.             fi
81683.           if N'BASE > 1
81684.             then
81685.               generate (BLOCK_LABEL_PREFIX || 'MBP  EQU   *')
81686.               MULTIBASE := true
81687.               J := 4096
81688.               I := 2
81689.               while I ≤ N'BASE
81690.                 do
81691.                   generate ('         USING *+' || J || ',' || BASE(I))
81692.                   I := I + 1
81693.                   J := J + 4096
81694.                 od /* (Termination:  Same proof as above.) */
81695.             fi
81696.         fi
81697.     corp
```

```
81699.   proc PROC_GEN_SAVEAREA
81700.           /* Generate appropriate save area according to SAVE_TYPE, SAVE_LENGTH,
81701.              and the SAVE suboperands. */

81703.       if SAVE_TYPE = 'OSSAVE'
81704.         then
81705.           call PROC_GEN_OSSAVE_AREA
81706.                /* Generate OS save area and chain it up following OS linkage
81707.                   conventions.  Also link up static chain of local save areas if
81708.                   this is the first proc and SAVETRACE requested. */
81709.         else
81710.           if (SAVE ¬= 'NONE' and SAVE_TYPE ¬= 'NONE') or
81711.                DEBUG_SAVETRACE_REQD
81712.             then
81713.               call PROC_GEN_LOCAL_SAVEAREA
81714.                    /* Generate local PROC save area according to SAVE_TYPE and SAVE_LENGTH
81715.                       and, if SAVETRACE requested, provide static save area chaining. */
81716.             fi
81717.         fi
81718.     corp
```

```
81720.   proc PROC_GEN_OSSAVE_AREA
81721.          /* Generate OS save area and chain it up following OS linkage
81722.              conventions.  Also link up static chain of local save areas if
81723.              SAVETRACE requested. */

81725.       call PROC_DEFINE_NEW_OSSAVE
81726.          /* Generate inline, out-of-line, or dynamic save area and point to
81727.              it with BASEREG or WORKREG;  put register name in SAVEREG. */
81729.       if DEBUG_SAVETRACE_REQD and FIRST_PROC
81729.         then
81730.              /* Static chain of local save areas must be linked to OS save areas. */
81731.           generate (LABEL || '  ST    13,$FIRSTSV+4')
81732.           LABEL := ''
81733.           generate ('        MVC   8(4,13),=A($FIRSTSV)')
81734.           generate ('        L     13,=A($LASTSAV)')
81735.           PREV_SAVETRACE_PTR := '$FIRSTSV'
81736.           PREV_SAVETRACE_AREA := '0'
81737.         fi
81738.       generate (LABEL || '  ST    ' || SAVEREG || ',8(13)')
81739.       LABEL := ''
81740.       generate ('        ST    13,4(' || SAVEREG || ')')
81741.       generate ('        LR    13,' || SAVEREG)
81742.       if USING13
81743.         then /* 13 now loaded--issue USING. */
81744.           generate ('        USING ' || BLOCK_LABEL_PREFIX || 'ISV,13')
81745.         fi
81746.       if DEBUG_PROCTRACE_REQD and FIRST_PROC and SAVE(3) ^= ''
81747.         then
81748.              /* Stash pointer to PROC trace vector in word 1 of OS save area. */
81749.           if WORK = 'WORK' and ¬ WORKREG_USED
81750.             then
81751.               generate ('        LR    0,WORKREG')
81752.             fi
81753.           generate ('        LA    ' || WORKREG || ',$TRACE')
81754.           generate ('        ST    ' || WORKREG || ',0(13)')
81755.           WORKREG_USED := true
81756.         fi
81757.     corp
```

```
81759.   proc PROC_DEFINE_NEW_OSSAVE
81760.           /* Generate inline, out-of-line, or dynamic save area for
81761.              OS_LINKAGE and point to it with the WORKREG or BASEREG.  Put name of
81762.              pointing register in SAVEREG. */
81763.        char X   /* Temporary. */

81765.        X := ''
81766.        SAVEREG := WORKREG  /* Assumed. */
81767.        if SAVE(3) = 'DYNAM'
81768.          then
81769.            generate (LABEL || '  LA    0,(' || SAVE_LENGTH || ')*4')
81770.            LABEL := ''
81771.            if WORK ¬= 'NONE'
81772.              then
81773.                generate ('         LR    ' || WORKREG || ',1')
81774.              fi
81775.            generate ('        GETMAIN  R,LV=(0)')
81776.            generate ('        LR    0,' || WORKREG)
81777.            generate ('        LR    ' || WORKREG || ',1')
81778.            if WORK = 'NONE'
81779.              then
81780.                generate ('         L     1,24(13)')
81781.                if FIRST ¬= '14' or LAST ¬= '12'
81782.                  then
81783.                    note (4, 'STRC8107 REG 1 MUST BE AMONG THOSE SAVED')
81784.                  fi
81785.              else
81786.                generate ('         LR    1,0')
81787.              fi
81788.            WORKREG_USED := true
81789.          else
81790.            if SAVE(3) = ''
81791.              then /* Inline save area. */
81792.                if LABEL = ''
81793.                  then
81794.                    if WORK = 'NONE' and ¬ WORKREG_USED
81795.                      then
81796.                        generate ('         LR    0,' || WORKREG)
81797.                      fi
81798.                    WORKREG_USED := true
81799.                    generate ('        CNOP  0,4')
81800.                    LABEL := BLOCK_LABEL_PREFIX || 'CC'
81801.                    generate ('        BAL   ' || WORKREG || ',' || LABEL)
81802.                  else
81803.                    X := BLOCK_LABEL_PREFIX || 'ISV'
81804.                    if ¬ USING13
81805.                      then
81806.                        SAVEREG := BASEREG
81807.                      fi
81808.                  fi
81809.                if DEBUG_PROCTRACE_REQD and FIRST_PROC
81810.                  then
81811.                    generate (X || '  DC    A($TRACE),(' || SAVE_LENGTH || "-1)F'0'")
81812.                        /* Generate inline save area with first word pointing to trace
81813.                           vector. */
81814.                  else
81815.                    generate (X || '  DC    (' || SAVE_LENGTH || ")F'0'")
81816.                  fi
81817.                call PROC_MULTIBASE_GEN
81818.                  /* Insert multiple base adcons after save area. */
81819.              else /* User-supplied out-of-line save area. */
81820.                if WORK = 'NONE' and ¬ WORKREG_USED
81821.                  then
81822.                    generate (LABEL || '  LR    0,' || WORKREG)
81823.                    LABEL := ''
81824.                  fi
81825.                WORKREG_USED := true
81826.                generate (LABEL || '  LA    ' || WORKREG || ',' || SAVE(3))
81827.                LABEL := ''
81828.              fi
81829.          fi
81830.            /* Area has been generated and address is in SAVEREG register. */
81831.      corp
```

```
proc PROC_GEN_LOCAL_SAVEAREA
        /* Generate local PROC save area according to SAVE_TYPE and SAVE_LENGTH
            and, if SAVETRACE requested, provide static save area chaining.  */
    char FWD_PTR /* Name used for next save area. */

    if LABEL = ''
      then
        LABEL := BLOCK_LABEL_PREFIX || 'DD'
        generate ('        B       ' || LABEL)
      fi
    generate (BLOCK_LABEL_PREFIX || 'SV   DS     0F')
    if SAVE_TYPE = 'FULL' or SAVE_TYPE[6,3] = 'HDR'
      then /* Word one should contain PROC count and ID byte. */
        generate ("        DC     X'FF" || PROC_ID_BYTE || "0000'" ||
                    ' FLAG (FE=ENTERED, FF=FINISHED), ID, COUNT')
      fi
    if SAVE_TYPE = 'FULL'
      then
        FWD_PTR := BLOCK_LABEL_PREFIX || 'NXT'
        generate (PREV_SAVETRACE_PTR || '  EQU    ' ||
                BLOCK_LABEL_PREFIX || 'SV')
        generate ('        DC     A(' || PREV_SAVETRACE_AREA || ',' ||
                FWD_PTR || ')')
        PREV_SAVETRACE_PTR := FWD_PTR
            /* Save label used as forward pointer. */
        PREV_SAVETRACE_AREA := BLOCK_LABEL_PREFIX || 'SV'
            /* Save name of this save area. */
      fi
    generate ('        DC     (' || SAVE_LENGTH || ")F'0'")
    call PROC_MULTIBASE_GEN
  corp
```

•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•

```
81865.   proc PROC_DEBUG_STUFF
81866.         /* Generate trace and count code for debugging. */
81867.       bit PCT_GENNED_WITH_VECTOR  /* Indicates whether -PCT labeled halfword
81868.                which holds PROC counter was generated following the trace vector. */
81869.       char COUNT_SPOT  /* Suffix of label for PROC counter. */

81871.       if DEBUG_PROCTRACE_REQD
81872.         then
81873.           if TRACE_VECTOR_GENNED
81874.             then /* Previously generated TRACE vector must be updated. */
81875.               if WORK = 'NONE' and ¬ WORKREG_USED
81876.                 then
81877.                   generate (LABEL || '  LR    0,' || WORKREG)
81878.                   LABEL := ''
81879.                 fi
81880.               generate (LABEL || '  L     ' || WORKREG || ',=A($TRACE)')
81881.               generate ('       MVC   0(256,' || WORKREG || '),1(' ||
81882.                     WORKREG || ')')
81883.               generate ('       MVI   256(' || WORKREG || "),X'" ||
81884.                     PROC_ID_BYTE || "'")
81885.               WORKREG_USED := true
81886.             else /* Trace vector must be generated. */
81887.               if LABEL = ''
81888.                 then /* Branch around trace vector. */
81889.                   LABEL := BLOCK_LABEL_PREFIX || 'EE'
81890.                   generate ('       B     ' || LABEL)
81891.                 fi
81892.               generate ("         DC    C'$TRACE'")
81893.               generate ("$TRACE   DC    256X'FF'")
81894.               TRACE_VECTOR_GENNED := true /* Only generate it once. */
81895.               if (DEBUG_PROCCOUNTS_REQD or DEBUG_BLOCKCOUNTS_REQD) and
81896.                     FIRST_PROC
81897.                 then
81898.                       /* PROC counter must be generated since word one of first save area
81899.                          points to proc trace vector, so we can't keep count there. */
81900.                   generate (BLOCK_LABEL_PREFIX || "PCT  DC    H'0' PROC COUNT")
81901.                   PCT_GENNED_WITH_VECTOR := true
81902.                 fi
81903.               generate (LABEL || '  MVC   $TRACE(256),$TRACE+1')
81904.               generate ("       MVI   $TRACE+256,X'" || PROC_ID_BYTE || "'")
81905.             fi
81906.           LABEL := ''
81907.         fi
81908.       if DEBUG_PROCCOUNTS_REQD or DEBUG_BLOCKCOUNTS_REQD
81909.         then /* We must update the count. */
81910.           if OS_LINKAGE or (SAVE = 'NONE' and ¬ DEBUG_SAVETRACE_REQD)
81911.             then /* Count will be in BLOCK_LABEL_PREFIX || 'PCT'. */
81912.               if ¬ PCT_GENNED_WITH_VECTOR
81913.                 then /* Define PCT. */
81914.                   if LABEL = ''
81915.                     then
81916.                       generate ('       NOP   0')
81917.                       generate (BLOCK_LABEL_PREFIX || 'PCT  EQU   *-2 PROC COUNT')
81918.                     else
81919.                       generate (BLOCK_LABEL_PREFIX || "PCT  DC    H'0' PROC COUNT")
81920.                     fi
81921.                 fi
81922.               COUNT_SPOT := 'PCT'
81923.             else
81924.               COUNT_SPOT := 'SV+2' /* Count is in local save area. */
81925.             fi
81926.           if WORK = 'NONE' and ¬ WORKREG_USED
81927.             then
81928.               generate (LABEL || '  LR    0,' || WORKREG)
81929.               LABEL := ''
81930.             fi
81931.           generate (LABEL || '  LH    ' || WORKREG || ',' ||
81932.                 BLOCK_LABEL_PREFIX || COUNT_SPOT)
81933.           LABEL := ''
81934.           generate ('       LA    ' || WORKREG || ',1(' || WORKREG || ')')
81935.           generate ('       STH   ' || WORKREG || ',' ||
81936.                 BLOCK_LABEL_PREFIX || COUNT_SPOT)
81937.           WORKREG_USED := true
81938.         fi
81939.       if DEBUG_SAVETRACE_REQD and ¬ OS_LINKAGE
81940.         then
81941.           generate (LABEL || '  MVI   ' || BLOCK_LABEL_PREFIX || "SV,X'FF'")
81942.           LABEL := ''  /* Mark PROC as entered. */
81943.         fi
81944.     corp
```

```
81946.   proc PROC_INFO_SAVE
81947.          /* Save all information needed at CORP time. */
81948.       char I  /* OFFSET_TO_GPRO, biased by 50 and converted to character
81949.                  format. */

81951.       OPERAND1(CURRENT_NEST_LEVEL) := FIRST
81952.       OPERAND2(CURRENT_NEST_LEVEL) := LAST
81953.       OPERAND3(CURRENT_NEST_LEVEL) := SAVE_LENGTH
81954.       OPERAND4(CURRENT_NEST_LEVEL) := PREVIOUS_DEBUG_VECTOR
81955.       I := OFFSET_TO_GPRO + 50
81956.          /* Bias value by 50 and convert to two-digit character string. */
81957.       INFORMATION(CURRENT_NEST_LEVEL) :=
81958.              I || OS_LINKAGE || (SAVE(3) = 'DYNAM') || FIRST_VALUE_KNOWN ||
81959.                  PROC_ID_BYTE || SPECIAL_PREFIX
81960.    corp
```

```
81962.   proc PROC_MULTIBASE_GEN
81963.           /* This proc generates any multiple base register adcons if
81964.               needed and not yet generated and notes that such adcons have
81965.               been generated. */
81966.       int I, J /* Temporaries. */
81967.       char X

81969.       if MULTIBASE
81970.         then /* Multibase adcons required but not yet generated. */
81971.           if LABEL = ''
81972.             then /* Branch around adcons. */
81973.               LABEL := BLOCK_LABEL_PREFIX || 'FF'
81974.               generate ('         B      ' || LABEL)
81975.             fi
81976.           I := 2
81977.           J := 4096
81978.           X := BLOCK_LABEL_PREFIX || 'MBR'
81979.           while I ≤ N'BASE
81980.             do
81981.               generate (X || '  DC     A(' || BLOCK_LABEL_PREFIX || 'MBP+' ||
81982.                         J || ')')
81983.               X := ''
81984.               I := I + 1
81985.               J := J + 4096
81986.             od /* [Termination:  I is incremented during loop, N'BASE is
81987.                         fixed;  I must eventually exceed N'BASE.] */
81988.           MULTIBASE := false
81989.         fi
81990.     corp
```

```
83001.   macro  CORP  (USER_NAME;  PROC_NAME, RETURN=, LINK=14, RESTORE=, RC=)
83002.           /* Defines the end of a procedure block.  The register or registers
83003.               indicated by RESTORE= are restored with the exception of those listed
83004.               in RETURN=.  If RESTORE= is omitted, all saved registers are restored
83005.               (except those in the RETURN= list).  The return code is set from the
83006.               RC= operand and return is made to the address specified by the
83007.               LINK= operand, unless LINK=NONE is specified. */

83009.           /* [Ground rule:  No CORP proc modifies CURRENT_NEST_LEVEL.
83010.               This can be shown via the cross-reference listing.] */

83012.       int
83013.           FIRST_SAVEAREA_REG,  /* Register number which may be placed into the
83014.                   first word of the save area.  This may be, for example, 14 when
83015.                   the first register saved is some symbolic of unknown value. */
83016.           OFFSET_TO_GPR0     /* Offset in save area (in words) to the storage
83017.                   place for GPR0.  This may be positive or negative. */
83018.       bit
83019.           ANY_REGS_SAVED,  /* Indicates whether any registers were saved in
83020.                   this proc. */
83021.           DYNAMIC_SAVEAREA, /* Indicates whether SAVE=(***,DYNAM,***)
83022.                   was coded on PROC. */
83023.           BCK_AREA_REQD,  /* Indicates whether BCK save area is needed. */
83024.           FIRST_VALUE_KNOWN, /* Indicates whether FIRST is other than a
83025.                   symbolic. */
83026.           FIRST_REST_VALUE_KNOWN, /* Indicates whether first register to be
83027.                   restored (in REST1) is other than symbolic. */
83028.           OS_LINKAGE     /* Indicates whether LINKAGE=(OS,***) was coded
83029.                   on PROC. */
83030.       char
83031.           LABEL, /* Any outstanding label waiting to be generated. */
83032.           GPR0_OFFSET_STRING, /* OFFSET_TO_GPR0 in character form. */
83033.           FIRST_REG_SAVED, LAST_REG_SAVED,
83034.               /* First and last registers saved at PROC time. */
83035.           MULT, COMMA2, /* Holds either a "M" and "," respectivly or else nulls
83036.                   to allow generation of either a "LM" or "L" instruction. */
83037.           LOCAL_POINTER, OS_POINTER, RESTORE_AREA,
83038.               /* Instruction segments to generate load instructions from proper
83039.                   save area. */
83040.           RC_REG, /* Register holding return code before restoring of
83041.                   registers. */
83042.           SAVE_LENGTH, /* Length of save area. */
83043.           REST1, REST2, /* First and last register to be restored. */
83044.           PROC_ID_BYTE, /* One-byte hex number used as identifier of current
83045.               proc in traces and the like. */
83046.           PREVIOUS_DEBUG_VECTOR   /* Value of debug switches (packed) before
83047.                   encountering this PROC or [if DEBUG=(***,GLOBAL) specified]
83048.                   null. */

83050.       call TRACE_PRINTER (  ;  'CORP')
83051.           /* Print macro name "CORP" in mnote if tracing on. */
83052.       if CURRENT_NEST_LEVEL > NESTING_LIMIT
83053.         then
83054.           call POP_OLD_BLOCK (  ;  )
83055.           mexit
83056.         fi
83057.       call VERIFY_END (  ;  'PROC', PROC_NAME)
83058.           /* Verifies current block has the name specified in  the PROC_NAME
83059.               operand on the CORP macro (if any) and that it is a PROC block.
83060.               Various errors receive messages and either intermediate blocks are
83061.               BLENDed as a fixup or ERROR_OCCURRED is set.
83062.               [Lemma:  If CURRENT_NEST_LEVEL > 0 and
83063.                   [PROC_NAME = '' or = BLOCK_NAME(CURRENT_NEST_LEVEL)] and
83064.                   BLOCK_TYPE(CURRENT_NEST_LEVEL) = 'PROC',  then
83065.                   ERROR_OCCURRED will be set false and CURRENT_NEST_LEVEL will not
83066.                   be modified.] */
83067.       if ERROR_OCCURRED
83068.         then
83069.           mexit
83070.         fi
83071.       LABEL := USER_NAME
83072.           /* Generate label at first opportunity. */
```

```
83074.       call CORP_GET_PROC_INFO
83075.           /* Get info saved at PROC macro:  FIRST_REG_SAVED, LAST_REG_SAVED,
83076.                OS_LINKAGE, FIRST_SAVEAREA_REG, FIRST_VALUE_KNOWN, DYNAMIC_SAVEAREA,
83077.                SAVE_LENGTH, OFFSET_TO_GPRO, PREVIOUS_DEBUG_VECTOR, PROC_ID_BYTE,
83078.                GPRO_OFFSET_STRING, BLOCK_LABEL_PREFIX, and ANY_REGS_SAVED. */
83079.       call CORP_SET_RESTORE_RANGE
83080.           /* Set RESTORE_AREA to 'SV'.  Set REST1 and REST2 to RESTORE=
83081.                operands or, if omitted, then to FIRST_ and LAST_REG_SAVED. */
83082.       call CORP_GEN_EXIT_LABEL
83083.           /* If an EXIT label is required, put it into LABEL (generating
83084.                any label already there). */
83085.       if OS_LINKAGE
83086.         then
83087.           call CORP_RESTORE_REG13
83088.               /* Move register 13 pointer to point to previous save area, saving
83089.                    pointer to current area in register 1 if it is dynamic. */
83090.           if DYNAMIC_SAVEAREA
83091.             then
83092.               call CORP_FREE_DYNAM_SAVEAREA
83093.                   /* Issue FREEMAIN for dynamic core. */
83094.             fi
83095.         else
83096.           call CORP_DEBUGGING_STORES
83097.               /* If DEBUG_CORPVALUES_REQD, copy registers into CRP save area.  If
83098.                    CORPVALUES and RETURN= (or RC=) specified, copy SV save area to BCK
83099.                    save area, set RESTORE_AREA to 'BCK', and set BCK_AREA_REQD. */
83100.         fi
83101.       call CORP_SET_RETURN_CODE
83102.           /* If RC=value (or implied zero), load it to GPR15, except that if it
83103.                is in a register other than 15, leave it in that register.  Set
83104.                RC_REG with register name which contains RC at exit, if any. */
83105.       call CORP_SAVE_RETURNING_REGS
83106.           /* If any registers are to be restored, do the following:  for the
83107.                register containing the return code and all those listed in
83108.                RETURN=, store each register into the appropriate word of the
83109.                save area from which the ultimate LM instruction will be issued.
83110.                Also set OS_POINTER and LOCAL_POINTER to reflect proper save
83111.                area. */
83112.       if ¬ OS_LINKAGE and DEBUG_SAVETRACE_REQD
83113.         then
83114.           generate (LABEL || '  MVI   ' || BLOCK_LABEL_PREFIX || "SV,X'FF'")
83115.           LABEL := ''
83116.         fi
83117.       call CORP_RESTORE_REGISTERS
83118.           /* Restore REST1 through REST2 from proper save area if saved. */
83119.       if OS_LINKAGE
83120.         then
83121.           generate LABEL || "  MVI   12(13),X'FF'")
83122.               /* Set flag in previous save area to  show return. */
83123.           LABEL := ''
83124.         fi
83125.       if LINK ≠ 'NONE'
83126.         then
83127.           generate (LABEL || '  BR    ' || LINK)
83128.           LABEL := ''
83129.         fi
83130.       call CORP_GEN_CRP_BCK_AREAS
83131.           /* Generate the CRP and BCK save areas. */
83132.       if LABEL ≠ ''
83133.         then
83134.           generate (LABEL || '  DS    0H')
83135.           LABEL := ''
83136.         fi
83137.       call POP_OLD_BLOCK ( ; )
83138.           /* Delete PROC block from the stack.  (Lemma:  POP_OLD_BLOCK
83139.                decrements CURRENT_NEST_LEVEL by exactly one.) */
83140.       call CORP_RESTORE_DEBUG_ENVIRONMENT
83141.           /* Restore value of debugging switches from PREVIOUS_DEBUG_VECTOR. */
83142.     end
83143.       /* (Lemma:  If CURRENT_NEST_LEVEL > 0 and
83144.                [PROC_NAME = '' or = BLOCK_NAME(CURRENT_NEST_LEVEL)] and
83145.                BLOCK_TYPE(CURRENT_NEST_LEVEL) = 'PROC',  then
83146.                CURRENT_NEST_LEVEL will be decremented by exactly one.) */
```

"CORP" Macro -- 6 July 1971

```
83148.   proc CORP_GET_PROC_INFO
83149.           /* Get info saved by PROC macro. */
83150.       char X  /* Temporary. */

83152.       FIRST_REG_SAVED := OPERAND1(CURRENT_NEST_LEVEL)
83153.       LAST_REG_SAVED  := OPERAND2(CURRENT_NEST_LEVEL)
83154.       X               := INFORMATION(CURRENT_NEST_LEVEL)
83155.       OFFSET_TO_GPR0  := X[1,2] - 50  /* Stored biased by 50. */
83156.       if OFFSET_TO_GPR0 < 0
83157.         then
83158.           GPR0_OFFSET_STRING := '-' || OFFSET_TO_GPR0
83159.               /* In string conversion, absolute value is taken;  restore sign. */
83160.         else
83161.           GPR0_OFFSET_STRING := '+' || OFFSET_TO_GPR0
83162.         fi
83163.       OS_LINKAGE              := X[3,1]
83164.       DYNAMIC_SAVEAREA        := X[4,1]
83165.       FIRST_VALUE_KNOWN       := X[5,1]
83166.       SAVE_LENGTH             := OPERAND3(CURRENT_NEST_LEVEL)
83167.       PREVIOUS_DEBUG_VECTOR   := OPERAND4(CURRENT_NEST_LEVEL)
83168.       ANY_REGS_SAVED          := (FIRST_REG_SAVED ≠ '')
83169.       PROC_ID_BYTE            := X[6,2]
83170.       if X[8,1]
83171.         then /* Special PROC prefix. */
83172.           BLOCK_LABEL_PREFIX := '$P' || PROC_ID_BYTE
83173.         else /* Standard prefix. */
83174.           BLOCK_LABEL_PREFIX := '$' || BLOCK_NUMBER(CURRENT_NEST_LEVEL)
83175.         fi
83176.       if FIRST_VALUE_KNOWN
83177.         then
83178.           FIRST_SAVEAREA_REG := FIRST_REG_SAVED /* Convert to integer. */
83179.         else
83180.           FIRST_SAVEAREA_REG := 14
83181.         fi
83182.           /* FIRST_SAVEAREA_REG is similar to the variable SAP of PROC. */
83183.    corp
```

```
proc CORP_SET_RESTORE_RANGE
        /* Set REST1 and REST2 to RESTORE= operands if present, else to FIRST_
            and LAST_REG_SAVED.  Set RESTORE_AREA to "SV".  Also set MULT and
            COMMA2 to proper values. */

        /* Assume: */
    REST1 := FIRST_REG_SAVED
    REST2 := LAST_REG_SAVED
    MULT  := 'M'
    COMMA2 := ','
    FIRST_REST_VALUE_KNOWN := FIRST_VALUE_KNOWN
        /* Now find out. */
    if ¬ ANY_REGS_SAVED
      then
        if RESTORE ≠ ''
          then
            mnote (8, 'SIRC8301 NO REGISTERS SAVED--RESTORE IGNORED')
          fi
      else
        if RESTORE ≠ ''
          then
            REST1 := RESTORE(1)
            REST2 := RESTORE(2)
            FIRST_REST_VALUE_KNOWN := (T'RESTORE(1) = 'N')
                /* true iff first suboperand is a self-defining term. */
          fi
      fi
    if REST2 = ''
      then
        MULT, COMMA2 := ''
      fi
    RESTORE_AREA := 'SV'
  corp
```

•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•

```
proc CORP_DEBUGGING_STORES
        /* Given:  ¬ OS_LINKAGE.  If DEBUG_CORPVALUES_REQD, store copy of
            register values into CRP savearea.  If CORPVALUES and RETURN= (or
            RC=) specified, copy SV save area to BCK save area, set
            RESTORE_AREA to "BCK", and set BCK_AREA_REQD. */
    char
        CRP_BCK_OFFSET,  /* Character string to be inserted to insure
                registers are stored at proper offset from  CRP/BCK label. */
        SV_OFFSET  /* Character string to be inserted to reference proper
                offset from SV save area label. */

    if DEBUG_CORPVALUES_REQD
      then
        if DEBUG_SAVETRACE_REQD
          then
            SV_OFFSET := '+12'
            CRP_BCK_OFFSET := '+12'
          else
            CRP_BCK_OFFSET := ''
            if DEBUG_PROCCOUNTS_REQD
              then
                SV_OFFSET := '+4'
              else
                SV_OFFSET := ''
              fi
          fi
        generate (LABEL || '  STM   14,12,' || BLOCK_LABEL_PREFIX ||
                'CRP' || CRP_BCK_OFFSET)
        LABEL := ''
        if RETURN ≠ '' or (RC ≠ '' and ≠ 'NONE')
          then
            BCK_AREA_REQD := true
            generate ('        MVC    ' || BLOCK_LABEL_PREFIX || 'BCK' ||
                    CRP_BCK_OFFSET || '(' || SAVE_LENGTH || '*4),' ||
                    BLOCK_LABEL_PREFIX || 'SV' || SV_OFFSET)
            RESTORE_AREA := 'BCK'
          fi
      fi
  corp
```

•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•

```
83259.   proc CORP_RESTORE_REG13
83260.           /* If current save area is dynamic, save pointer to it in GPR1.  In
83261.               any case, load GPR13 to point to previous save area.  Given:
83262.               OS_LINKAGE is true. */

83264.       if DYNAMIC_SAVEAREA
83265.         then
83266.           generate (LABEL || '  LR     1,13')
83267.           LABEL := ''
83268.         fi
83269.       if SAVETRACE_ON_FIRST_PROC and PROC_ID_BYTE = '01'
83270.         then
83271.           generate (LABEL || '  L      13,$FIRSTSV+4')
83272.         else
83273.           generate (LABEL || '  L      13,4(13)')
83274.         fi
83275.       LABEL := ''
83276.     corp
```

```
83278.   proc CORP_SET_RETURN_CODE
83279.           /* If RC=value (or implied zero), load value into GPR15, but nop if
83280.              RC=(reg).  Note in RC_REG what register (if any) contains RC at
83281.              exit. */

83283.       RC_REG := '' /* Indicate no return code. */
83284.       if RC = ''
83285.         then
83286.           if OS_LINKAGE
83287.             then
83288.               generate (LABEL || '  SR    15,15')
83289.                  /* Clear 15 for normal OS return. */
83290.               LABEL := ''
83291.               RC_REG := '15'
83292.             fi
83293.         else
83294.           if RC[1,1] = '('
83295.             then /* Register was specified. */
83296.               RC_REG := RC(1)
83297.                  /* Note what register return code is in. */
83298.             else /* Value was specified. */
83299.               if RC ¬= 'NONE'
83300.                 then
83301.                   if RC = '0'
83302.                     then
83303.                       generate (LABEL || '  SR    15,15')
83304.                     else
83305.                       generate (LABEL || '  LA    15,' || RC)
83306.                     fi
83307.                   LABEL := ''
83308.                   RC_REG := '15'
83309.                 fi
83310.             fi
83311.           fi
83312.    corp
```

```
83314.   proc CORP_SAVE_RETURNING_REGS
83315.         /* For the register containing the return code and all those listed
83316.            in RETURN=, store each register into the appropriate word of the
83317.            save area from which the ultimate LM instruction will occur
83318.            (setting OS_POINTER and LOCAL_POINTER to indicate this save area).
83319.            However, if no registers are to be restored, then this proc
83320.            is a nop. */
83321.       int OFFSET, I   /* Temporaries. */

83323.       if ANY_REGS_SAVED
83324.         then
83325.           if OS_LINKAGE
83326.             then
83327.               OS_POINTER := '(13)'
83328.               LOCAL_POINTER := ''
83329.             else
83330.               OS_POINTER := ''
83331.               LOCAL_POINTER := BLOCK_LABEL_PREFIX || RESTORE_AREA || '+'
83332.           fi
83333.           if RC_REG ^= ''
83334.             then /* RC must be restored. */
83335.               if FIRST_SAVEAREA_REG < 14
83336.                 then /* We will not be changing RC;  load it now. */
83337.                   if RC_REG ^= '15'
83338.                     then
83339.                       generate (LABEL || '  LR    15,' || RC_REG)
83340.                       LABEL := ''
83341.                     fi
83342.                 else
83343.                   OFFSET := (OFFSET_TO_GPR0 - 1) * 4
83344.                   generate (LABEL || '  ST    ' || RC_REG || ',' ||
83345.                             LOCAL_POINTER || OFFSET || OS_POINTER)
83346.                   LABEL := ''
83347.                 fi
83348.             fi
83349.           if FIRST_SAVEAREA_REG ^= 14 and N'RETURN > 0
83350.             then
83351.               mnote (4, 'STRC8302 WARNING--NO CHECK MADE TO INSURE RETURNING '
83352.                         || 'REGISTERS ARE AMONG THOSE SAVED IN TRUNCATED SAVE AREA')
83353.             fi
83354.           I := 1
83355.           while I ≤ N'RETURN
83356.             do
83357.               if T'RETURN(I) = 'N' /* Self-defining term. */
83358.                 then
83359.                   OFFSET := (OFFSET_TO_GPR0 + RETURN(I) - ((RETURN(I) + 2)/16*16)) * 4
83360.                   generate (LABEL || '  ST    ' || RETURN(I) || ',' ||
83361.                             LOCAL_POINTER || OFFSET || OS_POINTER)
83362.                 else
83363.                   if FIRST_SAVEAREA_REG < 14
83364.                     then
83365.                       generate (LABEL || '  ST    ' || RETURN(I) || ',' ||
83366.                                 LOCAL_POINTER || '(' || RETURN(I) || GPR0_OFFSET_STRING ||
83367.                                 ')*4' || OS_POINTER)
83368.                     else
83369.                       generate (LABEL || '  ST    ' || RETURN(I) || ',' ||
83370.                                 LOCAL_POINTER || '(' || RETURN(I) || GPR0_OFFSET_STRING ||
83371.                                 '-((' || RETURN(I) || '+2)/16*16))*4' || OS_POINTER)
83372.                   fi
83373.               fi
83374.               I := I + 1
83375.               LABEL := ''
83376.             od /* {Termination:  I is incremented, N'RETURN is fixed in loop;
83377.                    I must eventually exceed N'RETURN.} */
83378.         fi
83379.     corp
```

```
83381.    proc CORP_FREE_DYNAM_SAVEAREA
83382.            /* Issue FREEMAIN for dynamic save area. */
83383.         generate (LABEL || '  LA    0,' || SAVE_LENGTH || '*4')
83384.         LABEL := ''
83385.         generate ('        FREEMAIN  R,LV=(0),A=(1)')
83386.      corp
```

```
83388.   proc CORP_RESTORE_REGISTERS
83389.           /* Restore registers REST1 through REST2 from proper save area if
83390.              saved. */
83391.       int OFFSET, I  /* Temporaries. */

83393.       if ANY_REGS_SAVED
83394.         then
83395.           if FIRST_REST_VALUE_KNOWN
83396.             then
83397.               I := REST1 /* Convert to integer. */
83398.               OFFSET := (OFFSET_TO_GPR0 + I - ((I + 2)/16*16)) * 4
83399.               generate (LABEL || '  L' || MULT || '    ' || REST1 || ',' ||
83400.                         REST2 || COMMA2 || LOCAL_POINTER || OFFSET || OS_POINTER)
83401.             else
83402.               if FIRST_SAVEAREA_REG < 14
83403.                 then
83404.                   generate (LABEL || '  L' || MULT || '    ' || REST1 || ',' ||
83405.                             REST2 || COMMA2 || LOCAL_POINTER || '(' || REST1 ||
83406.                             GPR0_OFFSET_STRING || ')*4' || OS_POINTER)
83407.                 else
83408.                   generate (LABEL || '  L' || MULT || '    ' || REST1 || ',' ||
83409.                             REST2 || COMMA2 || LOCAL_POINTER || '(' || REST1 ||
83410.                             GPR0_OFFSET_STRING || '-((' || REST1 || '+2)/16*16))*4' ||
83411.                             OS_POINTER)
83412.                 fi
83413.               fi
83414.             LABEL := ''
83415.           fi
83416.       corp
```

```
83418.   proc CORP_RESTORE_DEBUG_ENVIRONMENT
83419.           /* Restore debug flags which were in progress before the PROC (unless
83420.               GLOBAL caused null value to suppress restore).  Values are packed
83421.               in PREVIOUS_DEBUG_VECTOR and need only be unpacked. */

83423.       if PREVIOUS_DEBUG_VECTOR ^= ''
83424.         then
83425.           DEBUG_BLOCKNAMES_REQD  := PREVIOUS_DEBUG_VECTOR[1,1]
83426.           DEBUG_PROCNAMES_REQD   := PREVIOUS_DEBUG_VECTOR[2,1]
83427.           DEBUG_LISTBLOCKS_REQD  := PREVIOUS_DEBUG_VECTOR[3,1]
83428.           DEBUG_BLOCKCOUNTS_REQD := PREVIOUS_DEBUG_VECTOR[4,1]
83429.           DEBUG_PROCCOUNTS_REQD  := PREVIOUS_DEBUG_VECTOR[5,1]
83430.           DEBUG_PROCTRACE_REQD   := PREVIOUS_DEBUG_VECTOR[6,1]
83431.           DEBUG_CORPVALUES_REQD  := PREVIOUS_DEBUG_VECTOR[7,1]
83432.           DEBUG_SAVETRACE_REQD   := PREVIOUS_DEBUG_VECTOR[8,1]
83433.         fi
83434.     corp
```

```
83436.   proc CORP_GEN_CRP_BCK_AREAS
83437.           /* If required, generate CRP and BCK save areas. */
83438.       char
83439.           LAST_AREA,  /* Label of CRP or BCK area, whichever is generated
83440.                  last. */
83441.           FWD_PTR, /* Label generated as forward pointer in last area. */
83442.           TARGET   /* Temporary. */

83444.       if ¬ OS_LINKAGE and DEBUG_CORPVALUES_REQD
83445.         then  /* We need a CRP save area. */
83446.           if LINK = 'NONE'
83447.             then /* We must generate branch around save areas. */
83448.               TARGET := BLOCK_LABEL_PREFIX || 'FIN'
83449.               generate (LABEL || '  B     ' || TARGET)
83450.               LABEL := TARGET
83451.             fi
83452.           LAST_AREA := BLOCK_LABEL_PREFIX || 'CRP'
83453.           generate (LAST_AREA || '  DS    0F')
83454.           if DEBUG_SAVETRACE_REQD
83455.             then
83456.               generate ("       DC    X'FC" || PROC_ID_BYTE || "0000'")
83457.               if BCK_AREA_REQD
83458.                 then
83459.                   FWD_PTR := BLOCK_LABEL_PREFIX || 'BCK'
83460.                 else
83461.                   FWD_PTR := BLOCK_LABEL_PREFIX || 'FWD'
83462.                 fi
83463.               generate (PREV_SAVETRACE_PTR || '  EQU   ' ||
83464.                       BLOCK_LABEL_PREFIX || 'CRP')
83465.               generate ('       DC    A(' || PREV_SAVETRACE_AREA || ',' ||
83466.                       FWD_PTR || ')')
83467.             fi
83468.           generate ("       DC    15F'0'")
83469.           if BCK_AREA_REQD
83470.             then /* We need the BCK save area. */
83471.               LAST_AREA := BLOCK_LABEL_PREFIX || 'BCK'
83472.                   /* The BCK area is now the last one generated. */
83473.               generate (LAST_AREA || '  DS    0F')
83474.               if DEBUG_SAVETRACE_REQD
83475.                 then
83476.                   generate ("       DC    X'FB" || PROC_ID_BYTE || "0000'")
83477.                   FWD_PTR := BLOCK_LABEL_PREFIX || 'FWD'
83478.                   generate ('       DC    A(' || BLOCK_LABEL_PREFIX || 'CRP,' ||
83479.                           FWD_PTR)
83480.                 fi
83481.               generate ('       DC    (' || SAVE_LENGTH || ")F'0'")
83482.             fi
83483.           PREV_SAVETRACE_PTR := FWD_PTR
83484.           PREV_SAVETRACE_AREA := LAST_AREA
83485.         fi
83486.     corp
```

"CORP" Macro -- 6 July 1973

```
83488.    proc CORP_GEN_EXIT_LABEL
83489.            /* If an EXIT label is required, put it into LABEL (generating
83490.               any label already there).  Issue mnote regarding EXIT references. */

83492.        if EXIT_LABEL_REQD(CURRENT_NEST_LEVEL)
83493.          then
83494.            if LABEL ≠ ''
83495.              then
83496.                generate (LABEL || '  DS    0H')
83497.              fi
83498.            LABEL := BLOCK_LABEL_PREFIX || 'EXIT'
83499.            if EXIT_SEVERITY = ''
83500.              then
83501.                EXIT_SEVERITY := '4'
83502.              fi
83503.            mnote (EXIT_SEVERITY,
83504.                           "SPRC8303 ONE OR MORE EXIT'S REFERENCE THIS POINT")
83505.            EXIT_LABEL_REQD(CURRENT_NEST_LEVEL) := false
83506.                /* EXIT label will have been generated by POP_OLD_BLOCK time. */
83507.          fi
83508.      corp
```

```
91001.   macro  EXIT_FIND  (  ;  REQD_NAME)
91002.           /* Set ULTIMATE_BRANCH_LABEL to exit label for block whose name
91003.                is the argument;  if no such block, issue message and set
91004.                ERROR_OCCURRED.  On valid block, that block is marked as needing
91005.                an XIT label. */
91006.        int I  /* Temporary. */

91008.        call TRACE_PRINTER (  ;  'EXITFIND')
91009.             /* Print macro name "EXITFIND" in mnote if tracing on. */
91010.        ERROR_OCCURRED := false
91011.             /* Assume all will go well. */
91012.        I := CURRENT_NEST_LEVEL - 1 /* Start search at surrounding block. */
91013.        if REQD_NAME ≠ '' and ≠ '*'
91014.          then /* We must search for the right block. */
91015.            while I > 0 and REQD_NAME ≠ BLOCK_NAME(I) and
91016.                  BLOCK_TYPE(I) ≠ 'PROC'
91017.              do
91018.                I := I - 1
91019.              od /* (Termination:  I is decremented and would eventually become
91020.                         ≤ 0 even if other tests never occurred.) */
91021.          fi
91022.        if I ≤ 0 or
91023.             (REQD_NAME ≠ BLOCK_NAME(I) and ≠ '' and ≠ '*')
91024.          then /* Not found in search. */
91025.            ERROR_OCCURRED := true
91026.            if REQD_NAME = '' or = '*'
91027.              then /* Didn't even search;  EXIT not nested. */
91028.                mnote (8, 'STRC9101 EXIT MACRO NOT SUFFICIENTLY NESTED')
91029.              else
91030.                if REQD_NAME = BLOCK_NAME(CURRENT_NEST_LEVEL)
91031.                  then
91032.                    mnote (8, 'STRC9103 EXIT TO IMMEDIATELY SURROUNDING BLOCK INVALID')
91033.                  else
91034.                    mnote (8, 'STRC9102 NO BLOCK ACTIVE NAMED "' || REQD_NAME || '"')
91035.                    if I > 0
91036.                      then
91037.                        mnote (*, '       WITHIN PROC ' || BLOCK_NAME(I))
91038.                      fi
91039.                  fi
91040.              fi
91041.          else /* Found. */
91042.            if BLOCK_TYPE(I) = 'DO' and
91043.                       (INFORMATION(CURRENT_NEST_LEVEL)[6,1] or
91044.                        INFORMATION(CURRENT_NEST_LEVEL)[7,1])
91045.              then
91046.                mnote (8, 'STRC9104 EXIT TO DO BLOCK INVALID WITHIN ATEND OR ONEXIT')
91047.                ERROR_OCCURRED := true
91048.              else
91049.                if BLOCK_TYPE(I) = 'PROC' and INFORMATION(I)[8,1]
91050.                  then /* Must use special PROC prefix form. */
91051.                    ULTIMATE_BRANCH_LABEL := '$P' || INFORMATION(I)[6,2] || 'XIT'
91052.                  else
91053.                    ULTIMATE_BRANCH_LABEL := '$' || BLOCK_NUMBER(I) || 'XIT'
91054.                  fi
91055.                EXIT_LABEL_REQD(I) := true
91056.              fi
91057.          fi
91058.   mend
```

"POP_OLD_BLOCK" Macro -- 10 July 1973

```
92001.    macro  POP_OLD_BLOCK  (  ; OLD_EXIT)
92002.            /* Remove the current block from the stack.  Also generate END and XIT
92003.               labels if required. */

92005.        call TRACE_PRINTER (  :  'POP')
92006.            /* Print macro name "POP" in mnote if tracing on. */
92007.        if CURRENT_NEST_LEVEL ≤ NESTING_LIMIT
92008.          then
92009.            if END_LABEL_REQD(CURRENT_NEST_LEVEL)
92010.              then
92011.                generate ('$' || BLOCK_NUMBER(CURRENT_NEST_LEVEL) || 'END  DS    0H')
92012.              fi
92013.            if EXIT_LABEL_REQD(CURRENT_NEST_LEVEL) or OLD_EXIT ≠ ''
92014.              then
92015.                if EXIT_SEVERITY = ''
92016.                  then
92017.                    EXIT_SEVERITY := '0'
92018.                  fi
92019.                mnote (EXIT_SEVERITY,
92020.                       "STRC9201 ONE OR MORE EXIT'S REFERENCE THIS POINT")
92021.                if EXIT_LABEL_REQD(CURRENT_NEST_LEVEL)
92022.                  then
92023.                    generate ('$' || BLOCK_NUMBER(CURRENT_NEST_LEVEL) || 'XIT  DS    0H')
92024.                  fi
92025.                if OLD_EXIT ≠ ''
92026.                  then
92027.                    generate (OLD_EXIT || '  DS    0H')
92028.                  fi
92029.              fi
92030.            if DEBUG_LISTBLOCKS_REQD
92031.              then
92032.                mnote (*, 'STRC9303 END OF BLOCK ' || BLOCK_NUMBER(CURRENT_NEST_LEVEL)
92033.                       || '(' || BLOCK_NAME(CURRENT_NEST_LEVEL) || ') AT DEPTH ' ||
92034.                       CURRENT_NEST_LEVEL)
92035.                mnote (*, '*******************************************************')
92036.              fi
92037.          fi
92038.        CURRENT_NEST_LEVEL := CURRENT_NEST_LEVEL - 1
92039.      mend
92040.        /* {Lemma:  Execution of POP_OLD_BLOCK always decrements
92041.                    CURRENT_NEST_LEVEL by exactly one.} */
```

```
macro PUSH_NEW_BLOCK (BLOCK_NAME_VALUE;
            BLOCK_TYPE_VALUE='',
            OPERAND1_VALUE='',
            OPERAND2_VALUE='',
            OPERAND3_VALUE='',
            OPERAND4_VALUE='',
            INFORMATION_VALUE='',
            END_LABEL_VALUE=false)
        /* Define new block; add to stack.  Save block specifications.
            All macro operands unspecified default to null string except
            END_LABEL_VALUE defaults to false. */

    call TRACE_PRINTER ( : 'PUSH')
        /* Print macro name "PUSH" in mnote if tracing on. */
    NESTING_LIMIT := 100
        /* Insure maximum depth of stack is set in variable.  Note that stack
            depth and this variable must match, but may be changed to any
            value. */
    GCASE_NEST_LIMIT := 9
        /* Same for general CASE stack. */
    ERROR_OCCURRED := false
    CURRENT_NEST_LEVEL := CURRENT_NEST_LEVEL + 1
    if CURRENT_NEST_LEVEL > NESTING_LIMIT
      then
        mnote (12, 'STRC9301 BLOCK NESTING LIMIT OF ' || NESTING_LIMIT ||
                ' EXCEEDED--MACROS MUST BE MODIFIED')
        ERROR_OCCURRED := true
      else
        LAST_BLOCK_NUMBER := LAST_BLOCK_NUMBER + 1
            /* Set block number for this block. */
        BLOCK_NUMBER(CURRENT_NEST_LEVEL)     := LAST_BLOCK_NUMBER
        EXIT_LABEL_REQD(CURRENT_NEST_LEVEL)  := false
        BLOCK_TYPE(CURRENT_NEST_LEVEL)       := BLOCK_TYPE_VALUE
        OPERAND1(CURRENT_NEST_LEVEL)         := OPERAND1_VALUE
        OPERAND2(CURRENT_NEST_LEVEL)         := OPERAND2_VALUE
        OPERAND3(CURRENT_NEST_LEVEL)         := OPERAND3_VALUE
        OPERAND4(CURRENT_NEST_LEVEL)         := OPERAND4_VALUE
        INFORMATION(CURRENT_NEST_LEVEL)      := INFORMATION_VALUE
        END_LABEL_REQD(CURRENT_NEST_LEVEL)   := END_LABEL_VALUE
        BLOCK_NAME(CURRENT_NEST_LEVEL)       := BLOCK_NAME_VALUE
        if BLOCK_NAME_VALUE = ''
          then
            BLOCK_NAME(CURRENT_NEST_LEVEL)  := 'BLK' || LAST_BLOCK_NUMBER
          fi
        BLOCK_LABEL_PREFIX := '$' || LAST_BLOCK_NUMBER
        if CURRENT_NEST_LEVEL > 1 and
                BLOCK_TYPE(CURRENT_NEST_LEVEL-1) = 'DOCASE' and
                BLOCK_TYPE_VALUE ≠ 'CASE'
          then
            mnote (8,
                'STRC9302 NON-CASE BLOCK IMMEDIATELY SURROUNDED BY DOCASE INVALID')
          fi
        if DEBUG_LISTBLOCKS_REQD
          then
            mnote (*, '**********************************************************')
            mnote (*, 'STRC9902 START OF BLOCK ' || LAST_BLOCK_NUMBER || '(' ||
                BLOCK_NAME(CURRENT_NEST_LEVEL) || ') AT DEPTH ' || CURRENT_NEST_LEVEL)
          fi
      fi
    mend
```

•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•o•

"SIMPLE_CONDITIONAL" Macro -- 10 July 1973

```
94001.   macro  SIMPLE_CONDITIONAL  (LABEL;
94002.                OP_CODE,
94003.                OPER1,
94004.                OPER2,
94005.                OPER3,
94006.                OPER4,
94007.                BRANCH_LABEL,
94008.                FALLTHRU_CONDITION,
94009.                OP_COUNT)
94010.           /* Generate indicated instruction followed by appropriate conditional
94011.              branch to indicated label. */
94012.       char
94013.           LOCAL_MASK, LOCAL_REL, /* Holds mask or relation for branch. */
94014.           BC_TAG  /* Label to go on bc instruction. */

94016.       call TRACE_PRINTER (  ;  'SIMPCOND')
94017.           /* Print macro name "SIMPCOND" in mnote if tracing on. */
94018.       call SIMPCOND_GET_MASK_OR_REL
94019.           /* Extract LOCAL_MASK or LOCAL_REL from OPER's.  If LOCAL_REL is a
94020.              extended value (GT, GE, EQ, LT, or LE), replace it with the
94021.              proper value (H, HL, E, L, or NH). */
94022.       docase OP_COUNT
94023.         of
94024.           case 1  /* Mask or relation only. */
94025.             BC_TAG := LABEL
94026.           esac
94027.           case 2
94028.             mnote (8, 'STRC9401 INSUFFICIENT OPERANDS FOR TEST "' || OP_CODE ||
94029.                       '"')
94030.           esac
94031.           case 3
94032.             generate (LABEL || '  ' || OP_CODE || '   ' || OPER1)
94033.           esac
94034.           case 4
94035.             generate (LABEL || '  ' || OP_CODE || '   ' || OPER1 || ',' ||
94036.                       OPER2)
94037.           esac
94038.           case 5
94039.             generate (LABEL || '  ' || OP_CODE || '   ' || OPER1 || ',' ||
94040.                       OPER2 || ',' || OPER3)
94041.           esac
94042.           case misc
94043.             mnote (8, 'STRC9402 SUPERFLUOUS OPERANDS FOR TEST "' || OP_CODE ||
94044.                       '"')
94045.           esac
94046.         esacod
94047.       if LOCAL_MASK = ''
94048.         then
94049.           if FALLTHRU_CONDITION /* is true: */
94050.             then /* Invert relation. */
94051.               if LOCAL_REL[1,1] = 'N'
94052.                 then
94053.                   LOCAL_REL := LOCAL_REL[2,7]
94054.                 else
94055.                   LOCAL_REL := 'N' || LOCAL_REL
94056.                 fi
94057.             fi
94058.           generate (BC_TAG || '  B' || LOCAL_REL || '     ' || BRANCH_LABEL)
94059.         else
94060.           if FALLTHRU_CONDITION /* is true: */
94061.             then /* Invert mask. */
94062.               generate (BC_TAG || "  BC     X'F'-" || LOCAL_MASK || ',' ||
94063.                         BRANCH_LABEL)
94064.             else
94065.               generate (BC_TAG || '  BC     ' || LOCAL_MASK || ',' || BRANCH_LABEL)
94066.             fi
94067.         fi
94068.     mend
```

```
94070.   proc SIMPCOND_GET_MASK_OR_REL
94071.           /* Extract LOCAL_MASK or LOCAL_REL from OPER's.  If LOCAL_REL is a
94072.              extended value (GT, GE, EQ, LT, or LE), replace it with the
94073.              proper value (H, NL, E, L, or NH). */

94075.       if OP_COUNT = 0
94076.         then
94077.           mnote (8, 'STRC9403 NO CONDITION SPECIFIED—"MASK=0" ASSUMED')
94078.           LOCAL_MASK := '0'
94079.         else
94080.           if SYSLIST(OP_COUNT)[1,5] = 'MASK='
94081.             then
94082.               LOCAL_MASK := SYSLIST(OP_COUNT)[6,8]
94083.             else
94084.               if SYSLIST(OP_COUNT)[1,4] = 'REL='
94085.                 then
94086.                   LOCAL_REL := SYSLIST(OP_COUNT)[5,8]
94087.                 else
94088.                   LOCAL_REL := SYSLIST(OP_COUNT)
94089.                 fi
94090.               docase LOCAL_REL ifany
94091.                 of
94092.                   case 'GT'
94093.                     LOCAL_REL := 'H'
94094.                   esac
94095.                   case 'GE'
94096.                     LOCAL_REL := 'NL'
94097.                   esac
94098.                   case 'EQ'
94099.                     LOCAL_REL := 'E'
94100.                   esac
94101.                   case 'LT'
94102.                     LOCAL_REL := 'L'
94103.                   esac
94104.                   case 'LE'
94105.                     LOCAL_REL := 'NH'
94106.                   esac
94107.                 esacpl
94108.             fi
94109.         fi
94110.     corp
```

"TRACE_PRINTER" Macro -- 10 July 1973

```
95001.    macro  TRACE_PRINTER  (  ;  MACRO_NAME)
95002.            /* Prints macro name if tracing on. */

95004.        if DEBUG_MACRONAMES_REQD
95005.          then
95006.            mnote (*, 'SPRC9500 ' || MACRO_NAME)
95007.          fi
95008.      mend
```

```
96001.  macro  VERIFY_END  (  ;  REQD_TYPE, REQD_NAME)
96002.          /* Verifies current block has name specified by REQD_NAME operand, if
96003.              any, and that it is of type REQD_TYPE.  Various errors receive
96004.              messages and either intermediate blocks are BLENDed as a fixup
96005.              or ERROR_OCCURRED is set. */
96006.      int I   /* Temporary. */
```

```
96008.        call TRACE_PRINTER ( ; 'VERIFY')
96009.           /* Print macro name "VERIFY" in mnote if tracing on. */
96010.        ERROR_OCCURRED := false /* Assumed. */
96011.        if REQD_NAME = ''
96012.          then
96013.            if CURRENT_NEST_LEVEL ≤ 0
96014.              then
96015.                mnote (8, 'STRC9607 NO BLOCKS ACTIVE--MACRO IGNORED')
96016.                ERROR_OCCURRED := true
96017.              else
96018.                if CURRENT_NEST_LEVEL ≥ 1 and
96019.                      BLOCK_TYPE(CURRENT_NEST_LEVEL) ≠ REQD_TYPE
96020.                  then
96021.                    if CURRENT_NEST_LEVEL ≥ 2 and
96022.                          BLOCK_TYPE(CURRENT_NEST_LEVEL - 1) = REQD_TYPE
96023.                      then
96024.                        mnote (8, 'STRC9601 ONE BLEND ASSUMED TO GET TO "' || REQD_TYPE ||
96025.                              '" BLOCK')
96026.                        call BLEND ( ; )
96027.                      else
96028.                        if CURRENT_NEST_LEVEL ≥ 3 and
96029.                              BLOCK_TYPE(CURRENT_NEST_LEVEL - 2) = REQD_TYPE
96030.                          then
96031.                            mnote (8, 'STRC9602 TWO BLENDS ASSUMED TO GET TO "' || REQD_TYPE ||
96032.                                  '" BLOCK')
96033.                            call BLEND ( ; )
96034.                            call BLEND ( ; )
96035.                          else
96036.                            mnote (8, 'STRC9603 CURRENT BLOCK IS NOT "' || REQD_TYPE ||
96037.                                  '" BLOCK--MACRO IGNORED')
96038.                            ERROR_OCCURRED := true
96039.                          fi
96040.                      fi
96041.                  fi
96042.            fi
96043.          else /* A block name was specified. */
96044.            I := CURRENT_NEST_LEVEL
96045.            while I > 0 and REQD_NAME ≠ BLOCK_NAME(I)
96046.              do
96047.                I := I - 1
96048.              od /* {Termination:  I is decremented and would eventually become
96049.                        ≤ 0 even if other test never occurs.} */
96050.            if I ≤ 0
96051.              then
96052.                mnote (8, 'STRC9604 NO ACTIVE BLOCK NAMED "' || REQD_NAME || '"')
96053.                ERROR_OCCURRED := true
96054.              else
96055.                if REQD_TYPE ≠ BLOCK_TYPE(I)
96056.                  then /* Block named found, but of wrong type. */
96057.                    mnote (8, 'STRC9605 BLOCK "' || REQD_NAME || '" IS NOT A ' ||
96058.                          REQD_TYPE || ' BLOCK--MACRO IGNORED')
96059.                    ERROR_OCCURRED := true
96060.                  else
96061.                    while CURRENT_NEST_LEVEL > I
96062.                      do /* BLEND any intermediate blocks. */
96063.                        mnote (8, 'STRC9606 END OF BLOCK "' || REQD_NAME ||
96064.                                '" IMPLIES END OF BLOCK "' ||
96065.                                BLOCK_NAME(CURRENT_NEST_LEVEL) || '"')
96066.                        call BLEND ( ; )
96067.                            /* {Lemma:  If CURRENT_NEST_LEVEL > 0 and no BLEND operands
96068.                                  specified, BLEND will decrement CURRENT_NEST_LEVEL
96069.                                  by exactly one.} */
96070.                      od /* {Termination:  On all iterations, I is fixed and
96071.                                CURRENT_NEST_LEVEL > I > 0.  But BLEND decrements
96072.                                CURRENT_NEST_LEVEL.  Therefore, CURRENT_NEST_LEVEL must
96073.                                eventually become ≤ (actually =) I.} */
96074.                    fi
96075.                fi
96076.            fi
96077.      mend
96078.        /* {Lemma:  If CURRENT_NEST_LEVEL > 0 and
96079.              [ REQD_NAME = '' or = BLOCK_TYPE(CURRENT_NEST_LEVEL) ] and
96080.              REQD_TYPE = BLOCK_TYPE(CURRENT_NEST_LEVEL), then
96081.              ERROR_OCCURRED will always be set false and CURRENT_NEST_LEVEL
96082.              will be unmodified.  Proof:  If the hypothesized conditions are
96083.              true, the module calls TRACE_PRINTER and sets ERROR_OCCURRED to
96084.              false as its only actions.  TRACE_PRINTER modifies no
96085.              globals.} */
```

```
97001.  macro  CONDITIONAL_EXPRESSION_PROCESSOR  (FIRST_ID;  )
97002.          /* Process the positional operands (the SYSLIST) as passed directly
97003.             from calling macro beginning with SYSLIST(FIRST_INDEX) through
97004.             SYSLIST(LAST_INDEX) generating the indicated tests to pass control
97005.             to ULTIMATE_FALLTHRU_LABEL when the ULTIMATE_FALLTHRU_CONDITION is
97006.             found to match the logical value tested and branches to the
97007.             ULTIMATE_BRANCH_LABEL otherwise;  the UNIQUE_LABEL_ID is used to
97008.             insure unique labels;  if a branch is made to the fall-through label,
97009.             FALLTHRU_LABEL_USED is set, else unaltered.  Only the "SYSLIST"
97010.             operand is passed as an actual macro operand.  The other variables
97011.             mentioned are globals. */
97012.      int
97013.          COND_COUNT,  /* Counts the simple conditionals within the
97014.                  conditional expression. */
97015.          DEPTH,  /* Angle bracket nesting depth of simple conditional being
97016.                  processed. */
97017.          INDEX,  /* Operand index within the SYSLIST of the simple conditional
97018.                  being processed. */
97019.          OP_COUNT, /* Number of suboperands in the simple conditional being
97020.                  processed. */
97021.          NEXT_INDEX, /* Index of the AND or OR which follows the current
97022.                  simple conditional. */
97023.          NEXT_DEPTH, /* Angle bracket depth of NEXT_INDEX. */
97024.          LA_DEPTH, /* Angle bracket depth during operand look-ahead. */
97025.          I         /* Operand index of operand being examined during
97026.                  look-ahead. */
97027.      bit
97028.          AND_OR_OUTSTANDING, /* Indicates whether an AND or OR follows
97029.                  the current simple conditional. */
97030.          LOCAL_FALLTHRU_CONDITION, /* Logical value of the simple conditional
97031.                  being processed which is to lead tc control falling through the
97032.                  test. */
97033.          LOCAL_LABEL_REQD(20)  /* Indicates whether the corresponding
97034.                  simple conditionals require a label due to branching logic. */
97035.      char
97036.          LABEL, /* Outstanding label waiting to be generated. */
97037.          LOCAL_BRANCH_LABEL  /* Label for branch target if current simple
97038.                  conditional has the opposite truth value from that stored in
97039.                  LOCAL_FALLTHRU_CONDITION. */
```

```
97041.       call TRACE_PRINTER (  ;  'CEP')
97042.           /* Print macro name "CEP" in mnote if tracing on. */
97043.       LABEL := FIRST_ID
97044.           /* Set calling label as outstanding. */
97045.       COND_COUNT := 0
97046.       DEPTH := 0
97047.       INDEX := FIRST_INDEX
97048.       while INDEX ≤ LAST_INDEX
97049.         do
97050.           call CEP_FIND_NEXT_CONDITION
97051.               /* Steps INDEX up to next simple conditional incrementing DEPTH for any
97052.                    "<" found and setting ERROR_OCCURRED on any syntax error.  Increments
97053.                    COND_COUNT for the condition found. */
97054.           if ERROR_OCCURRED
97055.             then
97056.               mexit
97057.             fi
97058.           AND_OR_OUTSTANDING := false
97059.           call CEP_LOOKAHEAD
97060.               /* Find LOCAL_BRANCH_LABEL or generate one.  If LOCAL_BRANCH_LABEL
97061.                    is the ULTIMATE_FALLTHRU_LABEL, set FALLTHRU_LABEL_USED.  Set
97062.                    LOCAL_FALLTHRU_CONDITION.  Also set NEXT_INDEX and NEXT_DEPTH with
97063.                    the INDEX/DEPTH of the AND or OR following this conditional.
97064.                    (The value of NEXT_INDEX returned is greater than the value
97065.                    of INDEX entered.) */
97066.           if ERROR_OCCURRED
97067.             then
97068.               mexit
97069.             fi
97070.           if LOCAL_LABEL_REQD(COND_COUNT)
97071.             then
97072.               LABEL := BLOCK_LABEL_PREFIX || UNIQUE_LABEL_ID || COND_COUNT
97073.             fi
97074.           OP_COUNT := N'SYSLIST(INDEX)
97075.           call SIMPLE_CONDITIONAL (LABEL;
97076.                    SYSLIST(INDEX,1),
97077.                    SYSLIST(INDEX,2),
97078.                    SYSLIST(INDEX,3),
97079.                    SYSLIST(INDEX,4),
97080.                    SYSLIST(INDEX,5),
97081.                    LOCAL_BRANCH_LABEL,
97082.                    LOCAL_FALLTHRU_CONDITION,
97083.                    OP_COUNT)
97084.           LABEL := ''
97085.           INDEX := NEXT_INDEX
97086.           DEPTH := NEXT_DEPTH
97087.         od /* (Termination:  INDEX is incremented (by CEP_LOOKAHEAD's
97088.                    return of NEXT_INDEX) and LAST_INDEX is fixed in loop;  INDEX must
97089.                    eventually exceed LAST_INDEX.) */
97090.       if AND_OR_OUTSTANDING
97091.         then
97092.           mnote (8, 'STRC9701 INSUFFICIENT OPERANDS')
97093.         fi
97094.       if DEPTH ≠ 0
97095.         then
97096.           mnote (8, 'STRC9702 INSUFFICIENT BRACKETS')
97097.         fi
97098.     mend
```

```
97100.   proc CEP_FIND_NEXT_CONDITION
97101.           /* Step up to next simple conditional, incrementing DEPTH for any "<"
97102.               (or "+") found.  Start with SYSLIST(INDEX) and advance INDEX till
97103.               found or LAST_INDEX.  Set ERROR_OCCURRED on any syntax error,
97104.               giving message.  Increment COND_COUNT by 1. */

97106.       ERROR_OCCURRED := false
97107.       while INDEX ≤ LAST_INDEX and (SYSLIST(INDEX) = '<' or = '+')
97108.         do
97109.           DEPTH := DEPTH + 1
97110.           INDEX := INDEX + 1
97111.         od /* (Termination:  INDEX is incremented, LAST_INDEX is fixed in
97112.                   loop;  INDEX would eventually exceed LAST_INDEX even if other
97113.                   tests never occur.) */
97114.       if SYSLIST(INDEX)[1,1] ≠ '('
97115.         then
97116.           mnote (8, 'STRC9706 SYNTAX ERROR--LOOKING FOR SIMPLE ' ||
97117.                   'CONDITIONAL, FOUND "' || SYSLIST(INDEX) || '"')
97118.           ERROR_OCCURRED := true
97119.         fi
97120.       COND_COUNT := COND_COUNT + 1
97121.     corp
```

```
97123.   proc CEP_LOOKAHEAD
97124.           /* Search operands beyond current simple conditional.  If AND/OR found,
97125.              set AND_OR_OUTSTANDING.  Find spot this test is to branch to and put
97126.              label found or generated for that spot into LOCAL_BRANCH_LABEL.  If
97127.              same as ULTIMATE_FALLTHRU_LABEL, set FALLTHRU_LABEL_USED.  Decide
97128.              whether this test is to fallthru on true or false and set
97129.              LOCAL_FALLTHRU_CONDITION.  If syntax error found during lookahead,
97130.              give message and set ERROR_OCCURRED.  Also set NEXT_INDEX and
97131.              NEXT_DEPTH with the index/depth of the AND/OR. */

97133.       ERROR_OCCURRED := false
97134.       LA_DEPTH := DEPTH
97135.       I := INDEX + 1
97136.       while I ≤ LAST_INDEX and (SYSLIST(I) = '>' or = '/')
97137.         do
97138.           I := I + 1
97139.           if LA_DEPTH > 0
97140.             then
97141.               LA_DEPTH := LA_DEPTH - 1
97142.             else
97143.               mnote (8, 'STEC9705 SUPERFLUOUS BRACKET IGNORED')
97144.             fi
97145.         od /* {Termination:  I is incremented, LAST_INDEX is fixed in
97146.                   loop;  INDEX would eventually exceed LAST_INDEX even if other
97147.                   tests never occur.} */
97148.       NEXT_INDEX := I + 1
97149.           /* {Lemma:  NEXT_INDEX > I > INDEX.} */
97150.       NEXT_DEPTH := LA_DEPTH
97151.       if I > LAST_INDEX
97152.         then
97153.           LOCAL_BRANCH_LABEL := ULTIMATE_BRANCH_LABEL
97154.           LOCAL_FALLTHRU_CONDITION := ULTIMATE_FALLTHRU_CONDITION
97155.         else
97156.           if SYSLIST(I) = 'AND' or = 'OR'
97157.             then
97158.               call CEP_SCAN_FOR_BRANCH
97159.                    /* Search ahead for branch target.  Set AND_OR_OUTSTANDING, LOCAL_
97160.                       FALLTHRU_CONDITION, and LOCAL_BRANCH_POINT. */
97161.             else
97162.               mnote (8, 'STEC9703 SYNTAX ERROR--LOOKING FOR "AND" OR "OR", ' ||
97163.                         'FOUND "' || SYSLIST(I) || '"')
97164.               ERROR_OCCURRED := true
97165.             fi
97166.         fi
97167.     corp
97168.       /* {Lemma:  The value of NEXT_INDEX returned is greater than the
97169.               value of INDEX entered.} */
```

```
97171.  proc CEP_SCAN_FOR_BRANCH
97172.          /* SYSLIST(I) is either "AND" or "OR".  Set AND_OR_OUTSTANDING.  Set
97173.              LOCAL_FALLTHRU_CONDITION according to current operation (AND or OR).
97174.              Continue scan over operands until simple conditional is found which
97175.              is target for LOCAL_BRANCH_LABEL of current simple conditional;  then
97176.              generate LABEL and set LOCAL_LABEL_REQD for target test.  If no target
97177.              test found set LOCAL_BRANCH_LABEL to either (a) the ULTIMATE_FALLTHRU_
97178.              LABEL (also setting FALLTHRU_LABEL_USED) or to (b) the ULTIMATE_
97179.              BRANCH_LABEL, depending on operation. */
97180.      char
97181.          LOOKFOR  /* Operation ("AND" or "OR") opposite of SYSLIST(I). */
97182.      int
97183.          MAX_DEPTH,  /* The maximum depth at which the LOOKFOR'ed
97184.                  operation is a possible branch target. */
97185.          TARGET  /* Simple conditional number which is the target for
97186.                  the branch. */
```

```
97188.       AND_OR_OUTSTANDING := true
97189.       if SYSLIST(I) = 'AND'
97190.         then
97191.           LOCAL_FALLTHRU_CONDITION := true
97192.           LOOKFOR := 'OR'
97193.           MAX_DEPTH := LA_DEPTH
97194.         else /* Operation is OR. */
97195.           LOCAL_FALLTHRU_CONDITION := false
97196.           if LA_DEPTH = 0
97197.             then
97198.               I := LAST_INDEX + 1
97199.                   /* Advance I to force skip of unnecessary search. */
97200.             fi
97201.           LOOKFOR := 'AND'
97202.           MAX_DEPTH := LA_DEPTH - 1
97203.         fi
97204.       I := I + 1
97205.       TARGET := COND_COUNT + 1
97206.       while I ≤ LAST_INDEX and
97207.                 (SYSLIST(I) ≠ LOOKFOR or LA_DEPTH > MAX_DEPTH)
97208.         do
97209.           if SYSLIST(I) = '<' or = '+'
97210.             then
97211.               LA_DEPTH := LA_DEPTH + 1
97212.             else
97213.               if SYSLIST(I) = '>' or = '/'
97214.                 then
97215.                   if LA_DEPTH > 0
97216.                     then
97217.                       LA_DEPTH := LA_DEPTH - 1
97218.                     fi
97219.                 else
97220.                   if SYSLIST(I)[1,1] = '('
97221.                     then
97222.                       TARGET := TARGET + 1
97223.                     else
97224.                       if SYSLIST(I) ≠ 'AND' and ≠ 'OR'
97225.                         then
97226.                           mnote (8, 'STRC9704 SYNTAX ERROR—"' || SYSLIST(I) ||
97227.                                   '" INVALID')
97228.                           ERROR_OCCURRED := true
97229.                           I := LAST_INDEX + 1
97230.                               /* Force break out of loop. */
97231.                         fi
97232.                     fi
97233.                 fi
97234.             fi
97235.           I := I + 1
97236.         od /* (Termination:  I is either incremented or set to LAST_INDEX +
97237.                 1;  LAST_INDEX is fixed;  I would eventually exceed LAST_INDEX even
97238.                 if other tests never occurred.) */
97239.       if I > LAST_INDEX
97240.         then
97241.           if (LOOKFOR = 'OR' /* Operand was AND. */ and
97242.                 ¬ ULTIMATE_FALLTHRU_CONDITION) or
97243.                 (LOOKFOR = 'AND' /* Operand was OR. */ and
97244.                 ULTIMATE_FALLTHRU_CONDITION /* is true */ )
97245.             then
97246.               LOCAL_BRANCH_LABEL := ULTIMATE_FALLTHRU_LABEL
97247.               FALLTHRU_LABEL_USED := true
97248.             else
97249.               LOCAL_BRANCH_LABEL := ULTIMATE_BRANCH_LABEL
97250.             fi
97251.         else
97252.           LOCAL_BRANCH_LABEL := BLOCK_LABEL_PREFIX || UNIQUE_LABEL_ID || TARGET
97253.           LOCAL_LABEL_REQD(TARGET) := true
97254.               /* Note that we are relying on the automatic initialization of the
97255.                   LOCAL_LABEL_REQD array to false at start of every invocation of
97256.                   CONDITIONAL_EXPRESSION_PROCESSOR. */
97257.         fi
97258.     corp
```

```
98001.  macro  TERMINATE_DO_LOOP  (  ;  )
98002.          /*  Called by OD macro to terminate the current DO block loop by
98003.              generating the necessary loop-terminating branches.  If control
98004.              can fall out of the bottom of the code at loop termination, set
98005.              TDL_FALLTHRU_OCCURS to true;  else set false. */
98006.      bit
98007.          LB_LABEL_REQ  /* Indicates whether label required on looping
98008.                  branch instruction. */
98009.      char
98010.          X,  /* Temporary string. */
98011.          LABEL, /* Holds looping branch label. */
98012.          BRANCH8, BRANCH10, BRANCH11, /* One character codes for flow points
98013.                  which are targets for flow points 8, 10, and 11. */
98014.          LB_OPCODE, OPER1, OPER2 /* Opcode and operands of looping branch
98015.                  to be generated. */

98017.      call TRACE_PRINTER(  ;  'TDL')
98018.          /* Extract the following stored by DO macro:  */
98019.      X := INFORMATION(CURRENT_NEST_LEVEL)
98020.      BRANCH8 := X[1,1]
98021.      BRANCH10 := X[2,1]
98022.      BRANCH11 := X[3,1]
98023.      LB_LABEL_REQ := X[4,1]
98024.      OPER1 := OPERAND1(CURRENT_NEST_LEVEL)
98025.      OPER2 := OPERAND2(CURRENT_NEST_LEVEL)
98026.      LB_OPCODE := OPERAND3(CURRENT_NEST_LEVEL)
98027.      BLOCK_LABEL_PREFIX := '$' || BLOCK_NUMBER(CURRENT_NEST_LEVEL)
98028.      if LB_LABEL_REQ
98029.        then
98030.          LABEL := BLOCK_LABEL_PREFIX || 'LPB')
98031.        fi
98032.          /* Generate all looping instructions. */
98033.      docase BRANCH8 only
98034.        of
98035.          case ('W', 'U')
98036.            generate ('       B     ' || BLOCK_LABEL_PREFIX || BRANCH8 || '1')
98037.          esac
98038.          case ('B')
98039.            generate ('       B     ' || BLOCK_LABEL_PREFIX || 'BEG')
98040.          esac
98041.          case ('L')
98042.                /* Nothing to generate;  fall through to looping branch. */
98043.          esac
98044.        esacod
98045.      TDL_FALLTHRU_OCCURS := false  /* Assume we will always branch. */
98046.      if LB_OPCODE ≠ ''
98047.        then /* A looping branch is required. */
98048.          TDL_FALLTHRU_OCCURS := true
98049.              /* Mark that we will fall out of the looping branch. */
98050.          if BRANCH10 = 'B'
98051.            then
98052.              X := 'BEG'
98053.            else /* BRANCH10 = 'W' or 'U'. */
98054.              X := BRANCH10 || '1'
98055.            fi
98056.          if LB_OPCODE = 'BCT'
98057.            then
98058.              generate (LABEL || '  BCT   ' || OPER1 || ',' || BLOCK_LABEL_PREFIX ||
98059.                      X)
98060.            else
98061.              generate (LABEL || '  ' || LB_OPCODE || '   ' || OPER1 || ',' ||
98062.                      OPER2 || ',' || BLOCK_LABEL_PREFIX || X)
98063.            fi
98064.          if BRANCH11 ≠ 'W'
98065.            then
98066.                  /* It must be 'W' or 'U';  looping branch fall-through is not to end
98067.                      of loop;  generate branch to proper alternative conditional test. */
98068.              generate ('       B     ' || BLOCK_LABEL_PREFIX || BRANCH11 || '1')
98069.              TDL_FALLTHRU_OCCURS := false
98070.                  /* Branch defeats fall through. */
98071.            fi
98072.        fi
98073.    end
98074.      /* {Lemma:  TERMINATE_DO_LOOP does not modify
98075.              CURRENT_NEST_LEVEL.} */
```

"XHEX" Macro -- 19 December 1973

```
99001.   macro  XHEX  (  ;  )
99002.            /* Converts the integer in HEX_IN to a two-character hex string in
99003.                HEX_OUT. */
99004.       int I, J  /* Temporaries. */

99006.       call TRACE_PRINTER (  ;  'XHEX')
99007.           /* Print macro name "XHEX" in mnote if tracing on. */
99008.       if HEX(1) ≠ '0'
99009.         then /* Hex array must be initialized. */
99010.           HEX(1)  := '0'   HEX(2)  := '1'   HEX(3)  := '2'   HEX(4)  := '3'
99011.           HEX(5)  := '4'   HEX(6)  := '5'   HEX(7)  := '6'   HEX(8)  := '7'
99012.           HEX(9)  := '8'   HEX(10) := '9'   HEX(11) := 'A'   HEX(12) := 'B'
99013.           HEX(13) := 'C'   HEX(14) := 'D'   HEX(15) := 'E'   HEX(16) := 'F'
99014.         fi
99015.       I := HEX_IN/16
99016.       J := HEX_IN - I*16 + 1
99017.       HEX_OUT := HEX(I+1) || HEX(J)
99018.     mend
```

¤DEFINED •Modified *Parameter

CROSS-REFERENCE LISTING

```
CASE_UPDATE_INFO                 33121.  ¤33604.
CEP_FIND_NEXT_CONDITION          97050.  ¤97100.
CEP_LOOKAHEAD                    97059.  ¤97123.
CEP_SCAN_FOR_BRANCH              97158.  ¤97171.
COMMA2                          ¤81024.   81466.  81469.  •81489.  •81510.  ¤83035.  •83194.
                                •83214.   83400.  83405.  83409.
COMP_LABEL                      ¤33043.  •33223.  33253.  33280.   33298.   33301.   33305.
                                •33307.  •33321.  33322.
COMP_LABEL_NO                   ¤33013.  •33192.  33223.  33249.   33258.  •33308.   33308.
                                 33321.  ¤37011.  37144.  37147.  •37195.   37196.   37285.
                                 37288.
COND_COUNT                      *97013.  •97045.  97070.  97072.  •97120.   97120.   97205.
CONDITIONAL_EXPRESSION_PROCESS   11180.   21719.  33507.  ¤97001.
CORP                             43049.  ¤83001.
CORP_DEBUGGING_STORES            83096.  ¤83219.
CORP_FREE_DYNAM_SAVEAREA         83092.  ¤83381.
CORP_GEN_CRP_BCK_AREAS           83130.  ¤83436.
CORP_GEN_EXIT_LABEL              83082.  ¤83488.
CORP_GET_PROC_INFO               83074.  ¤83148.
CORP_RESTORE_DEBUG_ENVIRONMENT   83140.  ¤83418.
CORP_RESTORE_REGISTERS           83117.  ¤83388.
CORP_RESTORE_REG13               83087.  ¤83259.
CORP_SAVE_RETURNING_REGS         83105.  ¤83314.
CORP_SET_RESTORE_RANGE           83079.  ¤83185.
CORP_SET_RETURN_CODE             83101.  ¤83278.
COUNT_SPOT                      ¤81869.  •81922.  •81924.  81932.   81936.
CRP_BCK_OFFSET                  ¤83225.  •83235.  •83237.  83246.   83252.
CURRENT_NEST_LEVEL              ¤  200.   11148.  11215.  11250.   13016.   13029.   13034.
                                 13038.   13039.  13046.  13048.   13049.   15007.   15026.
                                 15028.   15031.  21097.  21828.   21835.   21836.   21837.
                                 21861.   23021.  23034.  23042.   23056.   23059.   23063.
                                 25021.   25034.  25042.  25049.   25065.   25068.   27014.
                                 27030.   27041.  31251.  31266.   31267.   31268.   31269.
                                 31270.   33135.  33139.  33140.   33146.   33147.   33164.
                                 33594.   33609.  35007.  35028.   35030.   37040.   37096.
                                 37097.   37098.  37099.  37100.   37101.   37102.   41049.
                                 43011.   43015.  43019.  53008.   53010.   53014.   55006.
                                 81160.   81377.  81951.  81952.   81953.   81954.   81957.
                                 83052.   83152.  83153.  83154.   83166.   83167.   83174.
                                 83492.   83505.  91012.  91030.   91043.   91044.   92007.
                                 92009.   92011.  92013.  92021.   92023.   92032.   92033.
                                 92034.  •92038.  92038.  •93022.   93022.   93023.   93031.
                                 93032.   93033.  93034.  93035.   93036.   93037.   93038.
                                 93039.   93040.  93043.  93046.   93047.   93057.   93057.
                                 96013.   96018.  96019.  96021.   96022.   96028.   96029.
                                 96044.   96061.  96065.  98019.   98024.   98025.   98026.
                                 98027.
DEBUG                           *81002.   81194.  81196.  81278.   81303.
DEBUG_BLOCKCOUNTS_REQD          ¤  173.   11136.  11220.  11222.   11253.   21099.   21844.
                                 21846.   21864.  31230.  31253.   33568.   33570.   33597.
                                 41033.   41035.  41052.  81187.  •81228.  •81233.  •81276.
                                 81895.   81908.  •83428.
DEBUG_BLOCKNAMES_REQD           ¤  175.   11213.  11220.  11248.   21844.   21859.   31230.
                                 31249.   33568.  33592.  41033.   41047.   81184.  •81202.
                                •81208.  •81275.  •83425.
DEBUG_CORPVALUES_REQD           ¤  177.   81190.  •81254.  •81260.  •81277.   83230.  •83431.
                                 83444.
DEBUG_DEBUGMACROS_REQD          ¤  179.  •81295.  •81300.
DEBUG_LISTBLOCKS_REQD           ¤  181.   81186.  •81220.  •81225.  •81275.  •83427.   92030.
                                 93053.
DEBUG_MACRONAMES_REQD           ¤  184.  •81287.  •81292.  95004.
DEBUG_PROCCOUNTS_REQD           ¤  187.   81149.  81188.  •81236.  •81241.  •81276.   81574.
                                 81583.   81895.  81908.  83238.  •83429.
DEBUG_PROCNAMES_REQD            ¤  189.   81185.  •81211.  •81217.  •81275.   81363.  •83426.
DEBUG_PROCTRACE_REQD            ¤  191.   81148.  81189.  •81245.  •81251.  •81276.   81746.
                                 81809.   81871.  •83430.
DEBUG_SAVETRACE_REQD            ¤  193.   81147.  81191.  •81319.  •81327.  •81333.   81429.
                                 81562.   81711.  81728.  81910.   81939.   83112.   83232.
                                •83432.   83454.  83474.
DEPTH                           *97015.  •97046.  97086.  97094.  •97109.   97109.   97134.
DO                              ¤21001.
DO_BRANCH_FOR_LOOP_ENTRY         21107.  ¤21560.
DO_FIND_END_INDEXES_AND_MAIN_O   21142.  ¤21222.
DO_FIND_KEYWORDS_AND_PRESENCE    21137.  ¤21160.
DO_GENERATE_ALL_CONDITIONAL_TE   21110.  ¤21584.
DO_GENERATE_CONDITIONAL_SET      21610.  ¤21695.
DO_INFO_SAVE                     21116.  ¤21773.
```

¤DEFINED •Modified *Parameter

□DEFINED •Modified *Parameter

CROSS-REFERENCE LISTING

```
HEX_OUT                          □  163.   33583.   33585.   81157.  •99017.
BOLD                             □11121.  •11128.   11139.
I                                □21038.  •21173.   21175.   21177.   21181.   21184.   21187.
                                  21191.   21194.  •21198.   21198.  •21240.   21243.   21244.
                                  21246.  •21249.   21250.   21251.   21253.   21288.   21292.
                                 •21313.   21316.   21319.   21320.  •21333.   21336.   21341.
                                  21342.   21345.  □31192.  •31209.   31212.   31214.  •31215.
                                  31215.  □33015.  •33191.   33195.   33198.   33199.   33200.
                                  33201.   33203.  •33240.   33240.   33280.   33286.   33288.
                                  33292.   33299.   33302.   33310.   33314.   33316.   33327.
                                  33335.   33337.   33341.   33348.   33350.  •33371.   33372.
                                  33376.   33381.   33383.   33388.   33393.  •33423.   33423.
                                 •33469.   33470.  □37185.  •37187.   37188.   37189.   37193.
                                  37195.   37197.  □37238.  •37257.  •37260.   37264.   37266.
                                  37268.  •37272.   37272.  □43007.  •43019.   43022.   43022.
                                 •43024.   43024.   43027.   43034.  □81180.  •81193.   81194.
                                  81196.   81278.   81303.  •81306.   81306.  □81422.  •81464.
                                  81466.  □81540.  •81569.   81570.  □81619.  •81649.   81650.
                                  81652.   81656.  •81660.   81660.  •81688.   81689.   81691.
                                 •81692.   81692.  □81948.  •81955.   81958.  □81966.  •81976.
                                  81979.  •81984.   81984.  □83321.  •83354.   83355.   83359.
                                  83359.   83360.   83365.   83366.   83369.   83370.   83371.
                                 •83374.   83374.  □83391.  •83397.   83398.   83398.  □91006.
                                 •91012.   91015.   91015.   91016.  •91018.   91018.   91022.
                                  91023.   91035.   91037.   91042.   91049.   91049.   91051.
                                  91053.   91055.  □96006.  •96044.   96045.   96045.  •96047.
                                  96047.   96050.   96055.   96061.  *97025.  •97135.   97136.
                                  97136.  •97138.   97138.   97148.   97151.   97156.   97163.
                                  97189.  •97198.  •97204.   97204.   97206.   97207.   97209.
                                  97213.   97220.   97224.   97226.  •97229.  •97235.   97235.
                                  97239.  □99004.  •99015.   99016.   99017.
ID                               *81001.   81362.   81362.   81369.   81396.   81403.   81406.
                                  81408.   81410.
IF                               □11001.
IF_ASYNCH_BRANCH                  11044.  □11076.
IF_BLOCK_COUNT                    11061.  □11206.
IF_CONDITIONAL_GENERATOR          11057.  □11163.
IF_EXIT_LABEL                    □13011.  •13048.   13059.
IF_EXIT_SPECS                     11052.  □11116.
IF_SET_CONDITIONAL_TEST_SPECS     11046.  □11090.
INCR                             □37238.  •37248.  •37250.   37260.   37272.
INDEX                            *31001.   31100.   31105.   31107.   31108.   31128.   31132.
                                  31134.   31166.   31170.   31178.   31180.  *97017.  •97047.
                                  97048.   97074.   97076.   97077.   97078.   97079.   97080.
                                 •97085.   97107.   97107.  •97110.   97110.   97114.   97117.
                                  97135.
INDEX_ADDR                       □31031.  •31155.  •31180.   31267.  □33041.  •33170.   33278.
                                  33298.   33301.   33325.   33347.   33350.  □37030.  •37097.
                                  37207.   37225.   37229.   37240.   37244.   37254.   37276.
INDEX_LENGTH                     □31018.  •31123.  •31139.   31269.  □33046.  •33172.   33299.
                                  33347.
INDEX_RANGE_ASSURED              □31012.  •31092.  •31115.   31271.  □33019.  •33178.   33558.
                                 •33561.   33611.  □37018.  •37101.   37170.
INDEX_REG                        □31016.  •31128.  •31131.   31159.   31161.
INDEX_TYPE                       □31020.  •31129.  •31134.   31137.   31139.  •31140.  •31144.
                                  31156.
INFO                             □23017.  •23034.   23035.   23043.   23045.   23063.   23066.
                                 □25017.  •25034.   25035.   25047.   25050.   25068.   25070.
                                 □27010.  •27030.   27031.   27032.
INFORMATION                      □  221.  •21828.   23034.  •23063.   25034.  •25068.   27030.
                                 •31270.   33173.  •33610.   37099.   37100.   37101.  •81957.
                                  83154.   91043.   91044.   91049.   91051.  •93038.   98019.
INFORMATION_VALUE                *93007.   93038.
INLINE_SAVEAREA                  □81614.  •81624.   81625.   81632.   81664.
INSERT                           □33276.  •33285.  •33290.  •33294.   33299.  •33332.  •33339.
                                 •33343.   33348.
J                                □31192.  •31208.   31209.   31212.  □81619.  •81648.   81657.
                                 •81661.   81661.  •81687.   81691.  •81693.   81693.  □81966.
                                 •81977.   81982.  •81985.   81985.  □99004.  •99016.   99017.
LA_DEPTH                         *97024.  •97134.   97139.  •97141.   97141.   97150.   97193.
                                  97196.   97202.   97207.  •97211.   97211.   97215.  •97217.
                                  97217.
LABEL                            □11018.  •11038.   11064.   11066.   11085.  •11087.   11180.
                                 •11200.  •11202.   11225.  •11226.   11240.  •11243.  •11246.
                                 □21067.  •21089.   21101.  •21102.   21121.   21123.   21568.
                                 •21569.   21573.  •21574.   21578.  •21579.  •21629.   21705.
                                  21711.   21714.  •21717.   21760.   21763.   21765.  •21767.
```

□DEFINED •Modified *Parameter

□DEFINED •Modified *Parameter

CROSS-REFERENCE LISTING

```
NOM                    (cont.)  33179.  □33607.  *33609.  33610.  33613.
MULT                            □81024.  81465.  81488.  *81488.  *81510.  □83035.  *83193.
                                *83214.  83399.  83404.  83408.
MULTIBASE                       □81019.  *81686.  81969.  *81988.
MULTIPLESOP4                    □33032.  *33176.  *33414.  33610.  □37016.  *37100.  37218.
                                 37227.  37246.
NESTING_LIMIT                   □  202.  13016.  15007.  23021.  25021.  27014.  35007.
                                 35028.  37040.  43011.  53010.  55006.  83052.  92007.
                                *93015.  93023.  93025.
NEXT_CASE                       □33044.  *33470.  33476.  33499.
NEXT_COMP_LABEL_NO              □  231.  *31206.  33192.  *33258.  37067.  37195.
NEXT_DEPTH                      *97023.  97086.  *97150.
NEXT_INDEX                      *97021.  97085.  *97148.
NOCASE                          □37031.  *37161.  *37164.  *37167.  37212.  37214.  37217.
                                 37222.  37226.  37230.  37270.
NOT_FIRST_PROC                  □  149.  81138.  *81139.
CD                              □27001.  43040.
OFFSET                          □81043.  *81425.  81464.  81469.  *81551.  *81567.  *81577.
                                *81580.  *81586.  *81589.  81591.  *81603.  *81606.  81606.
                                □83321.  *83343.  83345.  *83359.  83361.  □83391.  *83398.
                                 83400.
OFFSET_TO_GPR0                  □81045.  *81425.  *81551.  *81567.  *81577.  *81580.  *81591.
                                *81594.  81594.  81603.  81955.  □83016.  *83155.  83156.
                                 83158.  83161.  83343.  83359.  83398.
OLD_EXIT                        *92001.  92013.  92025.  92027.
ONEXIT                          □25001.
ONEXIT_GENNED                   □25009.  *25035.  25037.
OP                              □33364.  *33376.  33377.  33377.  33391.  33394.  33395.
                                 33398.  33404.  33406.  33407.  33408.  33412.  33412.
                                *33416.  33416.
OP_CODE                         *94002.  94028.  94032.  94035.  94039.  94043.
OP_COUNT                        □21385.  *21394.  *21396.  21404.  □33496.  *94009.  94022.
                                 94075.  94080.  94082.  94084.  94086.  94088.  *97019.
                                *97074.  97083.
OPERAND_FORMAT                  □21036.  21094.  21105.  *21452.  *21454.  *21457.  *21464.
                                *21466.  *21469.  *21477.  *21479.  *21482.  *21498.  *21500.
                                *21505.  *21507.  *21513.  *21515.  *21523.  *21525.  *21528.
                                *21544.  *21546.  *21549.  *21552.  *21555.  21564.  21655.
                                 21745.  21757.  21785.  21802.  21815.
OPERAND1                        □  217.  13034.  13038.  15026.  15028.  *21835.  *31266.
                                 33165.  *33613.  37096.  *81951.  83152.  *93034.  98024.
OPERAND1_VALUE                   11024.  *93003.  93034.
OPERAND2                        □  217.  13029.  *21836.  *31267.  33170.  37097.  *81952.
                                 83153.  *93035.  98025.
OPERAND2_VALUE                   13058.  *93004.  93035.
OPERAND3                        □  217.  15031.  *21837.  *31268.  33171.  37098.  *81953.
                                 83166.  *93036.  98026.
OPERAND3_VALUE                   13059.  *93005.  93036.
OPERAND4                        □  217.  *31269.  33172.  *81954.  83167.  *93037.
OPERAND4_VALUE                  *93006.  93037.
OPER1                           *94003.  94032.  94035.  94039.  □98014.  *98024.  98058.
                                 98061.
OPER2                           *94004.  94036.  94040.  □98014.  *98025.  98062.
OPER3                           *94005.  94040.
OPER4                           *94006.
OPTION                          *31001.  31086.  31088.  31090.  31094.  31096.
OS_LINKAGE                      □81014.  *81114.  81129.  81316.  81362.  81373.  81384.
                                 81401.  81542.  81621.  81625.  81632.  81666.  81910.
                                 81939.  81958.  □83028.  83085.  83112.  83119.  *83163.
                                 83286.  83325.  83444.
OS_POINTER                      □81029.  81466.  81470.  *81544.  *81560.  □83037.  *83327.
                                *83330.  83345.  83361.  83367.  83371.  83400.  83406.
                                 83411.
PASS                            □21587.  *21589.  21590.  21592.  21619.  *21633.  21633.
PCT_GENNED_WITH_VECTOR          □81867.  *81901.  81912.
POP_OLD_BLOCK                    13052.  15009.  15031.  27045.  35024.  37084.  43013.
                                 43053.  83054.  83137.  □92001.
PREV_SAVETRACE_AREA             □  164.  53030.  *81736.  81854.  *81858.  83465.  *83484.
PREV_SAVETRACE_PTR              □  166.  53025.  53031.  *81735.  81852.  *81856.  83463.
                                *83483.
PREVIOUS_DEBUG_VECTOR           □81031.  *81183.  *81338.  81954.  □83046.  *83167.  83423.
                                 83425.  83426.  83427.  83428.  83429.  83430.  83431.
                                 83432.
PROC                            □81001.
PROC_COUNTER                    □  137.  81150.  *81153.  81153.  81155.
PROC_DEBUG_SET                   81144.  □81167.
PROC_DEBUG_STUFF                 81084.  □81865.
```

□DEFINED *Modified *Parameter

□DEFINED •Modified *Parameter

CROSS-REFERENCE LISTING

TARGET	(cont.)	83449.	83450.	□97185.	•97205.	•97222.	97222.	97252.
		97253.						
TDL_FALLTHRU_OCCURS		□ 146.	25055.	•98045.	•98048.	•98069.		
TDL_GENNED		□23011.	•23043.	23047.	□25011.	•25050.	25052.	□27005.
		•27031.	27033.					
TERMINATE_DO_LOOP		23049.	25054.	27035.	□98001.			
THIS_CONDITIONAL_REQD		□21052.	21608.	•21643.	•21680.			
TRACE_PRINTER		11020.	13014.	15005.	21075.	23019.	25019.	27012.
		31040.	33048.	35005.	37038.	41007.	43009.	53006.
		55004.	81051.	83050.	91008.	92005.	93013.	94016.
		□95001.	96008.	97041.	98017.	99006.		
TRACE_VECTOR_GENNED		□ 153.	81873.	•81894.				
ULTIMATE_BRANCH_LABEL		□ 260.	•11106.	11128.	11138.	•11139.	•11154.	•21648.
		•21659.	•21665.	•21685.	•33499.	55016.	•91051.	•91053.
		97153.	97249.					
ULTIMATE_FALLTHRU_CONDITION		□ 253.	•11111.	•11147.	•21652.	•21658.	•21689.	•33501.
		97154.	97242.	97244.				
ULTIMATE_FALLTHRU_LABEL		□ 263.	•11109.	•11146.	11200.	21629.	•21650.	•21662.
		•21666.	•21669.	•21687.	21748.	•33500.	97246.	
UNEXPECTED_OPERANDS_FOUND		□33034.	•33177.	•33226.	•33410.	33451.	33611.	
UNIQUE_LABEL_ID		□ 269.	•11177.	•21653.	•21690.	•33506.	97072.	97252.
UNTIL_COND_TEST		□21050.	•21319.	•21324.	•21328.	21412.	21492.	21540.
		21643.	21644.					
UNTIL_END_INDEX		□21035.	•21231.	•21243.	•21246.	21265.	21280.	21319.
		•21327.	21647.					
UNTIL_INDEX		□21035.	•21166.	21189.	•21191.	21202.	21214.	21216.
		21241.	21249.	21265.	21280.	•21313.	21313.	•21327.
		•21427.	21646.					
UNTIL_PRESENT		□21045.	•21214.	21217.	•21234.	21238.	•21270.	21280.
		•21282.	21285.	21311.	21448.	21460.	21473.	21519.
		21536.						
USER_NAME		*11001.	11022.	11038.	*13001.	13034.	13054.	13065.
		13067.	*15001.	15012.	*21001.	21077.	21089.	*23001.
		23025.	*25001.	25025.	*27001.	27016.	*31001.	31042.
		31052.	*33001.	33056.	33059.	33075.	33077.	*35001.
		35009.	*37001.	37042.	*41001.	41009.	41017.	*43001.
		43020.	43022.	43029.	43037.	43040.	43043.	43046.
		43049.	*55001.	55016.	*81001.	81053.	81061.	81371.
		81381.	81382.	*83001.	83071.			
USING13		□81018.	•81637.	81679.	81742.	81804.		
VALID_EXIT		□11012.	•11040.	11050.	11068.	11080.	•11083.	•11126.
VERIFY_END		13020.	15012.	23025.	25025.	27016.	35009.	37042.
		83057.	□96001.					
WHILE_COND_TEST		□21050.	•21341.	•21358.	•21363.	•21367.	21411.	21444.
		21680.	21681.					
WHILE_END_INDEX		□21034.	•21231.	•21250.	•21253.	21256.	21274.	21341.
		•21360.	•21366.	21684.				
WHILE_INDEX		□21031.	•21166.	21179.	•21181.	21202.	21215.	21240.
		21241.	21256.	21274.	•21333.	21333.	•21360.	•21366.
		•21425.	21683.					
WHILE_PRESENT		□21045.	•21215.	•21234.	21236.	•21263.	•21276.	21285.
		21331.	•21358.	21440.				
WORK		*81001.	81086.	81124.	81638.	81749.	81771.	81778.
		81794.	81820.	81875.	81926.			
WORKREG		□81041.	81088.	•81124.	81125.	•81131.	•81133.	81634.
		81640.	81753.	81754.	81766.	81773.	81776.	81777.
		81796.	81801.	81822.	81826.	81877.	81880.	81881.
		81882.	81883.	81928.	81931.	81934.	81934.	81935.
WORKREG_USED		□81021.	81086.	81638.	•81643.	81749.	•81755.	•81788.
		81794.	•81798.	81820.	•81825.	81875.	•81885.	81926.
		•81937.						
X		□31228.	•31241.	•31243.	31245.	□33161.	•33173.	33174.
		33175.	33176.	33177.	33178.	□81763.	•81765.	•81803.
		81811.	81815.	□81967.	•81978.	81981.	•81983.	□83150.
		•83154.	83155.	83163.	83164.	83165.	83169.	83170.
		□98010.	•98019.	98020.	98021.	98022.	98023.	•98052.
		•98054.	98059.	98062.				
XHEX		33582.	81156.	□99001.				

□DEFINED •Modified *Parameter

APPENDIX D

DIAGNOSTIC MESSAGES

The messages generated by the STRCMACS are described below. Each message has an identifying number.

Severity	Message-Number	Message-Text Explanation
8	STRC1101	EXIT= IGNORED WITH ASYNCH Both the EXIT= and ASYNCH operands were specified, but are mutually exclusive; the EXIT= operand has been ignored.
8	STRC1102	REL= OR MASK= NOT IN PARENTHESES—IGNORED The REL= and MASK= operands must be part of a simple conditional and thus must be inside parentheses. The keyword has been ignored.
8	STRC1103	EXIT= IGNORED WITH ELSE= The EXIT= and ELSE= operands were both specified but are mutually exclusive. The EXIT= operand has been ignored.
8	STRC1301	ELSE=*name1* SPECIFIED ON IF BLOCK *name2* The current IF block (whose name is *name2*) included ELSE=*name1* as an operand, but a different (or no) name appears in the label field of this ELSE macro. The discrepancy is ignored.
8	STRC1302	ELSE HAS ALREADY BEEN GENERATED FOR CURRENT IF An ELSE macro has already occurred in the current IF block. The macro is ignored.

Severity	Message-Number	Message-Text Explanation
8	STRC1501	ELSE BLOCK *elsename* NOT FOUND The operand ELSE=*elsename* was coded on the IF macro, but the FI has occurred before the ELSE occurred.
8	STRC2101	OPERANDS AFTER SECOND "WHILE" IGNORED The keyword "WHILE" appears more than once in the DO's operands.
8	STRC2102	OPERANDS AFTER SECOND "UNTIL" IGNORED The keyword "UNTIL" appears more than once in the DO's operands.
4	STRC2103	WARNING—"WHILE,(BCT,..." WILL LOOP ONE LESS TIME THAN VALUE IN REGISTER The looping branch BCT was coded in the WHILE looping group. Since the BCT is executed before the loop, the loop will occur one time fewer than the initial value in the register.
4	STRC2104	WARNING—LOOPING BRANCH MAY NOT BE EXECUTED ON EVERY ITERATION A looping branch is present in the WHILE looping group and the UNTIL looping group is also present. The two looping groups are connected by "OR". If loop execution is to be continued on the basis of the UNTIL group, the WHILE group will not be executed. Hence the indexing register of the looping branch will not be bumped in such cases.
8	STRC2105	TWO LOOPING BRANCHES INVALID IN "DO"—"WHILE" IGNORED Both the WHILE and UNTIL looping groups contain looping branches (BCTs, BXHs, or BXLE,); the WHILE looping group has been ignored.

Severity	Message-Number	Message-Text Explanation
8	STRC2106	INVALID NUMBER OF OPERANDS FOR *opcode* The looping branch *opcode* has the wrong number of operands; BCT should have one, BXLE or BXH should have two.
8	STRC2107	*xxx* INVALID AFTER LOOPING BRANCH—"AND" INSERTED The operand following the looping branch must be "AND" or "OR"; *xxx* was found.
8	STRC2108	FIRST OPERAND MUST BE "WHILE", "UNTIL", "FOREVER", OR OMITTED The first operand of the DO macro is invalid. Either WHILE or UNTIL has been inserted, depending on the remaining operands.
8	STRC2109	WHILE TEST IS VOID—IGNORED No looping group follows the keyword "WHILE"; the keyword has been discarded.
8	STRC2110	LOGIC OPERATOR BETWEEN "WHILE" AND "UNTIL" OMITTED—"AND" ASSUMED No logic operator occurs between the WHILE and UNTIL looping groups. An "AND" has been inserted.
8	STRC2111	UNTIL TEST IS VOID—IGNORED No looping group follows the keyword "UNTIL"; the keyword has been discarded.
8	STRC2112	PARENTHESES OMITTED AROUND *opcode* The looping branch *opcode* was not specified as a sublist.

Severity	Message-Number	Message-Text Explanation
8	STRC2113	REL= OR MASK= NOT IN PARENTHESES—IGNORED The REL= and MASK= operands must be a part of a simple conditional and thus must be inside parentheses. The keyword has been ignored.
8	STRC2114	SUPERFLUOUS LOOPING GROUP IGNORED Both WHILE and UNTIL are present, but other operands precede both. Such operands have been ignored.
8	STRC2301	MORE THAN ONE "ATEND" IN BLOCK More than one ATEND macro has been found for the same DO. Only the first is processed.
8	STRC2501	MORE THAN ONE "ONEXIT" IN BLOCK More than one ONEXIT macro has been found for the same DO. Only the first is processed.
8	STRC2502	NO EXIT FOR THIS "DO" No EXIT has occurred specifying the DO block for which an ONEXIT is being generated. The segment is dead code.
4	STRC3101	WARNING—*xxx* ASSUMED AS INDEX: USE "DOCASE,*xxx*" FOR RANGE SPEC *xxx* is either "IFANY" or "ONLY" and appears as the first operand. As such, it is assumed to be the address of the DOCASE index. If the range specification is intended, *xxx* must be the second or third operand.
8	STRC3102	*xxx* INVALID SECOND OPERAND—IGNORED The second operand of DOCASE may only be "SPARSE", "SIMPLE", "IFANY", "ONLY", or omitted. *xxx* was found.

Severity	Message-Number	Message-Text Explanation
8	STRC3103	*xxx* INVALID THIRD OPERAND—IGNORED The third operand of DOCASE may only be "IFANY", "ONLY", or omitted. *xxx* was found.
12	STRC3104	GENERAL/SPARSE/CHARCOMP DOCASE NESTING LEVEL *nestlev* EXCEEDS MAXIMUM OF *maxlev*—MACROS MUST BE MODIFIED The number of nesting levels of DOCASE macros (other than SIMPLE or conditional test type DOCASEs) is *nestlev*; but the internal stack limits such nesting to *maxlev*. Either the program or the macros must be modified.
8	STRC3301	"CASE" NOT IMMEDIATE DAUGHTER OF "DOCASE" The CASE macro is not immediately surrounded by a DOCASE macro. If one or two BLENDs are required, they will be inserted and message 3302 or 3303 will be issued; otherwise, message 3304 will follow.
8	STRC3302	ASSUMING "BLEND" OMITTED—INSERTED Preceded by message 3301. Since the second containing block is a DOCASE, one BLEND is inserted to get to it.
8	STRC3303	ASSUMING TWO "BLENDS" OMITTED—INSERTED Preceded by message 3301. Since the third containing block is a DOCASE, two BLENDs are inserted to get to it.
8	STRC3304	"CASE" TREATED AS "BLOCK" MACRO Preceded by message 3301. No fix-up was possible. The CASE is converted to a BLOCK.

Severity	Message-Number	Message-Text Explanation
8	STRC3305	*xxx* INVALID—*yyy* ASSUMED *xxx* appears as the operand of a CASE macro, but the format is invalid. Usually this occurs when a range is coded whose second value is less than the first (such as "(15,10)"). The operand *yyy* replaces *xxx*.
4	STRC3306	EARLIER UNEXPECTED OPERAND IMPLIES THIS TO BE CASE *xxx* CASE macro has no operands but earlier CASEs for the same DOCASE contained operands other than their ordinal position numbers. The operand *xxx* has been assumed.
8	STRC3307	OPERAND INVALID VALUE ON SIMPLE CASE *xxx* The ordinal position number of this CASE is *xxx*, but SIMPLE was coded on the DOCASE and an operand other than *xxx* on the CASE. The operand is ignored.
8	STRC3308	"DOCASE...,ONLY" INVALID WITH MISC A CASE MISC has been found in a DOCASE with the ONLY range option. Since these are mutually exclusive, the ONLY has been ignored.
8	STRC3309	OPERAND MUST BE SELF-DEFINING TERM OR OMITTED ON SIMPLE CASE *xxx* The ordinal position number of this CASE is *xxx* and SIMPLE was coded on the DOCASE, but an operand has been specified which is not a self-defining term. It has been ignored.
8	STRC3310	REL= OR MASK= NOT IN PARENTHESES—IGNORED The REL= and MASK= operands must be a part of a simple conditional and thus must be inside parentheses. The keyword has been ignored.

Severity	Message-Number	Message-Text Explanation
8	STRC3311	MULTIPLE MISC CASES IN THIS DOCASE—THIS BLOCK IS DEAD CODE More than one "CASE MISC" has occurred in the same DOCASE. Only the first is executable.
8	STRC3312	*xxx* INVALID—ONLY FIRST TWO SUBOPERANDS PROCESSED *xxx* is an operand list containing more than two sub-operands in a CASE macro for a DOCASE which specified an index. Operands after the first two are ignored.
*	STRC3313	CASE DEBUG ID=X'*hh*' Debug block counts are being kept for this CASE macro. When executed, this block will store X'*hh*' into the last-case variable in the immediately surrounding DOCASE block.
8	STRC3701	DOCASE CONTAINS NO VALID CASES No valid CASE macros were found as immediate sub-blocks of the DOCASE.
8	STRC4301	NO BLOCKS ACTIVE—"BLEND" IGNORED A BLEND macro was coded but no blocks were active (the current nest level was zero). The macro has been ignored.
8	STRC4302	NO BLOCK ACTIVE NAMED *xxx*— "BLEND" IGNORED A BLEND macro was issued for block *xxx*, but no block by that name is active.
8	STRC5301	BLEND OF *blname* ASSUMED The block *blname* has not yet been terminated; a BLEND is being issued for it by the FINAL macro.

Severity	Message-Number	Message-Text Explanation
8	STRC8101	LINKAGE=*xxx* INVALID—"OS" ASSUMED The first suboperand of the LINKAGE= keyword is invalid; "OS" has been substituted.
8	STRC8102	SECOND LINKAGE OPERAND IGNORED The second suboperand of the LINKAGE= keyword must be either "CSECT" or omitted. It is invalid and has been ignored.
4	STRC8103	WARNING—SAVETRACE REQUIRES "FINAL" MACRO The SAVETRACE debug option has been specified on the first PROC; warning is printed to indicate need for FINAL macro.
8	STRC8104	DEBUG=*xxx* INVALID—IGNORED An invalid debug specification is present; the first eight characters of the invalid operand are listed as *xxx*. That option is ignored.
8	STRC8105	SAVETRACE MUST BE SPECIFIED ON FIRST PROC The SAVETRACE debug option must be enabled on the first PROC. The operand has been ignored.
8	STRC8106	SAVETRACE REQUIRES FIRST PROC TO BE LINKAGE=OS The SAVETRACE debug option is valid only if the first PROC includes the LINKAGE=OS specification. The operand has been ignored.
4	STRC8107	REG 1 MUST BE AMONG THOSE SAVED Register 1 was destroyed during the GETMAIN for a dynamic save area and registers 14 through 12 were not specified (or defaulted) as being saved and WORK=NONE was specified. No further check is made to assure register one was among those saved,

Severity	Message-Number	Message-Text Explanation
4	STRC8107 (cont.)	but the restore has been issued. If register 1 was not among those saved, its value will be undefined.
*	STRC8108	PROC *proc-name*, DEBUG ID=X'*hh*' The save-trace, proc-trace, or proc-count debug option has been specified for this proc. The hex id byte *hh* will be used to identify this proc in the labels and dumps. This message is generated so that the user will know the proc id number even if "PRINT NOGEN" has been specified.
8	STRC8109	REGISTER 13 IS INVALID—IGNORED Register 13 was specified as a base register other than as the first register for an OS proc using an in-line save area. The operand has been ignored.
8	STRC8301	NO REGISTERS SAVED—RESTORE IGNORED RESTORE= was coded on the CORP macro, but SAVE=NONE was coded on the PROC. The operand is ignored.
4	STRC8302	WARNING—NO CHECK MADE TO INSURE RETURNING REGISTERS ARE AMONG THOSE SAVED IN TRUNCATED SAVE AREA The first operand on the PROC macro was a decimal integer other than 14. As a result, a small (truncated) save area was created. No check has been made to insure that the registers specified by the RETURN= operand will fit in the save area. If the returning registers are a subset of the RESTORE= registers and they, in turn, form a subsequence of the saved registers, the proper code will be generated.
0	STRC8303	ONE OR MORE EXIT'S REFERENCE THIS POINT This CORP is the target of one or more EXIT macros (or EXIT= operands of IF macros). The severity code of this message may be modified by specifying the EXIT= operand of a PROC macro.

Severity	Message-Number	Message-Text Explanation
8	STRC9103	EXIT TO IMMEDIATELY SURROUNDING BLOCK INVALID The block name specified as the EXIT target is the block immediately surrounding the EXIT.
8	STRC9104	EXIT TO DO BLOCK INVALID WITHIN ATEND OR ONEXIT An EXIT macro specifies (or implies) a DO block as its target, but the EXIT macro is nested within the ATEND or ONEXIT segment of the DO. The macro is ignored.
0	STRC9201	ONE OR MORE EXIT'S REFERENCE THIS POINT This block is the target of one or more EXIT macros (or EXIT= operands of IF macros). The severity code of this message may be modified by specifying the EXIT= operand of a PROC macro.
12	STRC9301	BLOCK NESTING LIMIT OF *limit* EXCEEDED—MACROS MUST BE MODIFIED The current static nesting level has exceeded the stack limit in the macros of *limit*. Either the program or macros must be modified.
8	STRC9302	NON-CASE BLOCK IMMEDIATELY SURROUNDED BY DOCASE INVALID The block being defined is not a CASE block but is immediately surrounded by a DOCASE block. The result is undefined.
8	STRC9401	INSUFFICIENT OPERANDS FOR TEST "*opcode*" The parenthesized list which is supposed to be a simple conditional contains two items, the operation code *opcode* and a bc-spec.

Severity	Message-Number	Message-Text Explanation
8	STRC9402	SUPERFLUOUS OPERANDS FOR TEST "*opcode*" The parenthesized list which is supposed to be a simple conditional contains more than 5 items.
8	STRC9403	NO CONDITION SPECIFIED—"MASK=0" ASSUMED A void simple conditional has been specified.
8	STRC9601	ONE BLEND ASSUMED TO GET TO "*type*" BLOCK In a block terminating macro (such as FI), the current block was not of the corresponding type (such as IF), but the surrounding block is of the proper type. One BLEND has been inserted.
8	STRC9602	TWO BLENDS ASSUMED TO GET TO "*type*" BLOCK In a block terminating macro (such as FI), the current block was not of the corresponding type (such as IF), but the second surrounding block is of the proper type. Two BLENDs have been inserted.
8	STRC9603	CURRENT BLOCK IS NOT "*type*" BLOCK—MACRO IGNORED In a block terminating macro (such as FI), the current block was not of the corresponding type (such as IF).
8	STRC9604	NO ACTIVE BLOCK NAMED "*blname*" The request to terminate block *blname* has been ignored because no block named *blname* is in the nest.

Severity	Message-Number	Message-Text Explanation
8	STRC9605	BLOCK "*blname*" IS NOT A *type* BLOCK—MACRO IGNORED The request to terminate a block named *blname* of type *type* has been ignored because the indicated block is of a different type.
8	STRC9606	END OF BLOCK "*blname1*" IMPLIES END OF BLOCK "*blname2*" Request to terminate block *blname1* must first terminate block *blname2* which is nested inside block *blname1*.
8	STRC9607	NO BLOCKS ACTIVE—MACRO IGNORED The request to terminate a block has been ignored since no blocks are outstanding.
8	STRC9701	INSUFFICIENT OPERANDS The conditional expression ends with a logical operation (AND or OR) outstanding.
8	STRC9702	INSUFFICIENT BRACKETS More left brackets ("<" or "+") than right brackets (">" or "/") are present.
8	STRC9703	SYNTAX ERROR—LOOKING FOR "AND" OR "OR", FOUND *xxx* The operand *xxx* was found where a logical operator was expected.
8	STRC9704	SYNTAX ERROR—*xxx* INVALID Invalid operand *xxx* in conditional expression.
8	STRC9705	SUPERFLUOUS BRACKET IGNORED More right brackets ("<" or "/") have been found in the conditional expression than left brackets (">" or "+").

Severity	Message-Number	Message-Text Explanation
8	STRC9706	SYNTAX ERROR—LOOKING FOR SIMPLE CONDITIONAL, FOUND "*xxx*" The operand *xxx* was found in a conditional expression where a simple conditional sublist was expected.
*	STRC9902	START OF BLOCK *nn* (*blname*) AT DEPTH *level* In response to the debug option LISTBLOCKS, the message indicates the start of the block whose sequential number is *nn*. The block name is *blname*; if no name was specified on the block initiation macro, *blname* is an internal name of the form BLK*nn*. The current static nest level is *level* .
*	STRC9903	END OF BLOCK *nn* (*blname*) AT DEPTH *level* In response to the debug option LISTBLOCKS, the message indicates the end of the block whose sequential number is *nn*. The block name is *blname*; if no name was specified on the block initiation macro, *blname* is an internal name of the form BLK*nn*. The current static nest level is *level* .

APPENDIX E

INSTALLING THE STRCMACS

The structured macros are available to any interested parties. They may be obtained by writing to:

C. Wrandle Barth
Code 603
Goddard Space Flight Center
Greenbelt, Maryland 20771

The normal distribution medium is a 9-track, 800 bpi unlabeled distribution tape reel (DTR). It contains four data sets.

	Data Set	BLKSIZE	LRECL	RECFM
(1)	PROSE	2000	80	FB
(2)	STRCMACS	2000	80	FB
(3)	LISTINGS	2000	137	VBA
(4)	RUFDRAFT	2000	80	FB

The first data set will contain any special instructions for installing the STRCMACS. It will also include any known restrictions, changes, or extensions to the macros as described in this document.

The second data set is the IEBUPDTE source for adding the STRCMACS to a macro library. Each macro is preceded by a "./ ADD" card and the last record is a "./ ENDUP" card.

The third data set is the current listing of the SIMPL-M source for the STRCMACS (printed here as Appendix C).

The fourth data set is the assembly language source of a program called RUFDRAFT. It is provided for those installations which do not have a TN (upper and lower case) print chain available. RUFDRAFT will translate the SIMPL-M listings of data set three for printing on HN, PN, or QN print trains. For instructions on using RUFDRAFT, see the comments in the beginning of the source.

When requesting a copy of the STRCMACS, it would be appreciated if you would enclose a tape—our supply of DTR's is limited.

Any comments, suggestions, or criticisms of the macros will be greatly appreciated.

INDEX