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

PRICES SUBJECT TO CHANGE

C. WRANDLE BARTH

GODDARD SPACE FLIGHT CENTER
GREENBELT, MARYLAND
NOTICE

THIS DOCUMENT HAS BEEN REPRODUCED FROM THE BEST COPY FURNISHED US BY THE SPONSORING AGENCY. ALTHOUGH IT IS RECORGNIZED THAT CERTAIN PORTIONS ARE ILLEGIBLE, IT IS BEING RELEASED IN THE INTEREST OF MAKING AVAILABLE AS MUCH INFORMATION AS POSSIBLE.
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

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>GOTO-LESS PROGRAMMING</td>
<td>2</td>
</tr>
<tr>
<td>STEPWISE REFINEMENT</td>
<td>11</td>
</tr>
<tr>
<td>EFFICIENCY OF STRUCTURED CODE</td>
<td>14</td>
</tr>
<tr>
<td>INTRODUCTION TO STRCMACS</td>
<td>16</td>
</tr>
<tr>
<td>Defining blocks</td>
<td>16</td>
</tr>
<tr>
<td>Decision making</td>
<td>18</td>
</tr>
<tr>
<td>Iteration</td>
<td>21</td>
</tr>
<tr>
<td>Multiple decisions</td>
<td>23</td>
</tr>
<tr>
<td>Abnormal block exit</td>
<td>27</td>
</tr>
<tr>
<td>Defining modules</td>
<td>30</td>
</tr>
<tr>
<td>Special services</td>
<td>38</td>
</tr>
<tr>
<td>STRCMACS debugging aids</td>
<td>38</td>
</tr>
<tr>
<td>Addressability, labels, and reentrant code</td>
<td>42</td>
</tr>
</tbody>
</table>

## APPENDIX A - MACRO INSTRUCTION DESCRIPTIONS

<table>
<thead>
<tr>
<th>Macro Instruction</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conditional expressions</td>
<td>A-1</td>
</tr>
<tr>
<td>ATEND</td>
<td>A-3</td>
</tr>
<tr>
<td>ATEXIT</td>
<td>A-4</td>
</tr>
<tr>
<td>BLEND</td>
<td>A-5</td>
</tr>
<tr>
<td>BLOCK</td>
<td>A-6</td>
</tr>
<tr>
<td>CASE</td>
<td>A-7</td>
</tr>
<tr>
<td>CASEEND</td>
<td>A-9</td>
</tr>
<tr>
<td>CORP</td>
<td>A-10</td>
</tr>
<tr>
<td>DO</td>
<td>A-12</td>
</tr>
<tr>
<td>DOCASE</td>
<td>A-14</td>
</tr>
<tr>
<td>DOCASEND</td>
<td>A-16</td>
</tr>
<tr>
<td>DOEND</td>
<td>A-17</td>
</tr>
<tr>
<td>ELSE</td>
<td>A-18</td>
</tr>
<tr>
<td>ESAC</td>
<td>A-19</td>
</tr>
<tr>
<td>ESACOD</td>
<td>A-20</td>
</tr>
<tr>
<td>EXIT</td>
<td>A-21</td>
</tr>
<tr>
<td>FI</td>
<td>A-22</td>
</tr>
<tr>
<td>FINAL</td>
<td>A-23</td>
</tr>
</tbody>
</table>

Preceding page blank
CONTENTS (continued)

<table>
<thead>
<tr>
<th>Command</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>A-24</td>
</tr>
<tr>
<td>IFEND</td>
<td>A-26</td>
</tr>
<tr>
<td>OD</td>
<td>A-27</td>
</tr>
<tr>
<td>ONEND</td>
<td>A-28</td>
</tr>
<tr>
<td>ONEXIT</td>
<td>A-29</td>
</tr>
<tr>
<td>PROC</td>
<td>A-30</td>
</tr>
<tr>
<td>PROCEND</td>
<td>A-35</td>
</tr>
</tbody>
</table>

APPENDIX B — INTRODUCTION TO ABSTRACT SOURCE LISTING OF STRCMACS ........ B-1

Introduction to SIMPL-M ................. B-1
DO Macro Decision Table ................. B-5

APPENDIX C — ABSTRACT SOURCE LISTING OF STRCMACS ... C-1

APPENDIX D — DIAGNOSTIC MESSAGES ........ D-1

APPENDIX E — INSTALLING THE STRCMACS ....... E-1
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

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 con-
tventional 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 instruc-
tion) in programming languages. While it has been known for some time that it
is theoretically possible to program any problem capable of algorithmic solu-
tion 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 in-
troduced to most programmers, there is understandable skepticism. The sus-
picion 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 under-
stand 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 pro-
grams 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:

```plaintext
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 = 1
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 truly 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.

---

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

```
10 IF (r) GO TO 10
   GO TO 60
20 IF (s) GO TO 80
   GO TO 30
30 IF (t) GO TO 40
   GO TO 30
40 IF (u) GO TO 70
50 IF (v) GO TO 60
60 RETURN
70 GO TO 50
80 IF (w) GO TO 90
90 IF (x) GO TO 110
100 IF (y) GO TO 110
110 IF (w) GO TO 120
120 IF (v) GO TO 60
```

goto-less programming

```
IF q
   THEN
      IF w
         THEN
            WHILE x
               DO
                  OD
               OD
         ELSE
            WHILE t
               DO
                  OD
               OD
      ELSE
            IF v
               THEN
                  FI
               FI
      FI
   FI
IF r
   THEN
      ELSE
      FI
```

5
in goto-less form. Alan Perlis has referred to similar situations as the 'Turing Tarpiț' 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:

1. goto-less programs are easier to understand, debug, and modify.
2. It is easier to prove assertions (in particular, to prove program correctness) about goto-less programs.
3. Goto-less programs are less likely to contain bugs due to their intellectual manageability.
4. Compilers are able to understand, and therefore to optimize, goto-less programs to a larger extent.
5. 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.

\(^2\)ibid. p. 795.
It should be mentioned that the languages in which goto-less is really feasible 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;
      
    end;
  
end;
```

α

β

γ
has the same effect as:

```
        OUTER: begin;
        
        INNER: begin;
        
        if I ≠ 0
        then
        begin
        α
        end;
        
        if I ≠ 0
        then
        begin
        β
        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 extensions 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, ISWIM, BLISS, OREGANO, GEDANKEN, and SIMPL-X. 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 truly 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.

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. Predominant 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 translator 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

---

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" B₁ for some abstract super-machine with arbitrarily complex instructions. Most of the instructions of B₁ 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 B₁, it would surely accomplish the task A. We can now take the instructions of B₁ (call them the A₂₁) and for whichever are not understandable to our real computer repeat the problem-solving process producing a program B₂₁—the "how" for each A₂₁ 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 indentation 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 incompatibilities. 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

---

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 inline 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 manage-ability 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
BNZ end-of-block or LTR 3,3
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:

<table>
<thead>
<tr>
<th>Relation</th>
<th>Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>H or GT</td>
<td>NH or LE (Not High)</td>
</tr>
<tr>
<td>L or LT</td>
<td>NL or GE (Not Low)</td>
</tr>
<tr>
<td>E or EQ</td>
<td>NE (Not Equal)</td>
</tr>
<tr>
<td>O (Ones)</td>
<td>NO (Not Ones)</td>
</tr>
<tr>
<td>P (Plus)</td>
<td>NP (Not Plus)</td>
</tr>
<tr>
<td>M (Minus)</td>
<td>NM (Not Minus)</td>
</tr>
<tr>
<td>Z (Zero)</td>
<td>NZ (Not Zero)</td>
</tr>
</tbody>
</table>

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:

```plaintext
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,

```plaintext
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)
   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: EXH, 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
ADD
CHANGE CASE
  :
  
ESAC
ADD
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 \( i \)th 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 1
A CASE   \( 3,7 \)
  .
  .
  ESAC
B CASE   \( 0,2,8 \)
  .
  .
  ESAC
C CASE   \( 4,(9,13),X'1C' \)
  .
  .
  ESAC
D CASE   \( \text{FIVE, (FOURTEEN, SIXTEEN)} \)
  .
  .
  ESAC
E CASE   \( \)  
  .
  .
  ESAC
ESAC
ESACOD
  .
  .
  .
```
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   } Fullword
or
DOCASE (I,W)
DOCASE (I,H) Halfword
DOCASE (I,B) One byte
DOCASE (3)

or

DOCASE (R3) Register index
```

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 disastrous. ("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>
SCANDO WHILE,(TM,FLAG,ENDOCARD,Z)
:
:
WHOOPIF (CLI,DELIMITR,C',',NE)
<Print "BAD DELIMITER" message>
EXIT
FI
:
:
OD
ODDOINFILE

27
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:

\[\text{IF } (\text{CLI, DELIMITR, C', },', \text{NE}); \text{EXIT=SCAN} \]

or

\[\text{IF } (\text{CLI, DELIMITR, C', },', \text{NE}); \text{EXIT=} \]

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

\[\text{DO FOREVER} \]

or just

\[\text{DO} \]

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
L 2,IDTAB
SEARCH DO WHILE, (CLI, 0(2), X'00', NE)
IF (CLC, ARG(8), 0(2), EQ), EXIT=SEARCH
  A 1,8(2)
  ST 1,8(2)
EXIT UPDATE
FI
L 2,12(2)
OD
MVC 0(8,2), ARG
ST 1,8(2)
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
L 2,IDTAB
SEARCH DO WHILE, (CLI, 0(2), X'00', NE)
IF (CLC, ARG(8), 0(2), EQ), EXIT=SEARCH
  A 1,8(2)
  ST 1,8(2)
ATEND
  MVC 0(8,2), ARG
  ST 1,8(2)
ONEXIT
  A 1,8(2)
  ST 1,8(2)
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.

![Flowchart](image)

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
```

33
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
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)
  .
  .
C PROC DEBUG=(NOPROCTRACE, PROCOUNTS)
  .
  .
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 "$Phxxx" 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 pro-
vided. 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 be-
fore 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 par-
ticular values can be a painstaking and somewhat error-prone process. A final
debug option provides the mechanism for having these areas formatted auto-
matically 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 SAVE-
TRACE 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 conven-
tions. 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 CORPVALUES'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 PROTRACT 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!)

42
TITLE 'EXAMPLE OF DEBUG FACILITIES'

ENTRY PROC LINKAGE=(DS,CSECT), DEBUG=(ALL,LISTBLOCKS,GLOBAL)

: LA 5,2
DO UNTIL,(RCT,3)
  BAL 14,SUBX
  BAL 14,SUBY

: CORP
EJECT

SUBX PROC SAVE=(3,5)
: L 3,XID
: CORP SUBX
EJECT

SUBY PROC DEBUG=NOCORVALYES
: L 3,YID
CALL NEXTPROG
: CORP RETURN=3
EJECT

SUBZ PROC
: L 3,ZID
LR 6,3
IF (C,5,'F1',EQ)
  BAL 14,SUBY
: FI
: CORP RETURN=6
EJECT
FINAL
DS OF
XID DC C'AAAA'
YID DC C'BBBB'
ZID DC C'CCCC'
LTORG
SPACE 3
END
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJECT CODE</th>
<th>ADDR1</th>
<th>ADDR2</th>
<th>STAT</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>000000</td>
<td>4770</td>
<td>F00C</td>
<td>ADDC</td>
<td>2 OURPRG PROC LINKAGE=(O5,CSCE),DEBUG= (ALL,WOLISTBLOCKS,GLOBAL)</td>
</tr>
<tr>
<td>000000</td>
<td>000000</td>
<td>07DE45927D96C7</td>
<td></td>
<td></td>
<td>3, STRING WARNING—SAVETRACE REQUIRES <em>FINAL</em> MACRO</td>
</tr>
<tr>
<td>000000</td>
<td>000000</td>
<td>90BC</td>
<td>D00C</td>
<td></td>
<td>4, STRING PROC GOURPRG, DEBUG ID=1’0151</td>
</tr>
<tr>
<td>000010</td>
<td>000010</td>
<td>4520</td>
<td>F052</td>
<td></td>
<td>** NOTE **</td>
</tr>
<tr>
<td>000020</td>
<td>000020</td>
<td>040007</td>
<td>D000</td>
<td></td>
<td>** *** 3</td>
</tr>
<tr>
<td>000030</td>
<td>000030</td>
<td>5000</td>
<td>F10C</td>
<td></td>
<td>4, STRC8103 WARNING—SAVETRACE REQUIRES <em>FINAL</em> MACRO</td>
</tr>
<tr>
<td>000040</td>
<td>000040</td>
<td>2030</td>
<td>0008</td>
<td>0008</td>
<td>16, A3, ST, 13, A ($LASTSTN)</td>
</tr>
<tr>
<td>000050</td>
<td>000050</td>
<td>5800</td>
<td>P0AC</td>
<td></td>
<td>17, ST, 13, 6(2)</td>
</tr>
<tr>
<td>00006A</td>
<td>00006A</td>
<td>5020</td>
<td>0009</td>
<td></td>
<td>18, LR, 13, 2</td>
</tr>
<tr>
<td>000072</td>
<td>000072</td>
<td>1802</td>
<td></td>
<td></td>
<td>19, USING $P01ISV, 13</td>
</tr>
<tr>
<td>000072</td>
<td>000072</td>
<td>47F0</td>
<td>F05Z</td>
<td></td>
<td>20+ OURPROG CSECT</td>
</tr>
<tr>
<td>000086</td>
<td>000086</td>
<td>0038</td>
<td></td>
<td></td>
<td>21+ USING *, 15</td>
</tr>
<tr>
<td>000096</td>
<td>000096</td>
<td>5020</td>
<td>0009</td>
<td></td>
<td>22+3P01ISV DC A($TRACE),(18-1)F'O'</td>
</tr>
<tr>
<td>0000A4</td>
<td>0000A4</td>
<td>0038</td>
<td></td>
<td></td>
<td>23+ $P01BB DC 2, $P01BB</td>
</tr>
<tr>
<td>0000B4</td>
<td>0000B4</td>
<td>0038</td>
<td></td>
<td></td>
<td>24+STRC8108 PROC OURPROG, DEBUG ID=X'01'</td>
</tr>
<tr>
<td>0000C4</td>
<td>0000C4</td>
<td>47F0</td>
<td>F05Z</td>
<td></td>
<td>** NOTE **</td>
</tr>
<tr>
<td>0000D4</td>
<td>0000D4</td>
<td>0038</td>
<td></td>
<td></td>
<td>** *** 3</td>
</tr>
<tr>
<td>0000E4</td>
<td>0000E4</td>
<td>47F0</td>
<td>F05Z</td>
<td></td>
<td>4, STRC8103 WARNING—SAVETRACE REQUIRES <em>FINAL</em> MACRO</td>
</tr>
<tr>
<td>0000F4</td>
<td>0000F4</td>
<td>0038</td>
<td></td>
<td></td>
<td>4, STRING PROC GOURPRG, DEBUG ID=1'0151</td>
</tr>
</tbody>
</table>

**EXAMPLE OF DEBUG FACILITIES**

LOC OBJECT CODE ADD1 ADD2 STAT SOURCE STATEMENT

---

**NOTE**

**EXAMPLE OF DEBUG FACILITIES**
### EXAMPLE OF DEBUG FACILITIES

<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJECT CODE</th>
<th>ADDR1</th>
<th>ADDR2</th>
<th>STMT</th>
<th>SOURCE</th>
<th>STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>61</td>
<td>SUBI</td>
<td>0912E</td>
<td></td>
<td>PROCI SAVE=(3,5)</td>
<td>00016000</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>*</td>
<td></td>
<td></td>
<td>STBCH0108 PROC SUBX, DEBUG ID=X'02'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>SUBI</td>
<td>B</td>
<td>$P202A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>DC</td>
<td>CL4'SUBX'</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>65</td>
<td>SP02AA</td>
<td>STM 14,12,SP02SV+12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66</td>
<td>B</td>
<td>$P02DD</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>67</td>
<td>SP02SV</td>
<td>DS OP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>68</td>
<td>DC</td>
<td>X'FF020000' FLAG (PF=FINISHED,FE=ENTERED), ID, COUNT</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>69</td>
<td>FIRSTSV</td>
<td>EQU $P02SV</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>70</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>71</td>
<td>DC</td>
<td>A(0,$P02NXT)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>72</td>
<td>SP02DD</td>
<td>L 1,=$P02DD</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>73</td>
<td>MVI</td>
<td>0(256,1),1(1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>74</td>
<td>MVI</td>
<td>256(1),X'02'</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>75</td>
<td>LA</td>
<td>1,$P02SV+2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>76</td>
<td>STH</td>
<td>1,$P02SV+2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>77</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>78</td>
<td>MVI</td>
<td>$P02SV,X'FE'</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>79</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>L</td>
<td>3,XID</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>81</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>82</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>83</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>84</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>85</td>
<td>STM</td>
<td>14,12,SP02CRP+12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>86</td>
<td>MVI</td>
<td>$P02SV,X'FF'</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>87</td>
<td>LM</td>
<td>3,5,$P02SV+32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>88</td>
<td>BR</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>89</td>
<td>$P02CRP</td>
<td>DS OP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>DC</td>
<td>X'FC020000'</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>91</td>
<td>$P02NXT</td>
<td>EQU $P02CRP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>92</td>
<td>DC</td>
<td>A($P02SV,$P02FWD)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>93</td>
<td>DC</td>
<td>15P'O'</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

PC020000

### PAGE 3

23 JAN 74
EXAMPLE OF DEBUG FACILITIES

<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJECT CODE</th>
<th>ADDR1</th>
<th>ADDR2</th>
<th>STMT</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>000208</td>
<td>47F0 D2AC</td>
<td>002C0</td>
<td>97+SUBY</td>
<td>B</td>
<td>$P03AA</td>
</tr>
<tr>
<td>00020C</td>
<td>424C2B8</td>
<td>98+ DC</td>
<td>C'LW'SUBY'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00020C</td>
<td>000C D2C0</td>
<td>032A4</td>
<td>99+$P03AA</td>
<td>STM</td>
<td>14,12,$P03SV+12</td>
</tr>
<tr>
<td>00020C</td>
<td>47F0 D2FC</td>
<td>031A0</td>
<td>100+ B</td>
<td>$P03DD</td>
<td></td>
</tr>
<tr>
<td>00020C</td>
<td>000C D2B8</td>
<td>10C+$P03SV</td>
<td>DS</td>
<td>OF</td>
<td></td>
</tr>
<tr>
<td>00020C</td>
<td>PF300000</td>
<td>10C+ DC</td>
<td>X'PF030000'</td>
<td>FLAG</td>
<td>(PF=FINISHED,FE=ENTERED), ID, COUNT</td>
</tr>
<tr>
<td>00020C</td>
<td>000C D2C0</td>
<td>10C+$P02PW</td>
<td>EQU</td>
<td>$P03SV</td>
<td></td>
</tr>
<tr>
<td>00020C</td>
<td>0000D27000000360</td>
<td>10C+ DC</td>
<td>A($P02CRP,$P03NX)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00020D</td>
<td>0000D24</td>
<td>10C+ DC</td>
<td>(15)+F'O'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000310</td>
<td>5B10 D49C</td>
<td>00480</td>
<td>0.106+$P03DD</td>
<td>L</td>
<td>1,=A($TRACE)</td>
</tr>
<tr>
<td>000314</td>
<td>D2FF 1000 1001 00000000000000000000000000000000</td>
<td>107+ MVC</td>
<td>0(256,1),1(1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00031A</td>
<td>9203 100</td>
<td>001A0</td>
<td>108+ MFI</td>
<td>256(1),X'03'</td>
<td></td>
</tr>
<tr>
<td>00031A</td>
<td>4B10 D286</td>
<td>002CA</td>
<td>109+ LA</td>
<td>1,+$P03SV+2</td>
<td></td>
</tr>
<tr>
<td>000322</td>
<td>4111 0001</td>
<td>00001</td>
<td>110+ STH</td>
<td>1,+$P03SV+2</td>
<td></td>
</tr>
<tr>
<td>000326</td>
<td>4010 D286</td>
<td>002CA</td>
<td>111+ MVI</td>
<td>$P03SV,X'FE'</td>
<td></td>
</tr>
<tr>
<td>00032A</td>
<td>920F D24B</td>
<td>002C9</td>
<td>112+ MVI</td>
<td>$P03SV,X'FE'</td>
<td></td>
</tr>
<tr>
<td>00032E</td>
<td>5230 D488</td>
<td>0049C</td>
<td>114+ L</td>
<td>3,YID</td>
<td></td>
</tr>
<tr>
<td>000332</td>
<td>0700</td>
<td></td>
<td>115+ CALL</td>
<td>NEXTPROG</td>
<td></td>
</tr>
<tr>
<td>000334</td>
<td>47F0 D32A</td>
<td>003JC</td>
<td>116+ CNOP</td>
<td>0,4</td>
<td></td>
</tr>
<tr>
<td>00033A</td>
<td>00000000</td>
<td>003JC</td>
<td>117* B</td>
<td>+8 BRANCH AROUND VCON</td>
<td></td>
</tr>
<tr>
<td>00033B</td>
<td>00000000</td>
<td>003JC</td>
<td>119+ L</td>
<td>15,INBOUROB LOAD 15 WITH ENTRY ADR</td>
<td></td>
</tr>
<tr>
<td>000340</td>
<td>005F</td>
<td>0033B</td>
<td>120+ L</td>
<td>15,INBOUROB LOAD 15 WITH ENTRY ADR</td>
<td></td>
</tr>
<tr>
<td>000342</td>
<td>5030 D2D4</td>
<td>002C8</td>
<td>121+ BALB</td>
<td>14,15 BRANCH TO ENTRY POINT</td>
<td></td>
</tr>
<tr>
<td>000342</td>
<td>5030 D2D4</td>
<td>002C8</td>
<td>125+ ST</td>
<td>3,+$P03SV+32</td>
<td></td>
</tr>
<tr>
<td>000346</td>
<td>92FF D284</td>
<td>002C9</td>
<td>126+ MVI</td>
<td>$P03SV,X'FF'</td>
<td></td>
</tr>
<tr>
<td>00034A</td>
<td>98EC D2C0</td>
<td>002D4</td>
<td>127+ LM</td>
<td>14,12,$P03SV+12</td>
<td></td>
</tr>
<tr>
<td>00034E</td>
<td>97FE</td>
<td>002D4</td>
<td>128+ BR</td>
<td>14</td>
<td></td>
</tr>
</tbody>
</table>
### Example of Debug Facilities

<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJECT CODE</th>
<th>ADDR1</th>
<th>ADDR2</th>
<th>STMT</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>000350</td>
<td>47F0 D354</td>
<td>0035A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000354</td>
<td>0224C259</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000355</td>
<td>02BC D354</td>
<td>0036C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00035C</td>
<td>47F0 D39A</td>
<td>003A8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000360</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000364</td>
<td>PD040000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000366</td>
<td>00000280000004</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00036C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00039A</td>
<td>5810 D49C</td>
<td>00400</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00039C</td>
<td>D2FF 1000 0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00039E</td>
<td>0200 D3E8</td>
<td>00362</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0003BE</td>
<td>4010 D4B2</td>
<td>00362</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0003C2</td>
<td>92FF D34C</td>
<td>00360</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0003C6</td>
<td>5830 D4BC</td>
<td>00400</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0003CA</td>
<td>1863</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0003CC</td>
<td>5950 D4A0</td>
<td>004B4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0003D0</td>
<td>00322</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0003D4</td>
<td>003B0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0003DE</td>
<td>4111 0001</td>
<td>0001</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0003DC</td>
<td>0110 D3D4</td>
<td>00323</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0003E0</td>
<td>0770 D36E</td>
<td>003A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0003F4</td>
<td>C233D2F6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0003FB</td>
<td>0090</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0003FA</td>
<td>4523 D2A4</td>
<td>002B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0003EE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0003F2</td>
<td>902C D400</td>
<td>00414</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0003F9</td>
<td>5060 D670</td>
<td>0047C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0003FC</td>
<td>92FF D342</td>
<td>00360</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000400</td>
<td>98EC D448</td>
<td>0045C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000404</td>
<td>077E</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000408</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00040C</td>
<td>00000360000004</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000410</td>
<td>00000000000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000450</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000454</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Example continues*
## Example of Debug Facilities

<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJECT CODE</th>
<th>ADDR1</th>
<th>ADDR2</th>
<th>STAT</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>000450</td>
<td></td>
<td></td>
<td></td>
<td>FINAL</td>
<td></td>
</tr>
<tr>
<td>000496</td>
<td></td>
<td></td>
<td></td>
<td>$LASTSAVE = PO4BCK</td>
<td></td>
</tr>
<tr>
<td>000498</td>
<td></td>
<td></td>
<td></td>
<td>DS OF $PO4WD = 0</td>
<td></td>
</tr>
<tr>
<td>000496</td>
<td></td>
<td></td>
<td></td>
<td>$LASTSAVE = PO4BCK</td>
<td></td>
</tr>
<tr>
<td>000498</td>
<td></td>
<td></td>
<td></td>
<td>DS OF $PO4WD = 0</td>
<td></td>
</tr>
<tr>
<td>0004A0</td>
<td></td>
<td></td>
<td></td>
<td>XID = 'XXXX'</td>
<td></td>
</tr>
<tr>
<td>0004A0</td>
<td></td>
<td></td>
<td></td>
<td>YID = 'yyyy'</td>
<td></td>
</tr>
<tr>
<td>0004A0</td>
<td></td>
<td></td>
<td></td>
<td>ZID = 'ZZZZ'</td>
<td></td>
</tr>
<tr>
<td>0004A0</td>
<td></td>
<td></td>
<td></td>
<td>LTORG</td>
<td></td>
</tr>
<tr>
<td>0004A0</td>
<td></td>
<td></td>
<td></td>
<td>=A(SFIRSTSV)</td>
<td></td>
</tr>
<tr>
<td>0004A0</td>
<td></td>
<td></td>
<td></td>
<td>=A($LASTSAVE)</td>
<td></td>
</tr>
<tr>
<td>0004A0</td>
<td></td>
<td></td>
<td></td>
<td>=A($TRACE)</td>
<td></td>
</tr>
<tr>
<td>0004A0</td>
<td></td>
<td></td>
<td></td>
<td>=F'1'</td>
<td></td>
</tr>
<tr>
<td>000490</td>
<td></td>
<td></td>
<td></td>
<td>END</td>
<td></td>
</tr>
</tbody>
</table>

- 00045000
- 00046000
- 00047000
- 00048000
- 00049000
- 00050000
- 00052000

23 JAN 74
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 true 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:

\[(\text{opcode}, \text{opl}, \ldots, \text{opn}, \text{bc-spec}) \text{ or } (\text{bc-spec})\]

The simple conditional has the logical value true if the bc-spec is true 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:

\[
\text{scond1}, \text{AND}, <, \text{scond2}, \text{OR}, \text{scond3}, >
\]

(*)

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

\[
\text{scond1}, \text{AND}, \text{scond2}, \text{OR}, \text{scond3}
\]

is the same as

\[
<, \text{scond1}, \text{AND}, \text{scond2}, >, \text{OR}, \text{scond3}
\]

The logical value of a conditional expression is \textbf{true} if the logical result of the indicated operations on the values of the simple conditionals is \textbf{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 \textit{scond1} is \textit{false}, the expression must be \textit{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`

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

**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.

\[ \text{BLEND [block-name] [other-ops]} \]

- **block-name**
  Indicates that this BLEND is intended to match the BLOCK or other block-defining macro named \textit{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**

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.

\[
\begin{array}{|c|c|}
\hline
\text{blname} & \text{CASE} \\
\hline
\text{index-list} & \text{MISC} \\
\text{char-index-list} & \text{sym} \\
\text{conditional-test} & \text{index-list} \\
\hline
\end{array}
\]

\textit{blname}

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

\textit{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.

\textit{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. \textit{index-list} is invalid with the character-string or conditional-test forms of the DOCASE. If \textit{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.

\textit{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).
CASE

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

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.

<table>
<thead>
<tr>
<th>[label]</th>
<th>CORP</th>
<th>[proc-name]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>[RESTORE=(first [,last])]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[RETURN=reg-list]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[RC= {NONE}]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[value (reg)]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[LINK= {NONE}]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[linkreg]</td>
</tr>
</tbody>
</table>

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

proc-name

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=\text{value}

Indicates the number \text{value} is to be returned in register 15.

RC=(\text{reg})

Indicates the value in register \text{reg} is to be returned in register 15.

If RC= is not coded the defaults are:

For OS procs: \hspace{1cm} RC=0
For non-OS procs: \hspace{1cm} 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=\text{linkreg}

Indicates a final "BR \text{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.

<table>
<thead>
<tr>
<th>biname</th>
<th>DO</th>
<th>[FOREVER]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>WHILE,looping-group</td>
<td>[AND],UNTIL,looping-group</td>
</tr>
<tr>
<td></td>
<td>UNTIL,looping-group</td>
<td>[AND],WHILE,looping-group</td>
</tr>
</tbody>
</table>

*biname*  
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:

\[
\left\{ \text{looping-branch} \left[ \{ \text{AND} \}, \text{cond-test} \right], \text{cond-test} \right\}
\]

looping-branch

One of the special looping instructions specified as:

- \((\text{BCT}, \text{reg1})\)
- \((\text{BXH}, \text{reg1}, \text{reg2})\)
- \((\text{BXLE}, \text{reg1}, \text{reg2})\)

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

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.)

<table>
<thead>
<tr>
<th>blname</th>
<th>DOCASE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>([index-word]&lt;br&gt;[(index-reg)]&lt;br&gt;{W, H, B, length}&lt;br&gt;[,SIMPLE]&lt;br&gt;[,IFANY]&lt;br&gt;[,ONLY]&lt;br&gt;[,SPARSE])</td>
</tr>
</tbody>
</table>

- **blname**: 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`.
Indicates the DOCASE index is located in the byte at address index.

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 ≤ 6 (if no MISC CASE is present) or n ≤ 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.
The DOCASEND macro is provided as an alias for the ESACOD macro. See ESACOD for description.
DOEND—Terminate Iteration Block

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

| DOEND | [block-name] |
ELSE

**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.

<table>
<thead>
<tr>
<th>else-name</th>
<th>ELSE</th>
<th>BLEND=if-name</th>
</tr>
</thead>
</table>

### else-name

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

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.

<table>
<thead>
<tr>
<th>ESAC</th>
<th>[block-name]</th>
</tr>
</thead>
</table>

*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

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.

<table>
<thead>
<tr>
<th>ESACOD</th>
<th>[block-name]</th>
</tr>
</thead>
</table>

*block-name*  

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**

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

**block-name**

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.

<table>
<thead>
<tr>
<th>FI</th>
<th>[block-name]</th>
</tr>
</thead>
</table>

*block-name* 

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.
IF

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.

\[
\begin{array}{|c|c|}
\hline
\text{blname} & \text{IF} \\
\hline
\{ \text{ASYNCH} \} & \{ \text{cond-test} \} \\
\{ \text{EXIT=} \{ \text{exit-block} \} \} & \{ \text{ELSE=} \text{else-block} \} \\
\hline
\end{array}
\]

\text{blname}

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

\text{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.

\text{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.

\text{EXIT=} \text{exit-block}

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

A-24
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  

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

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.

<table>
<thead>
<tr>
<th>OD</th>
<th>[block-name]</th>
</tr>
</thead>
</table>

**block-name**

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.

\begin{tabular}{|c|c|}
\hline
ONEND & [block-name] \\
\hline
\end{tabular}
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.

<table>
<thead>
<tr>
<th>ONEXIT</th>
<th>[block-name]</th>
</tr>
</thead>
</table>

*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

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.

<table>
<thead>
<tr>
<th>proc-name</th>
<th>PROC</th>
<th>LINKAGE=(OS, CSECT)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ID= NONE, id-string</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SAVE= NONE, first, last, DYNAM, NONE, length</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BASE= NONE, basereg, baselist</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WORK= NONE, workreg</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EXIT= severity</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DEBUG= options-list</td>
<td></td>
</tr>
</tbody>
</table>

proc-name

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.
PROC

LINKAGE=(, CSECT)

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

ID=NULL

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=NULL

SAVE=NULL 

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.
PROC

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=(,reg2,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

A-32
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 char-
acter constant.

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

A-33
PROC

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 invoked.

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., "NPROCTRACE" 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.

<table>
<thead>
<tr>
<th>$[\text{label}]$</th>
<th>PROCEND</th>
<th>$[\text{proc-name}]$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$[\text{(RC=} \text{NONE})]$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$[\text{(RC=} {\text{value}})]$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$[\text{(RC=} {\text{reg}})]$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$[\text{RESTORE=} {\text{first [, last]}}]$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$[\text{RETURN=} {\text{reglist}}]$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$[\text{LINK=} {\text{NONE}}]$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$[\text{LINK=} {\text{linkreg}}]$</td>
</tr>
</tbody>
</table>
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), 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 arbitrary 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.)

---


B-1
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** - Integers
- **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**: $0 \rightarrow false$; 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 $\rightarrow 0$; true $\rightarrow 1$

B-2
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 al-
lowed. They are dimensioned in their declarations as

\[ \text{int } X(20) \]

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 preceding 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:

\[ \text{if } I = 1 \text{ or } I = 19 \]

is the same as

\[ \text{if } I = 1 \text{ or } I = 19 \]

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

\[ \text{if } J \neq 0 \text{ and } I / J = 4 \]

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:

\[
\begin{align*}
\text{// } & \text{...} \\
\end{align*}
\]
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.
<table>
<thead>
<tr>
<th>FOREVER</th>
<th>WHILE</th>
<th>UNTIL</th>
<th>Entry point - 0</th>
<th>After Code - 8</th>
<th>Looping Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>7 4 4 4 4 4 4 4 4 4 4 9 9 9 9 9 9 9 9 9 9 9 9 9 7 7 7 7</td>
<td>7 4 1 1 1 9 9 9 9 9 9 9 9 9 9 9 9 1 1 1 1 1 9 1 9 9</td>
<td>4 7 1 4 1 7 7 7 7 4 4 4 4 7 7 7 1 7</td>
</tr>
<tr>
<td></td>
<td>looping-branch</td>
<td></td>
<td></td>
<td></td>
<td>Branch leg 2</td>
</tr>
<tr>
<td></td>
<td>AND/OR</td>
<td></td>
<td></td>
<td></td>
<td>'Branch' leg 10</td>
</tr>
<tr>
<td></td>
<td>cond-test A</td>
<td></td>
<td></td>
<td></td>
<td>'Branch' leg 11</td>
</tr>
<tr>
<td></td>
<td>AND/OR</td>
<td></td>
<td></td>
<td></td>
<td>'Fall thru' leg 6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>'Fall thru' leg 11</td>
</tr>
<tr>
<td></td>
<td>cond-test B</td>
<td></td>
<td></td>
<td></td>
<td>'Fall thru' leg 6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>'Fall thru' leg 11</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OPERAND FORMAT</td>
</tr>
</tbody>
</table>

| 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,
(CLISPOT,C'X',E)
```

Operand format number 6.

Code generated:

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

<Block's code>

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

B-8
APPENDIX C

ABSTRACT SOURCE LISTING OF STRCMACS
Global Definitions -- 14 July 1973

135. /* GENERAL PURPOSE GLOBALS. */
136. PROC_COUNTER, /* Number for last special proc label "$Pep". The
137. is PROC_COUNTER in hex. */
138. LAST_BLOCK_NUMBER, /* Number used in labels of most recently generated
139. block. */
140. HEX_IN, /* Input value to XHEX macro. */
141. ERROR_OCCURRED, /* General purpose error flag used by various
142. inner macros to report failure. */
143. SDL_FALLTHRU_OCCURS, /* Set true by TERMINATE DO LOOP if DO may
144. fall through bottom in terminating (when a looping branch is
145. present); else set false. */
146. NOT_FIRST_PROC, /* Set true by TERMINATE DO LOOP if DO may
147. may
148. fall through bottom in terminating (when a looping branch is
149. present); else set false. */
150. SAVETRACE_ON_FIRST_PROC, /* false until first PROC block
151. included */
152. DEBUG=(--.,SAVETRACE,-) operand. */
153. TRACE_VECTOR_GIVEN /* false until the trace vector generated on
154. the first PROC which includes the
155. DEBUG=(**,PROCTRACE,-*) operand. */
156. BLOCK_LABEL_PREFIX, /* Unique character string for each block
157. for use in generating labels. */
158. EXIT_SEVERITY, /* Mnote severity for EXIT target message. Can be
159. set by PROC macros. */
160. HEX(16), /* Constants used in converting decimal to hex by XHEX. */
161. HEX_OUT, /* Output value from XHEX macro. */
162. PREV_SAVETRACE_AREA, /* Holds label generated on last local PROC
163. save area to be used in producing the static chain for SAVETRACE. */
164. PREV_SAVETRACE_PTR /* Holds label generated as forward pointer
165. within last local PROC save area for static chain for SAVETRACE. */
166. DEBUG_FLAGS, /*
167. DEBUG_BLOCKCOUNTS_REQD, /* Causes code and counters to keep execution
168. counts on all blocks. */
169. DEBUG_BLOCKNAMES_REQD, /* Causes block names to be generated as
170. inline character constants to aid in locating within dumps. */
171. DEBUG_REGISTERVALUES_REQD, /* Causes register values at the start of proc (at
172. CORP macro start) to be saved in inline save areas for reference. */
173. DEBUG_BLOCKVALUES_REQD, /* Causes various intermediate values to be
174. printed during macro processing for debugging the macros. */
175. DEBUG_LISTBLOCKS_REQD, /* Causes mnotes to be generated at the start
176. and end of all blocks listing their name, number, and static
177. nesting depth. */
178. DEBUG_MACRONAMES_REQD, /* Causes mnotes to be generated whenever any
179. macro is entered (including inner macros) which list the macro's
180. name; for debugging the macros. */
181. DEBUG_PROCVALUES_REQD, /* Causes code and counters to keep execution
182. counts on PROC blocks only. */
183. DEBUG_PROCCOUNTS_REQD, /* Causes PROC macros to be generated as
184. character constants to aid in locating within dumps. */
185. DEBUG_PROCTRACE_REQD, /* Causes a trace vector to be generated and
186. code to be generated to keep track of the last 257 PROCs entered. */
187. DEBUG_SAVETRACE_REQD /* Causes all local save areas to be statically
188. chained together and code to be generated to link the chain to the
189. OS area to provide OS formatting within ABEND dumps. */
Global Definitions -- 19 July 1973

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

198. int CURRENT_NEST_LEVEL, /* Current depth of static nesting of blocks; stack pointer. */
199. NESTING_LIMIT, /* Holds dimension of main stack. */
200. BLOCK_NESTER(100), /* Block number of the Ith block. */
201. END_LABEL_REQD(100), /* Indicates whether Ith block needs an END label generated during POP_OLD_BLOCK. */
202. EXIT_LABEL_REQD(100) /* Indicates whether Ith block needs an EXIT label generated during POP_OLD_BLOCK. */
203. BLOCK_NAME(100), /* Block name of Ith block, either USER_NAME specified in macro label field or generated name "BLKnn" where nnn is the sequential block number. */
204. BLOCK_TYPE(100), /* Macro name which generated the Ith block (IF, DO, DOCASE, CASE, BLOCK, or PROC). */
205. OPERAND1(100), OPERAND2(100), OPERAND3(100), OPERAND4(100), /* These hold various data which are needed to close the blocks generated. Specific contents vary according to the type of block generated. See individual macros. */
206. INFORMATION(100) /* Similar to the OPERAND stacks above, the INFORMATION stack holds information for the closing of the block. Often the individual characters within the variables are used for different values, packed together into INFORMATION. */
207. /* GCASE STACK. Holds data for general DOCASES. Dimensioned to 9. */
208. int MAX_CASE_VALUE(9), /* Maximum branch vector value found. */
209. NEXT_CMP_LABEL_NO(9), /* Case number for next comparison case label to be generated. */
210. GCASE_NEST_LEVEL, /* Current depth of stacking in the GCASE stack; number of nested DOCASES with either GENERAL, SPARSE, or CHARCOMP operand formats. */
211. GCASE_NEST_LIMIT /* Maximum depth of nesting of GCASE stack; must be equal to stack dimension. */
212. CASE_OCCURS(2304) /* Each group of 256 bits are used to note which branch vector cases occur. */
213. /* CONDITIONAL_EXPRESSION_PROCESSOR PSEUDC-PARAMETERS. */
214. int FIRST_INDEX,
215. LAST_INDEX /* Pseudo-parameters to CONDITIONAL_EXPRESSION_PROCESSOR. Indicates indexes within SYSLIST of first and last parameter to be processed. */
216. ULTIMATE_FALLTHRU_LABEL, /* Logical value upon which conditional expression is to pass control (or fall through) to the ULTIMATE_FALLTHRU_LABEL. */
217. ULTIMATE_FALLTHRU_LABEL_USED /* CEP sets this true if a branch is generated to the ULTIMATE_FALLTHRU_LABEL (else no change occurs). */
218. UNIQUE_LABEL_ID /* One character unique to this call of CEP used to insure labels generated by this call will differ from all other labels, even others within the same macro (particularly for DO). */
"IF" Macro -- 21 June 1973

1100.1. Macro IF (USER_NAME, REL=, MASK=, EXIT=, ELSE=)

1100.3. /* Initiate a block in the structure. Save any information needed
1100.4. by ELSE or FL. For ASYNCH type, generate branch around block.
1100.5. For normal IF (EXIT= not specified), generate conditional expression
1100.6. tests with branch around block (or to ELSE) for false and fall
1100.7. through for true; if EXIT= specified, then generate branch to
1100.8. proper block end for true, fall through for false, and delete
1100.9. IP block from structure. Put USER_NAME on first executable
1100.10. instruction if one specified. */
1100.11. bit VALID_EXIT
1100.12. /* VALID_EXIT is true if EXIT= was specified and no errors have
1100.13. been found to cause the EXIT to be ignored. */
1100.14. char EXIT_LABEL, /* Label for EXIT= branch, when deferred until
1100.15. after block count has been incremented. */
1100.16. LABEL /* Outstanding label, waiting to be generated. */
1100.17. if ERROR_OCCURRED
1100.18. then
1100.19. mexit
1100.20. fi
1100.21. if REL "" or MASK ""
1100.22. then
1100.23. mnote (8, "STRC1102 REL= OR MASK= NOT IN PARENTHESES—IGNORED")
1100.24. fi
1100.25. LABEL := USER_NAME /* Generate USER_NAME at first opportunity. */
1100.26. VALID_EXIT := (EXIT 4,
1100.27. if SYSLIST(1,1) = 'ASYNCH'
1100.28. then /* Either "IF ASYNCH" or "IF (ASYNCH)" was entered. */
1100.29. call IF ASYNCH BRANCH /* Generate branch around block. */
1100.30. else
1100.31. call IF_CONDITIONAL_GENERATOR /* Generate conditional tests according to IF macro operands and
1100.32. the current conditional test specs. */
1100.33. fi
1100.34. if LABEL "",
1100.35. then
1100.36. generate (LABEL || DS OR*)
1100.37. fi
1100.38. if VALID_EXIT /* i.e., if EXIT specified and still valid...
1100.39. then
1100.40. call IF_CONDITIONAL_GENERATOR /* Generate conditional tests according to IF macro operands and
1100.41. the current conditional test specs. */
1100.42. call IF_CONDITIONAL_GENERATOR
1100.43. /* Generate conditional tests according to IF macro operands and
1100.44. the current conditional test specs. */
1100.45. call IF_CONDITIONAL_GENERATOR
1100.46. /* Generate conditional tests according to IF macro operands and
1100.47. the current conditional test specs. */
1100.48. fi
1100.49. call IF_CONDITIONAL_GENERATOR
1100.50. /* Generate conditional tests according to IF macro operands and
1100.51. the current conditional test specs. */
1100.52. if LABEL "`
1100.53. then
1100.54. generate (LABEL || DS OR*)
1100.55. fi
1100.56. if VALID_EXIT
1100.57. then /* No IF block remains after completion of "IF EXIT=..." macro;
1100.58. simulate presence of FI macro. */
1100.59. call FI ()
1100.60. fi
1100.61. send
11076. BLOCK IF_ASYNCH_BRANCH
11077. /* Give error message if EXIT specified. Generate branch to
11078. end of IF block. */
11079.
11080. if VALID_EXIT
11081. then
11082. -note (9, 'STRC1101 EXIT= IGNORED WITH "ASYNCH"')
11083. VALID_EXIT := false
11084. end
11085. generate (LABEL || ' B ' || BLOCK_LABEL_PRE-fix || 'END')
11086. /* Branch around asynchronous IF block. */
11087. LABEL := " "
**"IF" Macro**  --  21 June 1973

```
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 II '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 II 'BEG'
11110.   /* Fall-through label to be used in ELSE. */
11111.   ULTIMATE_FALLTHRU_CONDITION := true
11112.   /* Fall through if conditional test yields true result. */
11113.   FALLTHRU_LABEL_USED := false  /* Assume not required. */
11114.  ENDP
```
"IF" macro -- 21 June 1973

```
1116. BLOCK 'EXIT_SPECS
1117. /* An EXIT= operand has been specified; insure ELSE= not also
1118. specified. If valid, change conditional test specs to standard
1119. for EXIT-type IF including the assigning of the branch target
1120. as the XIT label of the block specified by the EXIT keyword. */
1121. CATCH HOLD /* Temporary. */
1122. if ELSE * * /* If ELSE= was not omitted... */
1123. then  
1124. "STRC1103 EXIT= IGNORED WITH ELSE=")
1125. VALID_EXIT := false
1126. else
1127. HOLD := ULTIMATE_BRANCH_LABEL
1128. /* Save old branch label, we may need it yet. */
1129. call EXIT_FIND ( : EXIT)
1130. /* Sets ULTIMATE_BRANCH_LABEL to XIT label of block whose name
1131. is specified in the argument; if none specified ("EXIT=*"),
1132. use block surrounding IF macro; if no such block, issue message,
1133. leave ULTIMATE_BRANCH_LABEL unmodified, and set ERROR_OCCURRED to
1134. true. Mark target block as requiring XIT label. */
1135. if DEBUG_BLOCKCOUNTS_REQD
1136. then
1137. EXIT_LABEL := ULTIMATE_BRANCH_LABEL
1138. ULTIMATE_BRANCH_LABEL := HOLD
1139. /* Make EXIT-type IF act like regular IF (i.e., fall through on true)
1140. so we can count the number of times the exit is taken; save the
1141. XIT_LABEL for a branch after the count is made and make the
1142. ULTIMATE_BRANCH_LABEL whatever it would have been had this been
1143. a regular IF. */
1144. else
1145. ULTIMATE_FALLTHRU_LABEL := BLOCK_LABEL_PREFIX
1146. ULTIMATE_FALLTHRU_CONDITION := false /* Fall through on false. */
1147. END_LABEL_REQD(CURRENT_NEST_LEVEL) := false
1148. if ERROR_OCCURRED /* on EXIT_FIND... */
1149. then
1150. /* Exit point not found and message has been issued. Make branch
1151. point same as fall-through point and clear error (i.e., fix up
1152. and continue). */
1153. ULTIMATE_BRANCH_LABEL := BLOCK_LABEL_PREFIX
1154. FALLTHRU_LABEL_USED := true
1155. /* Since it's also the branch label. */
1156. ERROR_OCCURRED := false
1157. fi
1158. else
1159. fi
1160. CATCH
```

C-7
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. */
11170. /* Set up further specifications required by CONDITIONAL_EXPRESSION-
11171. PROCESSOR. */
11172. FIRST_INDEX := 1
11173. LAST_INDEX := *'SYSLIST
11174. /* The operands of SYSLIST to be processed are operand 1 through the
11175. last operand (SYSLIST(\*'SYSLIST)), i.e., all of them. */
11176. UNIQUE_LABEL_ID := 'I'
11177. /* Used by CONDITIONAL_EXPRESSION_PROCESSOR to produce unique
11178. labels. */
11179. CALL CONDITIONAL_EXPRESSION_PROCESSOR (LABEL; SYSLIST)
11180. /* Generate code corresponding to the operands of the IF (referred
11181. to collectively as SYSLIST). Only the SYSLIST can be passed
11182. directly as arguments; the following variables are effectively
11183. arguments but are passed in global variables:
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

C-8
11206. **PROC IP_BLOCK_COUNT**
11207. /* If debugging in progress, generate block name and/or count of block execution. Note that ASYNCH blocks cannot be counted. */
11208. 11209.  
11210. **EXEC** TARGET /* Temporary. */
11211. **IF** SYSLIST(1,1) = 'ASYNCH'
11212. **THEN**
11213. **IF** DEBUG_BLOCKNAME_REQD
11214. **THEN**
11215. **GENERATE** (" DC C'\" || BLOCK_NAME(CURRENT_NEST_LEVEL) ||
11216. "*','0'\"")
11217. /* Asynch branch has already occurred; only name required. */
11218. **ELSE** /* Not ASYNCH. */
11219. **IF** DEBUG_BLOCKCOUNTS_REQD OR DEBUG_BLOCKNAMES_REQD
11220. **THEN**
11221. **IF** DEBUG_BLOCKCOUNTS_REQD
11222. **THEN**
11223. /* Generate code to increment block execution count. */
11224. **GENERATE** (LABEL || ' L ' || BLOCK_LABELPREFIX || 'IFC')
11225. LABEL := ' '  
11226. **GENERATE** (' IA 1,1(1)')
11227. **GENERATE** (' STH 1,1 ' || BLOCK_LABELPREFIX || 'IFC')
11228. ///* Generate branch around block name and/or block count. */
11229. **IF** EXIT_LABEL = ' '
11230. **THEN**
11231. TARGET := BLOCK_LABELPREFIX || 'GO'
11232. /* Branch directly around block name/count. */
11233. **ELSE**
11234. TARGET := EXIT_LABEL
11235. /* Branch to end of BXIT= block, postponed to here so we could do the counting. */
11236. **IF** EXIT_LABEL = ' '  
11237. **THEN**
11238. **GENERATE** (LABEL || ' B ' || TARGET)
11239. **IF** EXIT_LABEL = ' '  
11240. **THEN**
11241. **LABEL** := TARGET
11242. /* Label for branch-around must be defined. */
11243. **ELSE**
11244. **LABEL** := ' '
11245. **IF** DEBUG_BLOCKNAMES_REQD
11246. **THEN**
11247. **GENERATE** (' DC C'\" || BLOCK_NAME(CURRENT_NEST_LEVEL) ||
11248. "*','0'\"")
11249. **ELSE**
11250. **IF** DEBUG_BLOCKCOUNTS_REQD
11251. **THEN**
11252. **GENERATE** (BLOCK_LABELPREFIX || "IFC DC '0' IP COUNT")
11253. **ELSE**
11254. **GENERATE**
11255. /* Note: */
11256. **FILE**
11257. /* Note: */
11258. **FILE**
"ELSE" Macro -- 26 June 1973

```c
13001. BASIS 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 ELSE_BLOCK_NO /* Block number which will be assigned to
13008. upcoming ELSE block. */
13009. char IF_EXIT_LABEL /* Exit label for IF block if one was to have been
13010. generated, else null. */
13011. int ELSE_BLOCK_NO /* Block number which will be assigned to
13012. upcoming ELSE block. */
13013. char IF_EXIT_LABEL /* Exit label for IF block if one was to have been
13014. generated, else null. */
13015. CALL TRACE_PRINTER ( ; 'ELSE') /* Print macro name "ELSE" in mnote if tracing on. */
13016. if CURRENT_NEST_LEVEL > NESTING_LIMIT then
13017. then
13018. fi
13019. CALL VERIFY_END( ; 'IF', BLEND)
13020. /* Verifies current block has the name specified
13021. by the BLEND= operand on the ELSE macro (if any) and that it is an IF block.
13022. Various errors receive messages and either intermediate blocks are
13023. BLENDed as a fixup or ERROR_OCCURRED is set. */
13024. if ERROR_OCCURRED then
13025. then
13026. fi
13027. if OPERAND1(CURRBENT_NEST_ LEVEL) # OPERAND2(CURRBENT_NEST_ LEVEL) then
13028. /* ELSE=elsename specified on IF macro but different (or no) label
13029. field on ELSE macro. */
13030. then
13031. /* ELSE=elsename specified on IF macro but different (or no) label
13032. field on ELSE macro. */
13033. if OPERAND2(CURRENT_NEST_LEVEL) = 'ELSE' then
13034. /* ELSE=elsename specified on IF macro but different (or no) label
13035. field on ELSE macro. */
13036. then
13037. /* ELSE=elsename specified on IF macro but different (or no) label
13038. field on ELSE macro. */
13039. else
13040. if OPERAND2(CURRENT_NEST_LEVEL) = 'ELSE' and OPERAND2(CURRENT_NEST_LEVEL)
13041. then
13042. ELSE_BLOCK_NO := LAST_BLOCK_NUMBER + 1
13043. /* Generate branch to end of ELSE block. */
13044. if ELSE_BLOCK_NO < BLOCK_NUMBER(CURRENT_NEST_LEVEL)
13045. then
13046. EXIT_LABEL_REQD(CURRENT_NEST_LEVEL) := false
13047. /* Exit label has been postponed to FI. */
13048. if
13049. then
13050. CALL POP_OLD_BLOCK ( ; ) /* Remove IF block from stack. */
13051. CALL PUSH_NEW_BLOCK(USER_NAME; BLOCK_TYPE_VALUE='IF', /* Must be marked as "IF" block for FI's
13052. END_LABEL_VALUE=ELSE, OPERAND2_VALUE='ELSE',
13053. OPERAND3_VALUE=IF_EXIT_LABEL) /* Generate new block in structure. It is marked as an IF block
13054. to simplify FI checking; "ELSE" is stored in OPERAND2, however,
13055. to indicate that an ELSE has been generated for this IF. An END
13056. label will be required for the block and, if the IF block was
13057. marked as needing an XLT label, save it in OPERAND3. */
13058. if USER_NAME #'' then
13059. /* Define outstanding label. */
13060. endif
13061. endif
13062. endif
13063. endif
13064. endif
13065. endif
13066. endif
13067. endif
13068. endif
13069. endif
13070. endif
```
MACRO "FI" ( : USER_NAME)

/* Generates end to match IF (or ELSE) block. Standard block closing
occurs. */

CALL TRACE_PRINTER ( ; "FI")

/* Prints macro name "FI" in note if tracing on. */

if CURRENT_NEST_LEVEL > NESTING_LIMIT
  then
    CALL POP_OLD_BLOCK ( ; )
    fi
fi

CALL VERIFY_END ( ; 'IF', USER_NAME)

/* Verifies current block has the name specified by the USER_NAME operand on the FI macro (if any) and that it is an IF block. Various errors receive messages and either intermediate blocks are BLENDED as a fixup or ERROR_OCCURRED is set. */

if CURRENT_NEST_LEVEL > 0 and
  [USER_NAME = ' ' or = BLOCK_NAME(CURRENT_NEST_LEVEL)] and
  BLOCK_TYPE(CURRENT_NEST_LEVEL) = 'IF', then
  ERROR_OCCURRED will be set false and CURRENT_NEST_LEVEL will be unmodified. */

if ERROR_OCCURRED
  then
    mexit
fi

if OPERAND1(CURRENT_NEST_LEVEL) = ''
  then
    mnote (8, 'STRC1501 ELSE BLOCK "' II OPERAND1(CURRENT_NEST_LEVEL) " NOT FOUND')
  fi

call POP_OLD_BLOCK ( ; OPERAND3(CURRENT_NEST_LEVEL))

/* Delete current block, generating END and XII labels as required. (Lemma: Execution of POP_OLD_BLOCK always results in decrementing of CURRENT_NEST_LEVEL by exactly 1.) */

mend

/* (Lemma: If CURRENT_NEST_LEVEL > 0 and
  [USER_NAME = ' ' or = BLOCK_NAME(CURRENT_NEST_LEVEL)] and
  BLOCK_TYPE(CURRENT_NEST_LEVEL) = 'IF' at entry to FI, then
  CURRENT_NEST_LEVEL will be decremented by exactly 1.) */
DO Macro  --  21 June 1973

FLOW POINT 0: Branch to entry point for total block.
FLOW POINT 1: Start of UNTIL tests, labeled $nnnUl.
FLOW POINT 2: Branch on success of UNTIL tests.
FLOW POINT 3: Fall-through on failure of UNTIL tests.
FLOW POINT 4: Start of WHILE tests, labeled $nnnW1.
FLOW POINT 5: Branch on failure of WHILE tests.
FLOW POINT 6: Fall-through on success of WHILE tests.
FLOW POINT 7: Start of internal looping code (user code between DO and termination of DO loop by OD, ATEND, or TERMINATEDOLOOP).

FLOW POINT 8: End of internal DO code
FLOW POINT 9: Start of looping branch (BCT, BXH, or BILE), labeled $nnnLPB.
FLOW POINT 10: Branch of looping branch.
FLOW POINT 11: Fall-through of looping branch.
FLOW POINT 12: End of total block, labeled $nnnEND.

Flow points 1 through 7 are generated by DO; 8 through 12 by TERMINATEDOLOOP. */
DO Macro -- 21 June 1973

21075. call TRACERPRINT( : 'DO')
21076. /* Prints macro name 'DO' in trace if tracing on. */
21077. call PUSH_NEW_BLOCK(USER_NAME; BLOCK_TYPE_VALUE='EO')
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 EQ 0
21086. then
21087. /* Note (e.'STRC2113 REL=0 OR MASK= NOT IN PARENTHESES—IGNORED')
21088. */
21089. LABEL := USER_NAME
21090. call DO_SCAN_OPERANDS
21091. /* Collect scanning information and looping branch (BCZ, BXM, 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_BLOCKCOUNTERS_REQD
21100. then /* Generate reset of current loop count. */
21101. GENERATE(LABEL || 'SR 1,1')
21102. LABEL := ' ' || BLOCK_LABEL_PREFIX || 'DO1'
21103. fi
21104. if OPERAND_FORMAT #0
21105. then /* Not infinite loop. */
21106. call DO_LABEL_BLOCK
21107. /* Store begin label (flow point 7) into LABEL. */
21108. call DO_INFO_SAVE
21109. /* Insert into stack all information required by TERMINATE_DO_LOOP to
21110. close loop (flow points 8 through 12). */
21111. call DO_TRACE_COUNTERS
21112. /* Generate any debugging counters, etc. */
21113. fi
21114. call DO_LABEL_BLOCK
21115. /* Store begin label (flow point 7) into LABEL. */
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 OH')
21124. fi
21125. mend
PROC DO_SCAN_OPERANDS
/* Collect WHILE_INDEX, WHILE_END_INDEX, UNTIL_INDEX, UNTIL_END_INDEX, (limits of corresponding conditional test's operands within the SYSLIST) and note in WHILE_COND_TEST and UNTIL_COND_TEST whether the corresponding keywords include a conditional test to be generated; set looping branch information (LOOPING_BRANCH_TYPE, LB_OPCODE_ID, LB_OPERAND1, LB_OPERAND2, LB_LABEL_REQ, and LB_LOGIC_OP) which must be passed to TERMINATEDO_LOOP to close loop; and set OPERAND_FORMAT (case number code from decision table). */
call DO_FIND_KEYWORDS
and PRESENCE
/* Put operand index of "WHILE" and "UNTIL" keywords into xxxx_INDEX (or set to 0 if omitted) and note in xxxx_PRESENT whether these looping groups exist. Set LISTOP to index of last valid operand in the SYSLIST. */
call DO_FIND_ENDINDEXESAND_MAIN_OP
/* For each type xxxx (WHILE and UNTIL) which is present, put index of the last operand of looping group for that type into xxxx_END_INDEX; if both present, find logic operator which connects them and put it into MAIN_OP, else put in null string. */
call DO_LOOPING_BRANCH_AND_FIRST_OPERAND
/* Collect looping branch information and step WHILE_INDEX and UNTIL_INDEX to first operand of conditional test (not including looping branch and following operator) or set to zero if not present. Also set WHILE and UNTIL_COND_TEST to indicate presence of conditional tests. */
call DO_SET_FORMAT
/* Set type of operands according to decision table, using WHILE_PRESENT, UNTIL_PRESENT, MAIN_OP, WHILE_COND_TEST, UNTIL_COND_TEST, LOOPING_BRANCH_TYPE, and LB_LOGIC_OP to make decision. */
core
DO
Macro -- 21 June 1973

21160. DECK DO_FIND_KEYNDRS_AND_PRESENT

21163. Set WHILE_PRESENT and UNTIL_PRESENT true if corresponding looping

21164. group is present, else set to false. */

21165. WHILE_INDEX, UNTIL_INDEX := 0

21166. if SYSLIST(1) = 'WHILE' and SYSLIST(2) = 'UNTIL' and

21167. /* Find operand index

21168. of 'WHILE' and 'UNTIL' and

21169. "FOREVER" and */

21170. then

21171. error (8, 'STRC2108 FIRST OPERAND MUST BE "WHILE", "UNTIL", "FOREVER", OR OMITTED')

21172. fi

21173. if I = 1

21174. then

21175. LASTOP := N'SYSLIST /* Assuming they're all valid. */

21176. while I <= LASTOP

21177. do /* Search for WHILE and UNTIL keywords. */

21178. if SYSLIST(I) = 'WHILE'

21179. if WHILE_INDEX = 0

21180. then /* No WHILE found before */

21181. WHILE_INDEX := I

21182. else

21183. /* Operands After Second "WHILE" Ignored */

21184. LASTOP := I - 1

21185. fi

21186. else if UNTIL_INDEX = 0

21187. then /* No UNTIL found before */

21188. UNTIL_INDEX := I

21189. else

21190. /* Operands After Second "UNTIL" Ignored */

21191. LASTOP := I - 1

21192. fi

21193. fi

21194. fi

21195. fi

21196. fi

21197. fi

21198. if WHILE_INDEX > 1 and UNTIL_INDEX > 1

21199. then /* Garbage operands are present. */

21200. error (8, 'STRC2114 SUPERFLUOUS LOOPING GROUP IGNORED')

21201. fi

21202. /* Decide whether WHILE and UNTIL looping groups are present. The

21203. possible operand formats are:

21204. DO UNTIL,<looping-group>

21205. DO WHILE,<looping-group>

21206. DO UNTIL,<looping-group>,AND/OR,<looping-group>

21207. DO WHILE,<looping-group>,AND/OR,<looping-group>

21208. DO [No operand or single operand "FOREVER"

21209. means infinite loop.]

21210. until UNTIL_PRESENT := (UNTIL_INDEX > 0)

21211. WHILE_PRESENT := (WHILE_INDEX > 0 OR

21212. UNTIL_INDEX > 0)

21213. /* Last two alternatives are only to fix up when WHILE was

21214. omitted. */

21215. if ~UNTIL_PRESENT and LASTOP > 0)

21216. /* Do */

21217. /* */

21218. /* */

21219. /* */

21220. /* */

C-15
```
*DO* MACRO -- 21 June 1973

21223. DO_FIND_END_INDEXES_AND_MAIN_OP
21224. /* For each type xxxx (WHILE and UNTIL), put index of last operand of
21225. looping groups for that type into xxxx_END_INDEX; if both are
21226. present, find logic operator which connects them and put it into
21227. MAIN_OP, else MAIN_OP := **. WHILE_INDEX and UNTIL_INDEX currently
21228. point to the corresponding keyword or are zero if the corresponding
21229. keyword is omitted or implied (due to error). */

21230. MAIN_OP := **
21231. WHILE_INDEX, UNTIL_INDEX := LASTOP /* As initial guess. */
21232. if LASTOP = ' AND' and SYSLIST(I) = 'FOREVER'
21233. then WHILE_PRESENT, UNTIL_PRESENT := false
21234. else if WHILE_PRESENT
21235. then TO UNTIL_PRESENT
21236. fi
21237. if WHILE_INDEX = WHILE_END_INDEX
21238. then /* One of the following was entered:
21239. "DO WHILE," or "DO UNTIL," */
21240. fi
21241. if UNTIL_INDEX < WHILE_INDEX
21242. then /* UNTIL is first: */
21243. UNDER_INDEX := I - 1 /* Point at end of UNTIL. */
21244. if SYSLIST(I) = ' AND' and # ' OR'
21245. then UNTIL_INDEX := I /* Error message will be printed later. */
21246. else /* WHILE is first:
21247. "DO WHILE," */
21248. I := UNTIL_INDEX - 1 /* Point at <and/or>,
21249. WHILE_INDEX := I /* Error message will be printed later. */
21250. if SYSLIST(I) = ' AND' and # ' OR'
21251. then /* WHILE is first:
21252. "DO WHILE," */
21253. I := WHILE_INDEX - 1 /* Point at <and/or>,
21254. WHILE_INDEX := I /* Error message will be printed later. */
21255. fi
21256. if WHILE_INDEX = WHILE_END_INDEX
21257. then /* One of the following was entered:
21258. "DO WHILE," or "DO UNTIL," */
21259. fi
21260. if UNTIL_INDEX = UNTIL_END_INDEX
21261. then /* One of the following was entered:
21262. "DO UNTIL," */
21263. note (8, "STRC2110 WHILE TEST IS VOID--IGNORED")
21264. WHILE_PRESENT := false
21265. fi
21266. if WHILE_INDEX = WHILE_END_INDEX
21267. then /* One of the following was entered:
21268. "DO WHILE," */
21269. note (8, "STRC2111 WHILE TEST IS VOID--IGNORED")
21270. WHILE_PRESENT := false
21271. fi
21272. else /* WHILE_PRESENT but not UNTIL_PRESENT. */
21273. if WHILE_INDEX = WHILE_END_INDEX /* Which is equal to LASTOP. */
21274. then /* "DO WHILE" with no other operands. */
21275. WHILE_PRESENT := false /* Ignore to get infinite DO. */
21276. fi
21277. else /* WHILE_PRESENT */
21278. fi
21279. if WHILE_INDEX = WHILE_END_INDEX
21280. then /* "DO WHILE," */
21281. fi
21282. if WHILE_INDEX = WHILE_END_INDEX
21283. then /* "DO UNTIL," */
21284. fi
21285. if WHILE_INDEX = WHILE_END_INDEX
21286. then /* "DO WHILE," */
21287. if SYSLIST(I) = ' OR'
21288. then MAIN_OP := ' OR'
21289. fi
21290. else
21291. if SYSLIST(I) = ' AND'
21292. then MAIN_OP := ' AND'
21293. notes (8, "STRC2110 LOGIC OPERATOR BETWEEN "WHILE" AND "UNTIL"--"CMITTED--"AND" ASSUMED")
21294. fi
21295. fi
21296. fi
21297. fi
21298. fi
21299. fi
21300. /*
```
/* DO Macro -- 21 June 1973 */

21102. proc DO_LOOPING_BRANCH_AND_FIRST_OPERAND

21103. /* Step WHILE_INDEX and UNTIL_INDEX to first operand of conditional
21104. test or set to zero if not present. Collect all looping branch
21105. information. Set WHILE_ and UNTIL_COND_TEST to true if appropriate
21106. conditional test is present (as opposed to only a looping branch). */

21107. /* Assume no looping branch. */

21108. LB := 0

21109. LOOPING_BRANCH_TYPE := 'NCNE'

21110. if UNTIL_PRESENT

21111. then

21112. I, UNTIL_INDEX := UNTIL_INDEX + 1

21113. /* Move UNTIL_INDEX from pointing at "UNTIL" to pointing at first
21114. operand. */

21115. if SYSLIST(I,1) = 'BCT' or = 'BXLE' or = 'BXH'

21116. then

21117. LOOPING_BRANCH_TYPE := 'UNTIL'

21118. UNTIL_COND_TEST := (UNTIL_END_INDEX > I)

21119. /* UNTIL_INDEX is still pointing at the looping branch; we aren't
21120. sure how far to advance it yet. */

21121. else

21122. UNTIL_COND_TEST := false

21123. /* Turn off all UNTIL stuff. */

21124. fi

21125. if WHILE_PRESENT

21126. then

21127. I, WHILE_INDEX := WHILE_INDEX + 1

21128. /* Move WHILE_INDEX from pointing at "WHILE" to pointing at first
21129. operand. */

21130. if SYSLIST(I,1) = 'BCT' or = 'BXLE' or = 'BXH'

21131. then

21132. LOOPING_BRANCH_TYPE := 'WHILE'

21133. WHILE_COND_ANS := true

21134. /* WHILE_INDEX is still pointing at looping branch; we aren't sure
21135. how far to advance it yet. */

21136. if SYSLIST(I,1) = 'BCT'

21137. then

21138. endm (4, 'STRC2103 WARNING—"WHILE,(BCT,...) WILL LOOP ONE'
21139. 'LESS TIME THAN VALUE IN REGISTER')

21140. fi

21141. if MAIN_OP = 'OR'

21142. then

21143. endm (4, 'STRC2104 WARNING—LOOPING BRANCH MAY NOT BE '
21144. 'EXECUTED ON EVERY ITERATION')

21145. fi

21146. else /* There is also an UNTIL looping branch. */

21147. endm (4, 'STRC2105 TWO LOOPING BRANCHES INVALID IN "DO"—'
21148. 'WHILE IGNORED')

21149. WHILE_PRESENT, WHILE_COND_TEST := false

21150. MAIN_OP := 'I'

21151. /* WHILE_INDEX now pointing at looping branch operands. */

21152. fi

21153. if WHILE_COND_TEST := true

21154. /* Collect looping branch information and advance WHILE_ or
21155. UNTIL_INDEX over looping branch operands. */

21156. call DO_LOOPING_BRANCH_PROCESS
PLOG DOLOOPING_BRANCH_PROCESS

/* Collect looping branch information:

LB_OPCODE_ID  'BCT', 'BXL' or 'BXH'
LB_OPERAND1  First operand of branch operand
LB_OPERAND2  Second operand, null (or garbage) for BCT
LB_LOGIC_OP  Logic operator connecting looping branch to
rest of WHILE or UNTIL.
LB_LABEL_REQ  Indicates whether looping branch will need
a label.

Also step WHILE or UNTIL_INDEX over looping branch. If any looping
branch is present, LB contains its index; else, LB = 0. */

int OP_COUNT  /* Number of operands looping branch needs. */

LB_OPCODE_ID, LB_OPERAND1, LB_OPERAND2, LB_LOGIC_OP := **
/* Assume no looping branch is present. */

if LB = 0
   then
      LB_OPCODE_ID := SYSLIST(LB,1)
      if LB_OPCODE_ID = 'BCT'
      then
         OP_COUNT := 2
      else
         OP_COUNT := 3
      fi
   else
      LB_OPCODE_ID := SYSLIST(LB,2)
      LB_OPCODE_ID := SYSLIST(LB,3)
      if N'SYSLIST(LB) = CP_COUNT
      then
         /* Not a sublist */
      else
         /* Given as a sublist. */
         LE_OPERAND1 := SYSLIST(LB,2)
         LB_OPERAND2 := SYSLIST(LB,3)
         if N'SYSLIST(LB) = CP_COUNT
         then
            /* Set xxxxx_INDEX to point at start of conditional test, if any. */
         else
            /* Step LB past logic operator. */
         fi
      fi
      fi
   fi
   fi
fi

if LOOPING_BRANCH_TYPE = 'WHILE'
   then
      WHILE_INDEX := LB
   else
      UNTIL_INDEX := LB
   fi
/* Set xxxxx_INDEX to point at start of conditional test, if any. */

LB_LABEL_REQ := (LOOPING_BRANCH_TYPE = 'WHILE')
DO Macro -- 21 June 1973

21436. PROC DO_SET_FORMAT
21437. /* Set OPERAND_FORMAT according to decision table (see documentation). */
21438.
21439. if WHILE_PRESENT
21440. then
21441. if LOOPING_BRANCH_TYPE = 'WHILE'
21442. then
21443. if WHILE_COND_TEST
21444. then
21445. if LB_LOGIC_OP = 'AND'
21446. then
21447. if UNTIL_PRESENT
21448. then
21449. if MAIN_OP = 'AND'
21450. then
21451. OPERAND_FORMAT := 15
21452. else
21453. OPERAND_FORMAT := 16
21454. fi
21455. else
21456. OPERAND_FORMAT := 14
21457. fi
21458. else
21459. if UNTIL_PRESENT
21460. then
21461. if MAIN_OP = 'AND'
21462. then
21463. OPERAND_FORMAT := 17
21464. else
21465. OPERAND_FORMAT := 18
21466. fi
21467. else
21468. OPERAND_FORMAT := 11
21469. fi
21470. fi
21471. else
21472. if UNTIL_PRESENT
21473. then
21474. if MAIN_OP = 'AND'
21475. then
21476. OPERAND_FORMAT := 11
21477. else
21478. OPERAND_FORMAT := 12
21479. fi
21480. else
21481. OPERAND_FORMAT := 10
21482. fi
21483. fi
21484. fi
/* if WHILE_PRESENT then */
if LOOPING_BRANCH_TYPE = 'WHILE' then *** */

else
if LOOPING_BRANCH_TYPE = 'UNTIL' then
UNTIL_COND_TEST
if LD_LOGIC_OP = 'AND'
if MAIN_OP = 'AND'
OPERAND_FORMAT := 7
else
OPERAND_FORMAT := 9
fi
else
if MAIN_OP = 'AND'
then
OPERAND_FORMAT := 6
else
OPERAND_FORMAT := 8
fi
fi
else
if UNTIL_PRESENT then
if MAIN_OP = 'AND'
then
OPERAND_FORMAT := 2
else
OPERAND_FORMAT := 3
fi
else
OPERAND_FORMAT := 1
fi
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_LOGIC_OP = 'AND'
21543.                then
21544.                  OPERAND_FORMAT := 22
21545.                  fi
21546.                  OPERAND_FORMAT := 21
21547.                fi
21548.            else
21549.              OPERAND_FORMAT := 19
21550.          fi
21551.  else
21552.    OPERAND_FORMAT := 20
21553.  fi
21554.  else
21555.    OPERAND_FORMAT := 0
21556.  fi
21557.  fi
21558. corp
DO Macro -- 21 June 1973

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. PROC OPERAND_FORMAT IFARY
21566. CASE (2,3,6-9)
21567. /* Branch around UNTIL conditional test to WHILE conditional test. */
21568. generate (LABEL || 'B' || BLOCK_LABEL_PREFIX || 'W')
21569. LABEL := ''
21570. GOTO
21571. CASE (10-18)
21572. /* Branch to WHILE looping branch first. */
21573. generate (LABEL || 'B' || BLOCK_LABEL_PREFIX || 'LPB')
21574. LABEL := ''
21575. GOTO
21576. CASE (20,21,22)
21577. /* Branch around UNTIL conditional test to DO internal code. */
21578. generate (LABEL || 'B' || BLOCK_LABEL_PREFIX || 'REG')
21579. LABEL := ''
21580. GOTO
21581. CASE
21582. CODE
/* DO Macro -- 21 June 1973

PROC DO_GENERATE_ALL_CONDITIONAL_TESTS
   /* Cause WHILE and UNTIL conditional tests to be generated with proper
   labels. */
   int PASS /* Looping index */

   PASS := 1
   while PASS <= 2
   do
      if PASS = 1
         then
            call DO_UNTIL_PREPROCESS
            else
            call DO_WHILE_PREPROCESS
         endif

      /* The DO_xxxxx_PREPROCESS proc must set:
      TXTS_CONDITIONAL_REQD from xxxx_COND_TEST
      FIRST_INDEX from xxxx_INDEX
      LAST_INDEX from xxxx_END_INDEX
      UNIQUE_LABEL_ID with the first letter of xxxx
      ULTIMATE_BRANCH_LABEL with the branch target
      ULTIMATE_FALLTHRU_LABEL with the fallthru name
      ULTIMATE_FALLTHRU_CONDITION with the proper value
      FIRST_ID with the first label
      to insure proper test generation. */

      if THIS_CONDITIONAL_REQD
         then
            call DO_GENERATE_CONDITIONAL_SET
            /* Generate code to pass control to the ULTIMATE_FALLTHRU_LABEL (or
            to fall through to it) if the conditional test specified by
            SYSLIST(FIRST_INDEX) through SYSLIST(LAST_INDEX) has the logical
            value stored in ULTIMATE_FALLTHRU_CONDITION; else pass control to
            the ULTIMATE_BRANCH_LABEL. If a branch is generated to the
            ULTIMATE_FALLTHRU_LABEL, set FALLTHRU_LABEL_USED to true;
            else set it false. Include definition of any LABEL outstanding
            before generating code. */
         endif

      if PASS = 1
         then
            call DO_UNTIL_PCSTPROCESS
            /* For those cases where the ULTIMATE_FALLTHRU_LABEL was not to
            follow the conditional test as the next sequential instruction,
            generate an unconditional branch to the ULTIMATE_FALLTHRU_LABEL
            and clear FALLTHRU_LABEL_USED. */
         endif

      if FALLTHRU_LABEL_USED
         then
            LABEL := ULTIMATE_FALLTHRU_LABEL
            /* Generate label at next opportunity. */
         endif

      PASS := PASS + 1
   od /* (Termination: PASS incremented only (not modifiable by called
   procs), must eventually exceed 2.) */
"DO" MACRO -- 21 June 1973

```
21638. DECO DO_UNTIL_PREPROCESS
21639. /* Must set up THIS_CONDITIONAL_REQD, FIRST_INDEX, LAST_INDEX,
21640. UNIQUER LABEL_ID, ULTIMATE_BRANCH_LABEL, ULTIMATE_FALLTHRU_LABEL,
21641. ULTIMATE_FALLTHRU_CONDITION, and FIRST_ID. */
21642. THIS_CONDITIONAL_REQD := UNTIL_COND_TEST
21643. if UNTIL_COND_TEST
21644. then
21645. FIRST_INDEX := UNTIL_INDEX
21646. LAST_INDEX := UNTIL_END_INDEX
21647. ULTIMATE_BRANCH_LABEL := BLOCK_LABEL_PREFIX || 'END'
21648. /* Flow point 2 normally connects to flow point 12. */
21649. ULTIMATE_FALLTHRU_LABEL := BLOCK_LABEL_PREFIX || 'W1'
21650. /* Flow point 3 usually falls through to flow point 4. */
21651. ULTIMATE_FALLTHRU_CONDITION := false
21652. UNIQUE_LABEL_ID := 'U'
21653. FIRST_ID := BLOCK_LABEL_PREFIX || 'U1'
21654. do22ch OPERAND_FORMAT #amp
21655. of
21656. CASE (3,8,9) /* UNTIL test ORed with WHILE test. */
21657. ULTIMATE_FALLTHRU_CONDITION := true
21658. ULTIMATE_BRANCH_LABEL := BLOCK_LABEL_PREFIX || 'BEC'
21659. SEAC
21660. CASE (11,15,17) /* UNTIL test ANDed with WHILE looping branch. */
21661. ULTIMATE_FALLTHRU_LABEL := BLOCK_LABEL_PREFIX || 'LPB'
21662. SEAC
21663. CASE (12,16,18) /* UNTIL test ORed with WHILE looping branch. */
21664. ULTIMATE_BRANCH_LABEL := BLOCK_LABEL_PREFIX || 'LPB'
21665. ULTIMATE_FALLTHRU_LABEL := BLOCK_LABEL_PREFIX || 'BEC'
21666. SEAC
21667. CASE (20,21,22) /* UNTIL conditional test only. */
21668. ULTIMATE_FALLTHRU_LABEL := BLOCK_LABEL_PREFIX || 'BEC'
21669. SEAC
21670. ENDCASE
21671. ENDCASE
21672. END
21673. end
```

C-24
DO Macro -- 21 June 1973

21675. BEGIN 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. */
21679.  THIS_CONDITIONAL_REQD := WHILE_COND_TEST
21680.  if WHILE_COND_TEST
21681.     then
21682.       FIRST_INDEX := WHILE_INDEX
21683.       LAST_INDEX := WHILE_END_INDEX
21684.       ULTIMATE_BRANCH_LABEL := BLOCK_LABEL_PREFIX || 'END'
21685.       /* Flow point 5 always branches to flow point 12. */
21686.       ULTIMATE_FALLTHRU_LABEL := BLOCK_LABEL_PREFIX || 'BEG'
21687.       /* Flow point 6 always falls through to flow point 7. */
21688.       ULTIMATE_FALLTHRU_CONDITION := true
21689.       UNIQUE_LABEL_ID := 'W'
21690.       FIRST_ID := BLOCK_LABEL_PREFIX || 'W1'
21691.     fi
21692. fi
21693.  END
"DC" Macro -- 21 June 1973

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 /* No special label is required at the beginning of this conditional
21709. test. */
21710. FIRST_ID := LABEL /* Put the label on the conditional test. */
21711. else
21712. /* We also have a label waiting for the conditional test. */
21713. generate (LABEL [ ] " DS OH")
21714. /* Set the LABEL label out of the way. */
21715. fi
21716. fi
21717. fi
21718. call CONDITIONAL_EXPRESSION_PROCESSOR (FIRST_ID; SYSLIST)
21719. /* Generate code corresponding to the operands of the current set
21720. (WHILE or UNTIL) of the DO operands (referred to collectively as
21721. SYSLIST). Only the SYSLIST can be passed directly as arguments;
21722. the following variables are effectively arguments but are passed
21723. in global variables:
21724. FIRST_INDEX,
21725. LAST_INDEX,
21726. ULTIMATE_BRANCH_LABEL,
21727. ULTIMATE_FALLTHRU_LABEL,
21728. UNIQUE_LABEL_ID,
21729. FALLTHRU_LABEL_USED.
21730. Process operands of the SYSLIST beginning with SYSLIST (FIRST_INDEX)
21731. through SYSLIST (LAST_INDEX), generating the indicated tests to pass
21732. control as indicated above. If a branch is made to the
21733. ULTIMATE_FALLTHRU_LABEL, then FALLTHRU_LABEL_USED is set, else
21734. it is unaltered. */
21735. code
21736. code
DO Macro -- 21 June 1973

21738. ELOG 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. */

21744. dogase OPERAND_FORMAT ifany
21745. of
21746. case (11, 15-18)
21747. generate ("B \# ULTIMATE_FALLTHRU_LABEL")
21748. FALLTHRU_LABEL_USED := false
21749. esacd
21750. esacd
21751. orpcd
21752. core
'DO' Macro

PROC DO_LABEL BLOCK
/
/* If a begin label is required, generate it. */
/
CASE OPERAND FORMAT WHEN
CASE (0, 3, 5, 8-12, 14, 16-22)
CASE (LABEL # BLOCK_LABEL_PREFIX || 'BEG')
THEN
/* Begin label must be generated. */
CASE (LABEL '|
CASE (LABEL ' DS OH')
GENERIC (LABEL || ' DS OH')
CASE (LABEL := BLOCK_LABEL_PREFIX || 'BEG')
CASE
CASE
CASE
PROC
**DC Macro -- 21 June 1973**

```
21773. proc DC_INFO_SAVE
21774.   /* Insert into stack all information required to close loop at
21775.   TERMINATE_DO_LOOP. */
21776.   case B0, B10, B11
21777.   /* One character codes indicating flow point to follow points 8,
21778.   10, and 11. */
21779.   B8 := 'W' /* Assume branch at point 8 is to WHILE group (point 4). */
21780.   B10, B11 := 'O'
21781.   /* Assume no looping branch (thus no branches at 10 and 11). */
21782.   */ Set B8. */
21783.   dcsrc OPERAND_FORMAT around
21784.   of
21785.   case (2,3,11,12,15-18,20)
21786.   /* UNTIL conditional test but no UNTIL looping branch. */
21787.   B8 := 'O' /* To flow point 1. */
21788.   esac
21789.   case (6-10,13,14,19,21,22)
21790.   /* UNTIL looping branch or no UNTIL but WHILE looping branch. */
21791.   B8 := 'L' /* Fall through to point 9. */
21792.   esac
21793.   case (8,9,17,18)
21794.   /* Infinite loop. */
21795.   B8 := 'O' /* To flow point 7. */
21796.   esac
21797.   esac
21798.   esac
21799.   if LOOPING_BRANCH_TYPE = 'NONE'
21800.   then
21801.   /* Set B10. */
21802.   dcsrc OPERAND_FORMAT only
21803.   of
21804.   case (4,7,13,15,16)
21805.   B10 := 'W' /* To flow point 4. */
21806.   esac
21807.   case (6,10-13,15,16,19,21)
21808.   B10 := 'B' /* To flow point 7. */
21809.   esac
21810.   case (6,6,21)
21811.   B10 := 'O' /* To flow point 1. */
21812.   esac
21813.   esac
21814.   /* Set B11. */
21815.   dcsrc OPERAND_FORMAT only
21816.   of
21817.   case (4,6-10,13,15,16,19,21)
21818.   B11 := 'W' /* Fall through to flow point 12 (end of DO block). */
21819.   esac
21820.   case (5,6,14,17,18)
21821.   B11 := 'W' /* To flow point 8. */
21822.   esac
21823.   case (7,9,22)
21824.   B11 := 'O' /* To flow point 1. */
21825.   esac
21826.   esac
21827.   esac
21828.   fff
21829.   INFORMATION(CURRENT_DO_LEVEL) := B8 || B10 || B11 || LD_LABEL_REQ ||
21830.   false || false || false || false
21831.   /* Byte 5 is set true when the loop is terminated (by ATEND, ONEXIT, or DO). */
21832.   Byte 6 is set true when an ATEND occurs for this DC.
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 DO code. */
21835.   OPERAND1(CURRENT_DO_LEVEL) := LD_OPERAND1
21836.   OPERAND2(CURRENT_DO_LEVEL) := LD_OPERAND2
21837.   OPERAND3(CURRENT_DO_LEVEL) := LD_OPCODE_ID
21838.  endp
```
"DOm" Macro -- 21 June 1973

21840. **PROC DO_TRACE_COUNTERS
21841.      /* If debugging, generate block name and/or counters for block and
21842.      loop execution. */
21843.
21844. if DEBUG_BLOCKCOUNTS_REQD or DEBUG_BLOCKNAMES_REQD
21845.  then
21846.    if DEBUG_BLOCKCOUNTS_REQD
21847.      then
21848.          generate (LABEL || ' LA ', || BLOCK_LABEL_PREFIX || 'DOL')
21849.          LABEL := ''
21850.          generate (' ', LA 1,1(1)' ')
21851.          generate (' ', STH 1,1 '' || BLOCK_LABEL_PREFIX || 'DOL')
21852.          generate (' ', LA 1,1(1)'' ')
21853.          generate (' ', STH 1,1 '' || BLOCK_LABEL_PREFIX || 'DTR')
21854.          fi
21855.      /* Generate branch around block name and/or block counts. */
21856.      generate (LABEL || ' ', || BLOCK_LABEL_PREFIX || 'GO')
21857.      LABEL := BLOCK_LABEL_PREFIX || 'GO'
21858.
21859. if DEBUG_BLOCKNAMES_REQD
21860.  then
21861.      generate (' ', DC C'' || BLOCK_NAME(CURRENT_NEST_LEVEL) ||
21862.        '40' ||
21863.      fi
21864. if DEBUG_BLOCKCOUNTS_REQD
21865.  then
21866.      generate (BLOCK_LABEL_PREFIX || 'DOL DC W' || CURRENT_LOOP_COUNT)
21867.      generate (BLOCK_LABEL_PREFIX || 'DTR DC W' || OVERALL_LOOP_COUNT)
21868.  fi
21869.
21870. **end.
ATEND Macro -- 31 October 1973

```c
#include <stdio.h>

#define ATEND ( ; USER_NAME)
/** The ATEND macro causes the generation of the loop-terminating code
for the surrounding DO block if such code has not yet been generated.
The target for normal loop termination is then defined to allow the
code which follows to be executed at the end of normal loop termination. If
the ATEND has been preceded by an ONEXIT macro, the branch is generated
to the OD for the ONEXIT block. */

#define ATEND_GENNED, /* Indicates whether ATEND has been generated previously
for this block. */
#define TDL_GENNED, /* Indicates whether the TERMINATE_DO_LOOP macro has
been invoked for this DO by a previous macro (properly, only
by an ONEXIT). */
#define FIN_LABEL_REQD /* Indicates a branch to the label "FIND" has
been generated and must be defined at OD time. */

char
INFO /* Holds copy of INFORMATION(CURRENT_NEST_LEVEL). */

if CURRENT_NEST_LEVEL > NESTING_LIMIT
then
  mexit
fi

if ERROR_OCCURRED
then
  mexit
fi

INFO := INFORMATION(CURRENT_NEST_LEVEL)

CALL VERIFY_END ( ; 'DO', USER_NAME)
/** Verifies current block has the name specified by the USER_NAME
operand on the ATEND macro (if any) and that it is a DO block.
Various errors receive messages and either intermediate blocks are
BLENDed as a fixup or ERROR_OCCURRED is set. */

if ERROR_OCCURRED
then
  mexit
fi

INFO := INFORMATION(CURRENT_NEST_LEVEL)

ATEND_GENNED := INFO[6,1]
/** See if we've already generated an ATEND. */

if ATEND_GENNED
then
  note (B, "STRC2301 MORE THAN ONE "ATEND" IS BLOCK")
  mexit
fi

BLOCK_LABEL_PREFIX := 'F' || BLOCK_NAMES(CURRENT_NEST_LEVEL)
TDL_GENNED := INFO[5,1]
/** See if we've already generated the loop-terminating code. */

FIN_LABEL_REQD := INFO[8,1]
/** Note whether a FIN label has already been referenced. */

if ~ TDL_GENNED
then
  CALL TERMINATE_DO_LOOP ( ;
/** Terminate the loop by generating any necessary back branches. */
else
  /* TERMINATE_DO LOOP must have been done by previous ONEXIT. */
  GENERATE(| B 1| BLOCK_LABEL_PREFIX || 'FIN')
  FIN_LABEL_REQD := TRUE
  /* Terminate the ONEXIT block. */
fi

if END_LABEL_REQD(CURRENT_NEST_LEVEL)
then
  GENERATE(BLOCK_LABEL_PREFIX || 'END DS OD')
  END_LABEL_REQD(CURRENT_NEST_LEVEL) := FALSE
  /* If normal block termination required an END label, provide it and
note that we no longer require it. */
fi

INFORMATION(CURRENT_NEST_LEVEL) := INFO[1,4] ||

FILE */ TDL has not been generated. */ ||
INFO['ATEND has not been generated. */ ||
INFO[7,1] ||
FIN_LABEL_REQD /* Forward FIN_LABEL_REQD to OD. */

mend

C-31
```
"ONEXIT" Macro -- 31 October 1973

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.
25009. bit 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 "FIN" has
25015. been generated and must be defined at OD time. */
25016.
25017. char INFO /* Holds copy of INFORMATION(CURRENT_NEST_LEVEL). */
25018.
25019. call TRACE_PRINT(( ; 'ONEXIT')
25020. /* Prints macro name "ONEXIT" in mnote if tracing on. */
25021.
25022. if CURRENT_NEST_LEVEL > NESTING_LIMIT then
25023. mexit
25024. fi
25025.
25026. call VERIFY_END ( ; 'DO' USER_NAME)
25027. /* Verifies current block has the name specified
25028. by the USER_NAME operand of the ONEXIT macro (if any) and that it is a DO block.
25029. Various errors receive messages and either intermediate blocks are
25030. BLENDED as a fixup or ERROR_OCCURRED is set. */
25031. if ERROR_OCCURRED then
25032. mexit
25033. fi
25034.
25035. mnote (8, 'SRC2501 MORE THAN ONE "ONEXIT" IN BLOCK')
25036. mexit
25037.
25038. if -EXIT_LABEL_REQD(CURRENT_NEST_LEVEL) then
25039. mnote (8, 'SRC2502 NO EXIT FOR THIS "DO"')
25040. mexit
25041. fi
25042.
25043. FIRLABEL_BEQD := INFO[8,1]
25044. /* Note whether a FIN label has already been referenced. */
25045. BLOCK_LABEL_PREFIX := IS'BLOCK_NUMBER(CURRENT_NEST_LEVEL)
25046. TDL_GENNED := INFO[5,1]
25047. /* See if we've already generated the loop-terminating code. */
25048. if -TDL_GENNED then
25049. call TERMINATE_DO_LOOP ( ; )
25050. if TDL_FALLTHRU_OCCURS then /* Looping branch expects to fall through to END label. */
25051. generate ('B' || BLOCKLABELPREFIX || 'END')
25052. else /* TERMINATE_DOLOOP must have been done by previous ATEND. */
25053. generate ('B' || BLOCKLABELPREFIX || 'FIN')
25054. FIN_LABEL_REQD := true
25055. /* Provide branch to FIN for ATEND block. */
25056.
25057. generate (BLOCKLABELPREFIX || 'EXIT DS OD')
25058. EXIT_LABEL_REQD(CURRENT_NEST_LEVEL) := false
25059. /* Provide target for EXIT branch and note that it is no longer
25060. needed. */
25061. INFORMATION(CURRENT_NEST_LEVEL) := INFO[1,4] ||
25062. true /* TDL has now been generated. */
25063. INFO[6,1] :=
25064. false /* ONEXIT has now been generated. */
25065. FINLABEL_REQD := true
25066. /* Forward FIN_LABEL_REQD to OD */
25067.
25068. mend

C-32
Macro -- 26 June 1973

27001. }macro OD ( ; USER_NAME)
27002. /* Terminate DO loop if ATEND or ONEXIT have not done so and do
27003. standard block closing. */
27004. }
27005. TDL_GENNED, /* Indicates whether the looping code has been generated
27006. yet. */
27007. FIN_LABEL_REQD /* Indicates whether a "$nnnFIN" label is
27008. required. (It is used at the end of the ATEND or ONEXIT code.) */
27009. info /* Holds a copy of INFORMATION(CURRENT_NEST_LEVEL). */
27010. call TRACE_PRINT ( ; 'OD')
27011. /* Prints macro name "OD" in mnote if tracing on. */
27012. if CURRENT_NEST_LEVEL = NESTING_LIMIT
27013. then
27014. endif
27015. info := INFORMATION(CURRENT_NEST_LEVEL)
27016. TDL_GENNED := INFO[5,1]
27017. FIN_LABEL_REQD := INFO[8,1]
27018. if ~ TDL_GENNED
27019. then
27020. call TERMINATE_DO_LOOP ( ;
27021. /* Call separate macro to generate loop-terminating branches. */
27022. (Lem: TERMINATE_DO_LOOP does not modify CURRENT_NEST_LEVEL.) */
27023. else
27024. /* ATEND or ONEXIT occurred; we may need FIN label. */
27025. if FIN_LABEL_REQD
27026. then
27027. generate ('$' II BLOCK_NUMBER(CURRENT_NEST_LEVEL) II 'FIN DS OH')
27028. fi
27029. endif
27030. endif
27031. fi
27032. fi
27033. endif
27034. call POP_OLD_BLOCK ( ;
27035. /* Delete current block from the stack. */
27036. (Lem: POP_OLD_BLOCK decrements CURRENT_NEST_LEVEL by exactly
27037. one.) */
27038. endif
27039. /* (Lem: If CURRENT_NEST_LEVEL > 0 and
27040. (USER_NAME = ' ' or BLOCK_NAME(CURRENT_NEST_LEVEL) = 'DO'
27041. at entry to OD, then
27042. 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.  
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.  
31016.  char
31017.  INDEX_REG, /* Name of register containing DOCASE index, if
31018.  in a register. */
31019.  INDEX_LENGTH, /* Length (or symbol indicating length) of index for
31020.  CHARCOMP operand. */
31021.  INDEX_TYPE, /* Type of DOCASE index: "R" register, "W" word (or no
31022.  index--CONDTEST type DOCASE), "H" halfword, or "B" byte (or
31023.  character string--CHARCOMP type DOCASE). */
31024.  CASE_FORMAT, /* Format of CASE macros to follow: "GENERAL" (branch
31025.  vector and/or symbolic compares with index), "SPARSE" (symbolic
31026.  compares only), "CHARCOMP" (character string compares), "SIMPLE"
31027.  (short sequence of integers in order 1, 2, 3, ..., or
31028.  "CONDTEST" (no index on DOCASE, conditional test on each CASE
31029.  macro). */
31030.  LABEL, /* Any outstanding label, to be generated on next executable
31031.  instruction. */
31032.  INDEX_ADDR /* Symbolic address of byte-type or CHARCOMP index, if
31033.  any; else null. */
31034.  
31035.  /* Ground rules: LABEL is to be generated on the first of any
31036.  executable instruction sequence and then cleared to null; any label
31037.  which needs to be so generated may replace a null LABEL. BRANCH_TO_
31038.  CASE1 must be set by any branch directly or indirectly to the first
31039.  CASE (i.e., by all but falling through to the first CASE). */

C-34
**DOCASE** Macro -- 26 June 1973

```assembly
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. EXIT
31051. IF LABEL € USER_NAME
31052. /* Generate macro's label at first opportunity. */
31053. CALL DOCASE_EXTRACT_OPERANDS
31054. /* Validate operands and issue any error-messages; set INDEX_RANGE, 
31055. INDEX_TYPE, INDEX_RANGE_ASSURED, INDEX_LENGTH and CASE_FORMAT. */
31056. IF CASE_FORMAT = 'CONDTEST'
31057. THEN
31058. CALL DOCASE_INDEXTO_REG
31059. /* If case format is GENERAL, SPARSE, or CHARCOMP and the index is a 
31060. byte, save symbolic address of the index in INDEX_ADDR, otherwise 
31061. set INDEX_ADDR to null and generate code to put index into GPR1. */
31062. IF CASE_FORMAT = 'SIMPLE'
31063. THEN
31064. CALL DOCASE_GENERAL_SETUP
31065. /* Generate branch to general handler for GENERAL format. In any 
31066. case (GENERAL, SPARSE, or CHARCOMP), advance GCASE_NEST_LEVEL for the 
31067. GCASE stack and initialize the GCASE globals. */
31068. FI
31069. FI
31070. CALL DOCASE_DEBUGSTUFF
31071. /* Generate last-case variable and block-name constant if required. */
31072. CALL DOCASE_INFO_SAVE
31073. /* Store in stack all data needed by CASE and ESACOD to complete 
31074. case processing. */
31075. IF LABEL € ''
31076. THEN
31077. GENERATE (LABEL ' DS QN')
31078. FI
31079. FI
31080. END
```

C-35
N/A
Save symbolic address of the index in INDEX_ADDR, otherwise set INDEX_ADDR to null and generate code to put index into GPR1.

Given: This proc is not called for CCNGTEST format. */

INDEX_ADDR := "" /* Assume index will be stored in GPR1. */

DOCASE INDEX_TYPE ONLY

CASE ('E') /* Register index. */
    IF INDEX_REG # '1'
        generate (LABEL 11 ' LR 1,' || INDEX_REG)
    LABEL := ""
    fi
    esac

CASE ('W') /* Word index. */
    generate (LABEL 11 ' 1,' || INDEX(1))
    LABEL := ""
    esac

CASE ('H') /* Halfword index. */
    generate (LABEL 11 ' 1,' || INDEX(1))
    LABEL := ""
    esac

CASE ('B') /* Byte index. */
    IF CASE_FORMAT = 'SIMPLE'
        generate (LABEL 11 ' 1,' || INDEX(1))
        LABEL := ""
        else
    INDEX_ADDR := INDEX(1)
    fi
    esac

/* Postpone loading of index into register; we may want to do CLI's. */
The text appears to be a section of a programming manual or documentation, specifically discussing a macro named `OCASE`. The macro is used to handle different case formats in a programming context. The code snippet includes conditions to generate branches and advance stack levels for handling different case formats.

```c
/* Generate branch to beginning of general handler for general format */
int I, J /* Temporaries. */

if CASE_FORMAT == 'GENERAL'
    then
        generate (LABEL || 'B' || BLOCK_LABEL_PREFIX || 'BIG') /* Generate branch to general handler which is defined at ESACOD. */
        LABEL := **
        BRANCH_TO_CASE1 := ** /* Alternatively. */
    fi

if CASE_FORMAT == 'GENERAL'
    then
        MAX_CASE_VALUE(GCASE_NEST_LEVEL) := -1 /* Maximum branch vector value found. */
        NEXT_COMP_LABEL_NO(GCASE_NEST_LEVEL) := 1 /* Case number for next comparison case label to be generated. */
        J := GCASE_NEST_LEVEL * 256
    fi

if CASE_FORMAT == 'GENERAL'
    then
        CASE_OCCURS(I) := false
        I := I + 1 /* (Continuation: I is incremented, J is fixed during loop; I must eventually exceed J.) */
    fi

else /* GCASE stack overflow. */
    mnote (12, 'STRC3104 GENERAL/SPARSE/CHARCOMP GCASE NESTING LEVEL ' GCASE_NEST_LEVEL ' EXCEEDS MAXIMUM OF ' GCASE_NEST_LIMIT ' -- MACROS MUST BE MODIFIED')
    fi
```

The code is structured to handle different case formats and manage stack levels appropriately. It includes conditional logic to generate specific branches and manage variables appropriately to ensure correct operation under different case scenarios.
"DOCASE" Macro -- 26 June 1973

```c
31226. #DEF DOCASE_DEBUG_STUFF
31227. /* Generates last-case variable and block-name constant if required. */
31228. char X /* Temporary. */
31229. if DEBUG_BLOCKCOUNTS_REQD OR DEBUG_BLOCKNAMES_REQD
31230. then
31231. if ~ BRANCH_TO_CASE1
31232. then
31233. /* Branch must be generated around the last-case variable and/or
31234. block-name constant. Put target suffix into X. */
31235. if CASE_FORMAT = 'GENERAL'
31236. then
31237. /* This case should not occur since DOCASE_GENERAL_SETUP generates
31238. the branch for GENERAL cases; we include the code here for
31239. completeness. */
31240. X := 'BEG'
31241. else
31242. X := 'C1'
31243. fi
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_NESTLEVEL) || """)
31252. fi
31253. if DEBUG_BLOCKCOUNTS_REQD
31254. then
31255. generate (BLOCK_LABEL_PREFIX || "LSC DC X'00' LAST CASE NUMBER")
31256. fi
31257. fi
31258. endif
```

C-39
PROC DCASE_INFO_SAVE
    /* Store the case counter initial value (0) in OPERAND1; INDEX_ADDR
       in OPERAND2; CASE_FORMAT in OPERAND3; INDEX_LENGTH in OPERAND4; and
       various switches in INFORMATION. */

    OPERAND1(CURRENT_NEST_LEVEL) := '0' /* Case counter. */
    OPERAND2(CURRENT_NEST_LEVEL) := INDEX_ADDR /* Byte index address. */
    OPERAND3(CURRENT_NEST_LEVEL) := CASE_FORMAT
    OPERAND4(CURRENT_NEST_LEVEL) := INDEX_LENGTH
    INFORMATION(CURRENT_NEST_LEVEL) :=
        BRANCH_TO_CASE1 // CASE 1 11 false 11 1 false 11 INDEX_RANGE_ASSURED)

    /* Information:
       Byte 1: Indicates whether first CASE requires a label.
       Byte 2: Indicates whether a MISC CASE has been found.
       Byte 3: Indicates whether all self-defined operands are divisible
               by 4.
       Byte 4: Indicates whether any unexpected operands were found
               for general case processing (i.e., any operands which were not
               equal to their own sequential CASE number).
       Byte 5: Indicates whether index test for out-of-range value may be
               omitted. */

C-40
"CASE" Macro -- 27 June 1973

3301. `CASE` (USER_NAME; REL=, NASK)
3302. /* The CASE macro is used to specify a block of code which is one
3303. of the alternatives for the immediately surrounding DOCASE macro. If
3304. CASE macro is not the immediate daughter of a DOCASE and no fixup is
3305. possible, a BLOCK macro is substituted. Otherwise, the information
3306. stored by the DOCASE is extracted and the operands of the CASE are
3307. processed to produce the necessary code for the selecting of this
3308. block in the indicated case. Finally, any debugging code required
3309. is generated. */
3310. int
3311. CASE_COUNTER, /* Case number for this CASE maintained in
3312. mother info. */
3313. COMP_LABEL_NO, /* Number to be used in next compare label to be
3314. defined. */
3315. I /* SYSLIST index. */
3316. bit
3317. CASE_LABEL_REQD, /* true unless DOCASE is falling through into
3318. first CASE. */
3319. INDEX_RANGE_ASSURED, /* true if we have been assured (by
3320. "DOCASE ---,ONLY") that no values other than those specified
3321. by CASE operands will occur. */
3322. EQUAL_TEST_OUTSTANDING,
3323. /* Indicates that a compare for the current operand has been generated
3324. but the "ST" to the beginning of the block (or "BE" around the
3325. block) has not been generated yet. */
3326. RANGE_TEST_OUTSTANDING,
3327. /* Indicates that a compare for the current range operand has been
3328. generated as well as the branch if below the range; the branch
3329. if within the range to the beginning of the block (or "BH" around
3330. the block) has not been generated yet. */
3331. MISC_FOUND, /* Indicates whether MISC has been found yet. */
3332. MULTIPLESOF4, /* Indicates whether all the self-defining operands
3333. of the CASE macros processed so far are multiples of 4. */
3334. UNEXPECTED_OPERANDS_FOUND /* Indicates whether any operands have
3335. been found so far in the CASE macros' operands which either were
3336. symbolic or were self-definers not equal to their own case number. */
3337. char
3338. CASE_FORMAT, /* Type of CASE operands expected: GENERAL,
3339. SPARSE, SIMPLE, CHARCOMP, or CONDTEST. */
3340. MAMA_BLOCK_PREFIX, /* BLOCK_LABEL_PREFIX from mother DOCASE block. */
3341. INDEX_ADDR, /* Symbolic address of byte or CHARCOMP operand. */
3342. LABEL, /* Outstanding label waiting to be generated. */
3343. NEXT_CASE, /* Label to be generated on next SIMPLE or CONDTEST
3344. CASE macro. */
3345. INDEX_LENGTH /* Length of CHARCOMP index. */
"CASE* Macro -- 27 June 1973

33046. call TRACE_PRINTER ( ; "CASE")
33047. /* Print macro name "CASE" in anote if tracing on. */
33050. call CASE_POSITION_CHECK
33051. /* Verifies mother block is a DOCASE or attempts fixup with up to 2 BLENDs. Indicates whether un-fixup-able ERROR_OCCURRED. */
33053. if ERROR_OCCURRED then
33054. note (8, "STBC3304 "CASE" TREATED AS "BLOCK" MACRO")
33055. call BLOCK (USER_NAME;)
33056. note
33058. call PUSH_NEW_BLOCK (USER_NAME; BLOCK_TYPE_VALUE='CASE')
33059. /* Define new block; add to stack. Initialize block specifications. Note block type. Set up unique BLOCK_LABEL_PREFIX for use in generating unique labels. */
33060. if ERROR_OCCURRED /* during PUSH_NEW_BLOCK (viz., stack overflow) */ then
33061. mexit
33062. fi
33064. fi
33065. fi
33067. fi
33068. fi
33070. call CASE_GET_DOCASE_INFO
33071. /* Extract CASE_FORMAT, CASE_LABEL_REQD, CASE_COUNTER, MISC_FOUND, MAMA_BLOCK_PREFIX, INDEX_RANGE_ASSUMED, INDEX_ADDR, and INDEX_LENGTH from mother DOCASE block. */
33073. if USER_NAME # then
33074. generate (USER_NAME II DS OH)
33075. /* Any USER_NAME on a CASE macro is just a comment since a branch to it will produce unpredictable results. If one was specified, get it out of the way now. */
33079. if SYSLIST(1) = 'MISC' then
33080. call CASE_MISC_PROCESS
33081. /* Completely process miscellaneous CASE block. */
33083. else
33084. if CASE_FORMAT = 'GENERAL' or = 'SPARSE' or = 'CHARCOMP'
33086. then
33088. if GCASE_NEST_LEVEL <= GCASE_NEST_LIMIT then
33090. call CASE_PROCESSCOMPAREOPERANDS
33092. /* Generate code to handle all "symbolic" operands (i.e., all those which cannot be handled with the branch vector), or for all operands in the SPARSE or CHARCOMP format. These are all handled by generating compare-and-branch sequences. */
33094. if CASE_FORMAT = 'GENERAL' then
33096. call CASE_PROCESSVECTOROPERANDS
33098. /* Generate labels and save information about any operands which are to be handled via branch vector. */
33101. fi
33103. else
33104. call CASE_SET_NAMES
33106. /* Set LABEL if label required on first of code (usually is; only exception is when DOCASE is falling through to first CASE macro). Set NEXT_CASE with label to be used on next case. */
33107. if CASE_FORMAT = 'SIMPLE'
33109. then
33111. call CASE_BCT_GEN
33113. /* Generate BCT instruction for this case. */
33115. else
33117. call CASE_CONDTEST_GEN
33119. /* Generate conditional test specified on CASE macro. */
33121. fi
33123. call CASE_TRACE_COUNTER
33125. /* Generate code to count this block, note last case number, and/or display block name if appropriate debugging requested. */
33127. if LABEL # then
33129. generate (LABEL II DS OH)
33131. send

C-42
/* Verifies 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
  note (8, 'STRC3301 "CASE" NOT IMMEDIATE LAUGHTER OF "DOCASE"')
  ERROR_OCCURRED := true /* Assume no fixup possible. */
  if CURRENT_NEST_LEVEL > 1 and BLOCK_TYPE(CURRENT_NEST_LEVEL-1) = 'DOCASE' then
    note (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
      note (8, 'STRC3303 ASSUMING TWO "BLENDs" OMITTED--INSERTED')
      call BLEND ( ; )
      call BLEND ( ; )
      ERROR_OCCURRED := false
    fi
  fi
fi

endif
"CASE" Macro — 27 June 1973

PROG CASE_GET_DOCASE_INFO

 /* Extract DOCASE information being maintained in mother's stack */

  char x

  ANS MON /* Temporaries. */

  NON := CURRENT_NEST_LEVEL - 1

  CASE_COUNTER := OPERAND1(NON)

  if STLIST(1) ≠ 'MISC'

  then

  CASE_COUNTER := CASE_COUNTER + 1

fi

 INDEX ADDRS := OPERAND2(NON)

 CASE_FORMAT := OPERAND3(NON)

 INDEX_LENGTH := OPERAND4(NON)

 x := INFORMATION(NON)

 CASE_LABEL_REQD := x[1, 1]

 MISC_FOUND := x[2, 1]

 MULTIPLESOF := x[3, 1]

 UNEXPECTED_OPERANDS_FOUND := x[4, 1]

 INDEX_RANGE_ASSUMED := x[5, 1]

 NAME BLOCK_PREFIX := "$" | BLOCK_NUMBER(NON)

 CODE
33182. CASEPROCESS_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. */
33190. I := 1 /* Start search with first operand. */
33191. COMP_LABEL_NO := NEXT_COMP_LABEL_NO(GCASE_NEST_LEVEL)
33192. /* Note the next compare label number. */
33193. EQUAL_TEST_OUTSTANDING, RANGE_TEST_OUTSTANDING := false
33194. while I <= N'SYSLIST
33195. do /* (CASE_FORMAT = 'SPARSE' or = 'CHARCOMP' or
33196. [N'SYSLIST(I) = 1 and (T'SYSLIST(I) = 'N')
33197. or (T'SYSLIST(I) = 1 and (T'SYSLIST(I,1) # 'N' or
33198. SYSLIST(I,1) < 0 or > 255)
33199. or (T'SYSLIST(I,2) # 'N' or
33200. SYSLIST(I,2) < 0 or > 255))
33201. then
33202. if EQUAL_TEST_OUTSTANDING
33203. then
33204. LABEL := BLOCK_LABEL_PREFIX || 'BEG'
33205. generate ('BE A ' || LABEL)
33206. /* After leaving this proc, someone will generate the BEG label
33207. at the beginning of the block. */
33208. EQUAL_TEST_OUTSTANDING := false
33209. else
33210. if RANGE_TEST_OUTSTANDING
33211. then
33212. LABEL := BLOCK_LABEL_PREFIX || 'BEG'
33213. generate ('BNE ' || LABEL)
33214. /* Again, by leaving BEG label in LABEL, it will be generated
33215. after leaving this proc. */
33216. RANGE_TEST_OUTSTANDING := false
33217. else
33218. if CASE_LABEL_REQD
33219. then
33220. COMP_LABEL := MAMA_BLOCK_PREFIX || 'C' || COMP_LABEL_NO
33221. /* Generate label name to be attached to first instruction. */
33222. fi
33223. fi
33224. fi
33225. fi
33226. fi
33227. fi
33228. call CASEGENCOMPARE
33229. /* Generate compare for the single compare operand at SYSLIST(I) —
33230. either general case non-self-definer or any SPARSE or CHARCOMP
33231. operand. GCASE index is at ADDR_ADDR unless that's null, then is
33232. GEN1. Length is in INDEX_LENGTH for CHARCOMP type. Any label to be
33233. generated is in COMP_LABEL; once defined, COMP_LABEL_NO must be
33234. increased. Any branch target outstanding at exit is to be
33235. put into COMP_LABEL. Also on exit, EQUAL_TEST_OUTSTANDING or
33236. RANGE_TEST_OUTSTANDING should be set to indicate which type of
33237. operand was processed. */
33238. fi
33239. I := I + 1 /* Advance to next operand of CASE. */
33240. od /* (Termination: I is incremented above and not modified by
33241. called proc; N'SYSLIST is fixed; I must eventually exceed
33242. N'SYSLIST.'] */
33243. if EQUAL_TEST_OUTSTANDING
33244. then
33245. /* Generate branch to next symbolic case. */
33246. generate ('BNE ' || MAMA_BLOCK_PREFIX || 'C' ||
33247. COMP_LABEL_NO)
33248. else
33249. if RANGE_TEST_OUTSTANDING
33250. then
33251. generate ('BN '|| COMP_LABEL)
33252. /* Generate branch to next compare case. Label was left in COMP_LABEL
33253. when we branched on lower end of range. */
33254. fi
33255. NEXT_COMP_LABEL_NO(GCASE_NEST_LEVEL) := COMP_LABEL_NO
33256. /* Store case number of next symbolic case to be defined. */
"CASE" Macro -- 27 June 1973

33261. \texttt{PROC CASE GEN_COMPARE} *
33262. \texttt{\textsl{/* Generate compare for the single compare operand at SYSLIST(I) —}}
33263. \texttt{either general case non-self-definer or any SPARSE or CHARCOMP}
33264. \texttt{operand. DOCASE index is at INDEXADDR unless that's null, then in}
33265. \texttt{GPR1. Length is in INDEX_LENGTH for CHARCOMP type. Any label to be}
33266. \texttt{generated is in COMP_LABEL; any branch target at exit is to be}
33267. \texttt{put into COMP_LABEL. Also on exit, EQUAL_TEST_OUTSTANDING or}
33268. \texttt{RANGE_TEST_OUTSTANDING should be set to indicate which type of}
33269. \texttt{operand was processed. Operands may be of the form g or (g, h),}
33270. \texttt{the latter implying the range from g to h. g and h may be}
33271. \texttt{self-defining terms or symbols \texttt{EQU}ated to absolute expressions for}
33272. \texttt{GENERAL or SPARSE format; for CHARCOMP, they may be absolute or}
33273. \texttt{symbolic addresses of character strings or may be literals (with the}
33274. \texttt{leading \"=\" and, for character literals, \"C\" possibly omitted). */}
33275. \texttt{CHAR INSERT} /* Temporary, */
33276. \texttt{CASE GEN_COMPARE} *
```
"CASE" macro -- 27 June 1973

if INDEX_ADDR = ""
  then /* Index is in GPR1. */
    generate (COMP_LABEL || ' LA 0,' || SYSLIST(I,1))
else /* Index is at INDEX_ADDR. */
  if CASE_FORMAT = 'CHARCMP'
    then
      insert := 
      if SYSLIST(I,1)[1,1] # "
        then
          if SYSLIST(I,1)[1,1] = "*
            then /* Character string. */
              insert := =C'
          else
            if SYSLIST(I,1)[1,1] = ";
              then /* Literal without the "" (operand ends with "") */
                insert := "
          fi
        fi
      else
        generate (COMP_LABEL || ' CLC ',
          || INDEX_ADDR || ',' || INSERT || SYSLIST(I,1))
      fi
    else
      generate (COMP_LABEL || ' CLI ',
        || INDEX_ADDR || ', ' || SYSLIST(I,1))
  fi
else
  if COMP_LABEL = ""
    then
      COMP_LABEL := ""
      COMP_LABEL_NO := COMP_LABEL_NO + 1
  fi
  if SYSLIST(I) < 1
    then /* Operand is not a range. */
      EQUAL_TEST_OUTSTANDING := true
    else /* A range has been specified: (g,) */
      if SYSLIST(I) > 2
        then /* Generate another label. */
          RANGTESTOUTSTANDING := true
        else
          generate (" CLC ',
            || INDEX_ADDR || ',' || INSERT || SYSLIST(I,1))
        fi
      fi
    else
      if INDEX_ADDR = ""
        then
          generate (' LA 0,' || SYSLIST(I,2))
        else
          generate (' CR 1,0')
      fi
  fi
  if CASE_FORMAT = 'CHARCOMP'
    then
      insert := 
      /* Go through the same business figuring out the insert for a as
      we did for g. */
      if SYSLIST(I,2)[1,1] # "
        then
          if SYSLIST(I,2)[1,1] = "*
            then /* Literal without the "" (operand ends with "") */
              insert := "
          else
            if SYSLIST(I,2)[K'SYSLIST(I,2),1] = "*
              then /* Literal without the "" (operand ends with "") */
                insert := "
            fi
          fi
        fi
      else
        generate (' CLC ',
          || INDEX_ADDR || ', ' || INDEX_LENGTH ||
            ', ' || INSERT || SYSLIST(I,2))
      fi
    else
      generate (' CLI ',
        || INDEX_ADDR || ', ' || SYSLIST(I,2))
fi
```
```
"CASE" macro -- 27 June 1973

33356. BLOC CASE_PROCESS_VECTOR_OPERANDS
33357. /* Generate labels and note that CASE_OCCURS for any operands which
33358. can be handled via branch vector: viz., any of the form G or
33359. (G,M) where G and M are self-defining terms in the range
33360. 0-255 inclusive. This procedure assumes the CASE_FORMAT is general
33361. (not sparse). */
33362.
33363. int
33364. BASE, /* Array position in CASE_OCCURS of the case for zero. */
33365. OP, /* Case value currently being considered. */
33366. LIMIT /* High limit in range operands. */
33367.
33368. BASE := ((GCASE_NEST_LEVEL - 1) * 256) + 1
33369. /* Calculate offset in CASE_OCCURS array for this DOCASE. */
33370. if N'SYSLIST > 0 then /* One or more operands were specified. */
33371. I := 1 /* Start with first operand. */
33372. while I <= N'SYSLIST do
33373. if T'SYSLIST(I,1) = 'N' then /* g is a self-defining term. */
33374. OP := SYSLIST(I,1)
33375. if OP = 255 and OP = 0 then /* It's in the range. */
33376. if T'SYSLIST(I,2) = 'O' or = 'N' then /* n is self-defining or not present. */
33377. if N'SYSLIST(I) > 2 then /* No SYSLIST(I) */
33378. mnote (8, 'STRC3312 II SYSLIST(I) II INVALID--ONLY FIRST TWO SUBOPERANDS PROCESSED')
33379. else
33380. if T'SYSLIST(I,2) = 'N' then
33381. LIMIT := SYSLIST(I,2)
33382. if LIMIT > MAX_CASE_VALUE(GCASE_NEST_LEVEL) then /* We have found a new max case number. */
33383. MAX_CASE_VALUE(GCASE_NEST_LEVEL) := LIMIT
33384. if LIMIT > MAX_CASE_VALUE(GCASE_NEST_LEVEL) then /* Termination: OP is incremented, LIMIT is fixed; OP must eventually exceed LIMIT. */
33385. else
33386. do
33387. if T'SYSLIST(I,2) = 'N' then
33388. LIMIT := SYSLIST(I,2)
33389. if LIMIT > MAX_CASE_VALUE(GCASE_NEST_LEVEL) then /* We have found a new max case number. */
33390. if LIMIT < OP then /* g is in the right range also. */
33391. if N'SYSLIST(I) > 2 then /* No SYSLIST(I) */
33392. mnote (8, 'STRC3305 II SYSLIST(I) II INVALID--')
33393. LIMIT := OP /* ASSUMED */
33394. if N'SYSLIST(I) > 2 then
33395. if T'SYSLIST(I,2) = 'N' then
33396. LIMIT := OP
33397. if LIMIT > MAX_CASE_VALUE(GCASE_NEST_LEVEL) then /* We have found a new max case number. */
33398. MAX_CASE_VALUE(GCASE_NEST_LEVEL) := LIMIT
33399. if LIMIT > MAX_CASE_VALUE(GCASE_NEST_LEVEL) then /* Termination: OP is incremented, LIMIT is fixed; OP must eventually exceed LIMIT. */
33400. else
33401. do
33402. if OP = CASE_COUNTER then /* Unexpected OPERANDS found */
33403. if OP/4*4 = 0 then /* MultipleOP4 */
33404. if GP/MANUAL_BLK_RC_PREFIX = true then /* If all the CASE macros so far have had no operands or only */
33405. "expected" ones (integers which match the case counter), assume
33406. this one matches too and generate the single operand. */
33407. fi
33408. fi
33409. fi
33410. fi
33411. fi
33412. fi
33413. fi
33414. fi
33415. fi
33416. fi
33417. fi
33418. fi
33419. fi
33420. fi
33421. fi
33422. fi
33423. fi
33424. fi
33425. fi
33426. fi
33427. fi
33428. fi
33429. fi
33430. fi
33431. fi
33432. fi
```
/* Generate label for branch vector cases with no operands. Value used */

nt GUESS /* Assumed operand. */

GUESS := MAX_CASE_VALUE(GCASE_NEST_LEVEL) + 1 /* Guess at what omitted operand was intended. */

if GUESS ≤ 0 then /* First guess. */

GUESS := 1

fi

MAX_CASE_VALUE(GCASE_NEST_LEVEL) := GUESS

CASE_OCCURS(BASE+GUESS) := true

if UNEXPECTED_OPERANDS_FOUND then

/* STRC3306 EARLIER UNEXPECTED OPERAND IMPELS THIS TO BE CASE */

fi

C-49
"CASE" macro -- 27 June 1973

33459. BLOCK CASE_SET_NAMES
33460. /* Set LABEL if one will be required on this SIMPLE or CONDTETEST case
33461. code (usually is, only exception involves when DOCASE falls through
to first case). Also set NEXT_CASE with label of next case to be
generated. LABEL is always null at entry. */
33465. if CASE_LABEL_REQ
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. else
33473.      DDD CASE_BCT_GEN
33474.       /\* Generate BCT for this simple case. Verify operand, if any. */
33476.       generate (label i1 'bct 1' ii next_case)
33477.       label := ''
33478.       if 'syslist'(1) = '0'
33479.          then /* An operand was specified. */
33480.             if 'syslist'(1) = 'n'
33481.                then /* Operand is a self-defining term. */
33482.            if 'syslist'(1) = case_counter
33483.                then
33484.                    error (9, 'stnc:3327 operand invalid value of simple case' ii
33485.                      case_counter)
33486.            else /* Operand is not self-defining term. */
33487.                error (9, 'stnc:3309 operand must be self-defining term or omitted' ii
33488.                     case_counter)
33490.            fi
33491.      fi
33492.      end
PROC CASE_CONDTEST Gen

/* Generate conditional test indicated by operands. */
int OP_COUNT /* Number of operands for instruction being passed to SIMPLE_CONDITIONAL. */

ULTIMATE_BRANCH_LABEL := NEXT_CASE
ULTIMATE_FALLTHRU_LABEL := BLOCK_LABEL_PREFIX || 'BEG'
ULTIMATE_FALLTHRU_CONDITION := true
FALLTHRU_LABEL_USED := false
FIRST_INDEX := 1
LAST_INDEX := 'SYSLIST'

/* Process entire operand list as a single conditional expression. */
CASE_CONDITIONAL_EXPRESSION_PROCESSOR (LABEL; 'SYSLIST')
/* Generate code corresponding to the operands of the CASE macro 
(referred to collectively as 'SYSLIST'). Only the 'SYSLIST' can be passed 
directly as arguments; the following variables are effectively 
arguments but are passed in global variables: 
FIRST_INDEX, 
LAST_INDEX, 
ULTIMATE_BRANCH_LABEL, 
ULTIMATE_FALLTHRU_LABEL, 
ULTIMATE_FALLTHRU_CONDITION, 
UNIQUE_LABEL_ID, 
FALLTHRU_LABEL_USED. 
Process operands of the 'SYSLIST' beginning with 'SYSLIST(FIRST_INDEX)' 
through 'SYSLIST(LAST_INDEX)' (for the CASE macro, this is the entire 
'SYSLIST'), generating the indicated test to pass control to the 
ULTIMATE_FALLTHRU_LABEL if the test succeeds, else to the 
ULTIMATE_BRANCH_LABEL. The UNIQUE_LABEL_ID is used to insure 
unique labels where needed. If a branch is made to the 
ULTIMATE_FALLTHRU_LABEL, then set FALLTHRU_LABEL_USED; else 
it is unaltered. */

if FALLTHRU_LABEL_USED
  then
    LABEL := BLOCK_LABEL_PREFIX || 'BEG'
  else
    LABEL := 
fi

C-52
if ~ CASE_LABEL_REQD
  then
    generate ('B || MAMABLOCK_PREFIX || 'C || CASE_COUNTER)
  /* Generate branch to next case number (probably C1). */
  fi

if MISC_FOUND
  then
    note (8, 'STRC3311 MULTIPLE MISC CASES IN DOCASE--THIS BLOCK ' ||
           'IS DEAD CODE')
  else
    LABEL := MAMABLOCK_PREFIX || 'MSC'
    /* Make MSC label outstanding (generate on next instruction).
    It is assumed that no LABEL can be outstanding when CASE_MISC_PROCESS
    is called. */
    MISC_FOUND := true
  fi

if INDEX_RANGE_ASSURED
  then
    note (8, 'STRC3308 "DOCASE ... ONLY" INVALID WITH MISC')
  else
    INDEX_RANGE_ASSURED := false
  fi
CASE- Macro — 27 June 1973

33565. 코드 CASE_TRACE_COUNTER
33566. /* Generate any debugging counters and/or labels requested. */
33566. ndef DEBUG_BLOCKCOUNTS_REQD or DEBUG_BLOCKNAMES_REQD
33567. ndef DEBUG_BLOCKCOUNTS_REQD
33568. ndef 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.'
33576. generate (' STB 1,' || BLOCK_LABEL_PREFIX || 'CTR')
33577. if SYSLIST(1) = 'MISC' or CASE_COUNTER > 255
33578. then
33579. generate (' MVI ' || NAME_BLOCK_PREFIX || 'LSC,'X'FF')
33580. else
33581. HEX_IN := CASE_COUNTER
33582. call XHEX ( ; )
33583. generate ('"STBC3313 CASE DEBUG ID=X" || HEX_OUT || "")
33584. generate (' MVI ' || NAME_BLOCK_PREFIX || 'LSC,'X" ||
33585. HEX_OUT || "" CASE NUMBER FOR TRACING"
33586. fi
33587. generate (LABEL || ' B ' || BLOCK_LABEL_PREFIX || 'GO')
33588. LABEL := BLOCK_LABEL_PREFIX || 'GO'
33589. /* Search around count and/or block name and set up label to be defined eventually. */
33590. if DEBUG_BLOCKNAMES_REQD
33591. then
33592. generate ('="DC C" || BLOCK_NAME(CURRENT_NEST_LEVEL) ||
33593. DC ' || OH'0"
33594. if DEBUG_BLOCKCOUNTS_REQD
33595. then
33596. generate (BLOCK_LABEL_PREFIX || "CTR DC H'0' CASE COUNT")
33597. fi
33598. fi
33599. fi
33600. fi
33604.  BLOG CASE_UPDATE_INFO
            /* Returns to another DOCASE level possibly updated information which
            was extracted by CASE_GET_DOCASE_INFO. */
33606.        int NOM /* Index level of DOCASE block. */
33609.        NOM := CURRENT_NEST_LEVEL - 1
33610.        INFORMATION(NOM) := 'CSE' || MISC_FOUND || MULTIPLE_SOF4 ||
33611.        UNEXPECTED_OPERANDS_FOUND || INDEX_RANGE_ASSURED
33612.        /* First byte indicates case label is required on next case. */
33613.        OPERAND1(NOM) := 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. */
35004.
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)]
35016.     and 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.
35024.     call POP_OLD.Block ( ; )
35025.     /* Delete current block, generating END and XII 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 (' $' 11 BLOCK_NUMBER(CURRENT_NEST_LEVEL) 11)
35031.     'END')
35032.     /* Generate branch to end of DOCASE. */
35033.
35034.     end
35035.     /* (Lemma: If CURRENT_NEST_LEVEL > 0 and
35036.     [USER_NAME = '' or BLOCK_NAME(CURRENT_NEST_LEVEL)]
35037.     and BLOCK_TYPE(CURRENT_NEST_LEVEL) = 'CASE' at entry to ESAC, then
35038.     CURRENT_NEST_LEVEL will be decremented by exactly one.) */

C-56
MACRO ESACOD ( ; USER_NAME)

/* Generates final part of DOCASE processing: for SIMPLE, CONDTST,
  or SPARSE type DOCASE, the EQU for the MISC block (or END of DOCASE)
  to the last generated branch target is required; for GENERAL type
  DOCASE, the branch vector and the transfer to any symbolic
  compare or MISC block must be generated. Finally, the block is
  popped. */

INT CASE_COUNTER, /* Holds number of last case generated. */
T, /* Temporary. */
COMP_LABEL_NO, /* Label number of outstanding compare case. */
MAX_SD_VALUE, /* Maximum self-defined operand. */
BASE /* Index within CASE_OCCURS array for CASE 0. */

BIT MISC_FOUND, /* Indicates whether a MISC CASE was found. */
MULTIPLESOF4, /* Indicates whether all branch-vector operands were
  multiples of 4. */
INDEX_RANGE_ASSURED, /* true if we have been assured (by
  "DOCASE *** ,ONLY") that no values other than those specified
  by CASE operands will occur. */
ANY_CMP_CASES, /* Indicates whether any "compare" cases were
  generated (either CHARCMP or symbolic general case operands). */
ANY_SELFDEF_CASES, /* Indicates whether any "self-defining" cases (to
  be handled by branch vector) were generated. */
RANGE_TESTREQD /* Indicates that both branch vector and compare
  operands were present. */

CHAR CASEFORMAT, /* Type of CASEs present: GENERAL, SPARSE,
  CHARCMP, SIMPLE, or CONDTST. */
INDEX_ADDR, /* Address of DOCASE index. */
NOCASE, /* Label for branch vector processing used for unspecified
  cases. */
LABEL /* Any outstanding label waiting to be generated. */

/* (Ground rules: No ESACOD proc modifies CURRENT_NEST_LEVEL.
This can be shown by referring to the cross-reference index.) */
CALL TRACE_PRINTER ( ; 'ESACOD')
/* Print macro name "ESACOD" in mnote if tracing on. */

if CURRENT_NEST_LEVEL ≤ NESTING_LIMIT
  then
    CALL VERIFY_END ( ; 'DOCASE', USER_NAME)
    /* Verifies current block has the name specified by the USER_NAME operand of the ESACOD macro (if any) and that it is a DOCASE block. Various errors receive messages and either intermediate blocks are BLENDed as a fixup or ERROR_OCCURRED is set. */
    [Lemma: If CURRENT_NEST_LEVEL > 0 and
     [USER_NAME = ' ' or = BLOCK_NAME(CURRENT_NEST_LEVEL)] and
     BLOCK_TYPE = 'DOCASE', then
     ERROR_OCCURRED will be set false and CURRENT_NEST_LEVEL will not be modified.] */
    if ERROR_OCCURRED
      then
        mexit
      fi
    call ESACOD_INFO_UNPACK
    /* Extracts CASE_FORMAT, CASE_COUNTER, INDEX_ADDR, MISC_FOUND,
     BLOCK_LABELPREFIX, INDEX_RANGE_ASSURED, and MULTIPLESOF4 from stack. */
    if CASE_FORMAT = 'GENERAL'
      then
        call ESACOD_GENERAL_CASE_CHOICE
        /* Generate all code to complete processing of general case. */
      else
        if CASE_FORMAT = 'SPARSE' or = 'CHARCCMP'
          then
            T := NEXT_COMP_LABEL_NO(GCASE_NEST_LEVEL)
            /* We need to define last compare case target. */
            GCASE_NEST_LEVEL := GCASE_NEST_LEVEL - 1
            /* POP GCASE stack. */
          else /* CONDTST or SIMPLE. */
            T := CASE_COUNTER + 1
            /* We need to define last conditional test target. */
          fi
        if MISC_FOUND
          then
            GENERATE (BLOCK_LABELPREFIX || 'C' || T || ' EQU ' ||
            BLOCK_LABELPREFIX || 'MSC')
            else
              GENERATE (BLOCK_LABELPREFIX || 'C' || T || ' DS OH')
          fi
        fi
    call POP_OLD_BLOCK ( ; )
    /* (Lemma: POP_OLD_BLOCK decrements CURRENT_NEST_LEVEL by exactly one.) */
  end
end
/* (Lemma: If CURRENT_NEST_LEVEL > 0 and
  [USER_NAME = ' ' or = BLOCK_NAME(CURRENT_NEST_LEVEL)] and
  BLOCK_TYPE(CURRENT_NEST_LEVEL) = 'DOCASE' at entry to ESACOD, then
  CURRENT_NEST_LEVEL will be decremented by exactly one.) */
DECO ESACOD_INFO_UNPACK

/* Extract the following information from the stack: */

CASE_COUNTER := OPERAND1(CURRENT_NEST_LEVEL)
INDEX_ADDR := OPERAND2(CURRENT_NEST_LEVEL)
CASE_FORMAT := OPERAND3(CURRENT_NEST_LEVEL)
MISC_FOUND := INFORMATION(CURRENT_NEST_LEVEL)[2,1]
MULTIPLESOF4 := INFORMATION(CURRENT_NEST_LEVEL)[3,1]
INDEX_RANGE_ASSURED := INFORMATION(CURRENT_NEST_LEVEL)[5,1]
BLOCK_LABEL_PREFIX := "$" | BLOCK_NUMBER(CURRENT_NEST_LEVEL)

C-59
CALL ESACOD_GENERAL_CASE_INFO
   /* Pops MAX_SD_VALUE, COMP_LABEL_NO,
      and BASE (of CASE_OCCURS array) out of GCASE stack. */
IF ~ ERROR_OCCURRED
   THEN ANY_SELFDEF_CASES
      THEN CALL ESACOD_SELFDEF_GEN
         /* Handles branch vector-type implementation for all cases which
            contain self-defining terms (of value < 256). Also generates
            linkage for any other terms and/or MISC case which were used
            with the self-definers. */
      ELSE /* No self-definers were present. */
         IF ANY_COMP_CASES
            THEN CALL ESACOD_GENERAL_SYM_ONLY
               /* Generate linkage to process symbolic operands and MISC in the
                absence of self-definers. */
         ELSE
            ERRORS (8, 'SYNC3701 NO CASE CONTAINS NO VALID CASES')
         FI
      FI
   FI
   CORR
DIGG ESACOD_GENERAL_SYM_ONLY

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

37140.  generate (BLOCK_LABEL_PREFIX || 'B' | |
37141.         $ | | COMP_LABEL_NO | |> EQU | |
37142.         $ | | MISC | |
37143.         $ | | COMP_LABEL_NO | | | | DS | | DS)
37144.  end
C-61
ESACOD Macro -- 3 July 1973

37151. BLOCK 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. */
37155. LABEL := BLOCK_LABEL_PREFIX || 'BEG'
37156. /* Note that BEG label must be generated on first instruction. */
37157. docase ifany
37158. of
37159. ANY_COMP_CASES
37160. NOCASE := BLOCK_LABEL_PREFIX || 'C1'
37161. esac
37162. MISC_FOUND
37163. NOCASE := BLOCK_LABEL_PREFIX || 'MSC'
37164. esac
37165. MISC
37166. NOCASE := BLOCK_LABEL_PREFIX || 'END'
37167. esac
37168. esacod
37169. RANGE_TEST_REQD := ((INDEX_RANGE_ASSURED) or ANY_COMP_CASES)
37170. if
37171. then
37172. call ESACOD_OUTOF_RANGE_CHECK
37173. /* Generate check for index out of the range 0 through
37174. MAX_SD_VALUE. */
37175. fi
37176. call ESACOD_BRVCT_GEN
37177. /* Generate branch vector and all final constants and equates
37178. required. */
37182. PROC ESACOD GENERAL_CASE_INFO
37183. /* Pops following information out of GCASE stack. Indicates success
37184. (or lack thereof) in ERROR_OCCURRED. */
37185.
37186. I := GCASE_NEST_LEVEL
37187. if I > GCASE_NEST_LIMIT
37188. then
37189. "ERROR_OCCURRED := true"
37190. else
37191. "MAX_SD_VALUE := MAX_CASE_VALUE(I)"
37192. "ANY_SELFDEF_CASER := (MAX_SD_VALUE > 0)"
37193. "COMP_LABEL_NO := NEXT_COMP_LABEL_NO(I)"
37194. "ANY_COMP_CASER := (COMP_LABEL_NO > 1)"
37195. "BASE := ((I-1) * 256) + 1"
37196. "ERROR_OCCURRED := false"
37197. fi
37198. core C-63

*ESACOD* Macro -- 3 July 1973

C-63
ESACOD Macro -- 3 July 1973

37202. DEFINE 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. */
37206. if INDEX_ADDR = 0
37207. then /* Index is in GPR1. */
37208. generate (LABEL || ' LTX 1,1')
37209. if CASE_OCCURS(BASE)
37210. then */ CASE 0 occurs. */
37211. generate (' $N ' || NOCASE)
37212. else
37213. generate (' $N ' || NOCASE)
37214. fi
37215. generate (' C 1,* || BLOCK_LABEL_PREFIX || 'SIZ')
37216. generate (' $N ' || NOCASE)
37217. if MULTIPLESOF4
37218. then
37219. generate (' LA 0,3')
37220. generate (' $N ' || NOCASE)
37221. generate (' $N ' || NOCASE)
37222. fi
37223. fi
37224. else
37225. generate (LABEL || ' CLI ' || INDEX_ADDR || ',' || MAX_SD_VALUE)
37226. generate (' $N ' || NOCASE)
37227. if MULTIPLESOF4
37228. then
37229. generate (' TM ' || INDEX_ADDR || ',B'0000000111')
37230. generate (' $N ' || NOCASE)
37231. fi
37232. fi
37233. LABEL := ''
37234. C-64
/*ESACOD* Macro -- 3 July 1973

27236. PSEU ESACOD_BRVCT_GEN  
27237.  /* Generate branch vector proper. */  
27238. int I, INCH  
27240.  if INDEX_ADDR # ""  
27241.    then  /* Generate code to put byte index into GPRI. */  
27243.     generate (LABEL || ' SR 1,1')  
27244.     LABEL := ""  
27245.     generate (' IC 1,1 || INDEX_ADDR)  
27246.  fi  
27247.  if MULTIPLESOF4  
27248.    then  
27249.      INCH := 4  
27250.    else  
27251.      INCH := 1  
27252.     generate (LABEL || ' SLA 1,2')  
27253.      LABEL := ""  
27254.  fi  
27255.  if CASE_OCCURS(BASE) OR INDEX_ADDR # ""  
27256.    then /* Zero case must be included in branch vector. */  
27257.      generate (LABEL || ' B **(1)')  
27258.      I := 0  
27259.    else  
27260.      generate (LABEL || ' B *(1)')  
27261.      I := INCH  
27262.  fi  
27263.  while I <= MAX_SD_VALUE  
27264.    do  
27265.      if CASE_OCCURS(BASE+I)  
27266.        then  
27267.          generate (' B ' || BLOCK_LABEL_PREFIX || 'G' || I)  
27268.        else  
27269.          generate (' B ' || NOCASE)  
27270.        fi  
27271.      I := I + INCH  
27272.    od  /* Termination: INCH > 0, so I is incremented in loop;  
27273.      MAX_SD_VALUE is fixed, therefore I must eventually exceed  
27274.      MAX_SD_VALUE. */  
27275.  if RANGE_TEST_OK AND INDEX_ADDR = ""  
27276.    then  
27277.      generate (BLOCK_LABEL_PREFIX || "SIZ DC F" || MAX_SD_VALUE ||  
27278.          "")  
27279.    fi  
27280.  fi  
27281.  if ANY_CMP_CASES  
27282.    then  
27283.      if MISFOUND  
27284.        then  
27285.          generate (BLOCK_LABEL_PREFIX || ' C' || COMP_LABEL_NO || ' EQU ' ||  
27286.            BLOCK_LABEL_PREFIX || 'RSC')  
27287.        else  
27288.          generate (BLOCK_LABEL_PREFIX || ' C' || COMP_LABEL_NO || ' DS DH')  
27289.        fi  
27290.    fi  
27291.  corp
```
"ELOCK" 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 OH')
41025.      /* Define label if one required and not yet defined. */
41026.    fi
41027.  fi
```
"BLOCK" Macro -- 15 June 1973

**PROG BLOCK_TRACE_COUNTERS**

/* Generate debugging information required---block name constant
and/or block counters. */

```plaintext
if DEBUG_BLOCKCOUNTERS_REQD or DEBUG_BLOCKNAMES_REQD
then
  if DEBUG_BLOCKCOUNTERS_REQD
    then
      /* Generate block count incrementing instructions. */
      generate (LABEL II 'LB 1,' II BLOCK_LABEL_PREFIX II 'BLC')
      LABEL := ' ' /* Clear LABEL to show it has been generated. */
      generate ("LA 1,1(((')")
      generate ('STB 1,' II BLOCK_LABEL_PREFIX II 'BLC')
      fi
    fi
  generate (LABEL II 'II BLOCK_LABEL_PREFIX II 'GO')
  LABEL := BLOCK_LABEL_PREFIX II 'GO'
  /* Establish GO label as requiring definition. */
  if DEBUG_BLOCKNAMES_REQD
    then
      generate ("DC C'" II BLOCK_NAME(CURRENT_NEST_LEVEL) II "C'"
      generate (BLOCK_LABEL_PREFIX II 'BC DC H'0' BLOCK COUNT")
    fi
  fi
fi
```
**BLEND** Macro -- 15 June 1973

43001. **MACRO BLEND** ( ; USER_NAME, RETURN=, LINK=14, RESTORE=, RC=)
43002. /* The BLEND macro acts as a generic name for IF, OD, ESAC, ESACOD, and
43003. PROC and as a terminating macro for BLOCK. For any
43004. block type being terminated. For BLOCK blocks, the block is
43005. simply terminated */
43006. **AI I */ Temporary. */
43007. **CALL TRACE.PRINTER** ( ; 'BLEND')
43008. /* Prints macro name "BLEND" in mnote if tracing on. */
43009. **IF CURRENT_NEST_LEVEL > NESTING_LIMIT**
4310. **THEN**
4312. **CALL POP_OLD_BLOCK** ( ; )
4313. **ELSE**
4314. **IF CURRENT_NEST_LEVEL = 0**
4316. **THEN**
4317. **MNOTE** (6, 'STRC4317 NO BLOCKS ACTIVE--"BLEND" IGNORED')
4318. **ELSE**
4319. **I := CURRENT_NEST_LEVEL**
4320. **IF USER_NAME = ""**
4321. **THEN**
4322. **WILE I > 0 and BLOCK_NAME(I) # USER_NAME**
4323. **DO */ (Termination: I is decremented—must eventually become
4324. < 0.) */
4325. **IF I = 0**
4326. **THEN**
4327. **MNOTE** (6, 'STRC4327 NO BLOCK ACTIVE NAMED ' USER_NAME II
4328. '—"BLEND" IGNORED')
4329. **EXIT**
4330. **FI**
4331. **DOCASE BLOCK_TYPE(I) only**
4332. **OF**
4333. **CASE 'IF'**
4334. **CALL IF** ( ; USER_NAME)
4335. **ESAC**
4336. **CASE 'DO'**
4337. **CALL OD** ( ; USER_NAME)
4338. **ESAC**
4339. **CASE 'CASE'**
4340. **CALL ESAC** ( ; USER_NAME)
4341. **ESAC**
4342. **CASE 'DCase'**
4343. **CALL ESACOD** ( ; USER_NAME)
4344. **ESAC**
4345. **CASE 'PROC'**
4346. **CALL CORP** ( ; USER_NAME, RETURN=RETURN, LINK=LINK, RESTORE=RESTORE,
4347. RC=RC)
4348. **ESAC**
4349. **CASE 'BLOCK'**
4350. **CALL POP_OLD_BLOCK** ( ; )
4351. **ESACED**
4352. **ESAC**
4353. **FI**
4354. **END**
4355. /* (Lemma: If CURRENT_NEST_LEVEL > 0 and
4356. [USER_NAME = "" or \= BLOCK_NAME(CURRENT_NEST_LEVEL)] at entry to
4357. BLEND, then CURRENT_NEST_LEVEL will be decremented by exactly
4358. one.) */

C-68
"FINAL" Macro -- 10 July 1973

53001. \texttt{macro} FINAL ( ; )
53002. \texttt{/* Insure all blocks are closed. Then if SAVETRACE_ON_FIRST_PROC,}
53003. \texttt{define label $LASTSAV to be PREV_SAVETRACE_AREA and equate}
53004. \texttt{PREV_SAVETRACE_PTR to 0. */}
53006. \texttt{call TRACE_PRINTER ( ; 'FINAL')}
53007. \texttt{/* Print macro name "FINAL" in note if tracing on. */}
53009. \texttt{while CURRENT_NEST_LEVEL > 0}
53101. \texttt{if CURRENT_NEST_LEVEL > NESTING_LIMIT}
53102. \texttt{mnote (6, 'STRC53C1 BLEND OF OUTSTANDING BLOCK ASSUMED')}
53103. \texttt{else}
53104. \texttt{mnote (6, 'STRC5301 BLEND OF ' || BLOCK_NAME(CURRENT_NEST_LEVEL) ||
53105. \texttt{" ASSUMED")}
53106. \texttt{fi}
53107. \texttt{call BLEND ( ; )}
53108. \texttt{/* (Lemma: If CURRENT_NEST_LEVEL > 0 and no BLEND operands are}
53109. \texttt{specified, CURRENT_NEST_LEVEL will be decremented by exactly}
53110. \texttt{one.) */}
53121. \texttt{od} \texttt{/* (Termination: CURRENT_NEST_LEVEL decreases monotonically}
53122. \texttt{and therefore must eventually become 0.) */}
53123. \texttt{if SAVETRACE_ON_FIRST_PROC}
53124. \texttt{then}
53125. \texttt{if PREV_SAVETRACE_PTR == '$FIRSTSV'}
53126. \texttt{then \texttt{/* No non-OS proc occurred; generate dummy area. */}
53129. \texttt{else}
53130. \texttt{generate ('$LASTSAV EQU $FIRSTSV')}
53131. \texttt{generate ('$LASTSAV EQU ' || PREV_SAVETRACE_AREA)}
53132. \texttt{fi}
53133. \texttt{fi}
53134. \texttt{mend}

\texttt{C-69}
"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. error
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 needed as IT label; if no such block,
55013. issue message and set ERROR_OCCURRED. */
55014. if not ERROR_OCCURRED
55015. then
55016. generate (USER_NAME || "B" || ULTIMATE_BRANCH_LABEL)
55017. fi
55018. end
"PROC" Macro -- 5 July 1973

81001. MACRO PROC (USER_NAME: LINKAGE=, ID=, BASE=, WCRK=, SAVE=, DEBUG=, XIT=)
81002. /* Defines a procedure block. If LINKAGE=OS is specified, standard
81003. OS save area conventions are followed; otherwise a simple non-
81004. linked save area is provided. A base register is established
81005. (unless BASE=None is specified under OS_LINKAGE or BASE= is
81006. omitted on local PROCs). Register values upon entry are
81007. saved to allow restoring at CORP time. */
81008.
81009. bit
81010. FIRST_PROC, /* Indicates whether this is the first PROC macro coded
81011. in this assembly. */
81012. FIRST_VALUEKNOWN, /* 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,--)
81015. was entered. */
81016. SPECIAL_PREFIX, /* Indicates whether the BLOCK_LABEL_PREFIX was
81017. changed to the special debugging form "$Pp$". */
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.
81024. char
81025. COMMA2, /* Contain "," and "H" respectively if a range of
81026. registers is to be saved, or the null string if a single register
81027. to be saved. Used to generate "STM" or "STM" instructions. */
81028. FIRST, LAST, /* First and last register in range to be saved. */
81029. LABEL, /* Any outstanding label waiting to be generated. */
81030. LOCAL_POINTER, OS_POINTER, /* Instruction segments to generate
81031. store instruction for proper save area. */
81032. PREVIOUS_DEBUG_VECTOR, /* Holds value of debug switches on entry to
81033. PROC macro for restoring on exit from CORP. */
81034. PROC_ID_BYTE, /* Value of hex proc number (PROC_COUNTER in hex)
81035. used in various debugging instructions. */
81036. SAVE_LENGTH, /* Length of save area (in words), except length of
81037. register part only for local PROCs. */
81038. SAVETYPE, /* Type of save area generated: FULL (savetrace), OSSAVE,
81039. NORMAL, NORMALHDR, TRUNC, TRUNCHDR, or NONE. */
81040. SAVEBEG, /* Register (work or base) which in pointing at new
81041. save area before chaining. */
81042. WORKREG /* Register used for setting up linkage, etc. */
81043.
81044. int
81045. OFFSET, /* Offset (in words) to either FIRST (if FIRST_VALUE_KNOWN),
81046. or to GPRO within save area. */
81047. OFFSET_TO_GPRO, /* Offset in words to GPRO within save area. */
81048. SAF, SAL /* Register number to go into first register word of
81049. save area; this, for example, could be 14 even though FIRST
81050. is a symbolic register of unknown value at macro expansion time.
81051. SAL is similar but for last register. */

C-71
"PROC" Macro -- 5 July 1973

81051. call TRACE_PRINTER ( ; "PROC")
81052. /* Print macro name "PROC" in mnote if tracing on. */
81053. call PUSH_NEW_BLOCK(USER_NAME; BLOCK_TYPE_VALUE=PROC)
81054. /* Define new block: add to stack. Initialize block specifications.
81055. Note block type and set up unique BLOCK_LABEL_PREFIX for use
81056. in generating unique labels. */
81057. if ERROR_OCCURRED /* during PUSH_NEW_BLOCK (viz., stack overflow) */
81058. then
81059. mexit
81060. fi
81061. LABEL := USER_NAME
81062. /* Generate PROC's name at first opportunity. */
81063. call PROC_SCAN_OPTIONS
81064. /* Validate LINKAGE= and WORK= keywords; issue error messages and set
81065. OS_LINKAGE and WORKREG. Process completely DEBUG and EXIT keywords.
81066. Change BLOCK_LABEL_PREFIX if necessary to special PROC form
81067. (indicating change in SPECIAL_PREFIX) and set value of FIRST_PROC. */
81068. call PROC_HEADER
81069. /* Generate "CSECT" and "USING *,15" if required. Handle in-line
81070. ID (a la IBM SAVE macro). */
81071. call PROC_REG_SAVE
81072. /* Set SAVE_TYPE and SAVE_LENGTH to indicate type of save area
81073. required. Save contents of general purpose registers, if required. */
81074. call PROC_ESTABLISH_BASE
81075. /* Set up base register where required and issue USING. Set USING13 if
81076. base register to be loaded into 13 was put temporarily into
81077. WORKREG. If multiple base registers, set MULTIBASE. */
81078. call PROC_GEN_SAVEAREA
81079. /* Generate proper save area depending on the variables SAVE_TYPE and
81080. SAVE_LENGTH set by PROC_REG_SAVE and depending on the save macro
81081. operands. */
81082. call PROC_MULTIBASE_GEN
81083. /* Generate definition of adverbs for multiple base registers. */
81084. call PROC_DEBUG_STUFF
81085. /* Generate trace and count code for debugging, if requested. */
81086. if WORK = 'NONE' and WORKREG_USED
81087. then
81088. generate (LABEL || 'LR ' || WORKREG || ',0')
81089. /* Restore WORKREG. */
81090. LABEL := ''
81091. fi
81092. if LABEL = ''
81093. then
81094. generate (LABEL || 'DS 0H')
81095. fi
81096. if LINKAGE(2) = 'CSECT'
81097. then
81098. generate (" DROP 15")
81099. /* DROP for USING generated by PROC_HEADER. */
81100. fi
81101. call PROC_INFO_SAVE
81102. /* Save any information necessary to generate CORP macro. */
PROC SCAN OPTIONS

/* Validate LINKAGE= and WORK= keywords; issue error messages and set
OS LINKAGE and WORKREG (the latter receiving either the register
specified by the WORK= operand or some default). Process completely
the DEBUG= and EXIT= keywords. Change BLOCK_LABEL_PREFIX if
necessary from the normal "$nnn" to the special PROC prefix
"$P". Set FIRST PROC to indicate whether this is the first
PROC macro coded in this assembly. */

 OS_LINKAGE := (LINKAGE(1) # 'OS' and # '')

if LINKAGE(1) # 'OS' and # ''
then
\Note (8, 'STC6801 LINKAGE= ' || LINKAGE(1) ||
' INVALID--"OS" ASSUMED')

fi

if LINKAGE(2) # 'CSECT' and # ''
then
\Note (8, 'STC6802 SECOND LINKAGE OPERAND IGNORED')

fi

WORKREG := WORK,

If WORKREG = 'NONE' or = '
then
/* Pick default WORKREG. We will restore it later if WORK=NONE and
it gets clobbered. */

if OS_LINKAGE
then
WORKREG := '2'

else
WORKREG := '1'

fi

fi

EXIT_SEVERITY := EXIT

FIRST_PROC := - NOT FIRST_PROC /* Brilliant, eh? */

NOT FIRST_PROC := true

/* Note whether this is the first proc and set global NOT_FIRST_PROC
so we will be able to tell on later PROCs. Note that we are making
use here of the fact that global bit variables are initialized to
false. */

call PROC_DEBUG_SET

/* Set debugging switches according to DEBUG= operand, saving prior
values for restoring at CORP time in PREVIOUS_DEBUG_VECTOR. */

SPECIAL_PREFIX := (DEBUG SAVETRACE REQD or DEBUG PROCTRACE REQD or
DEBUGPROCCOUNTS_REQD) and

PROCCOUNTER < 254

if SPECIAL_PREFIX
then /* We want to label this proc with a hex proc number. */

PROC_COUNTER := PROCCOUNTER + 1

/* Advance counter only on those procs which use it. */

HEX_IN := PROC_COUNTER

call HEX ( )

PROC_ID_BYTE := HEX_OUT

BLOCK_LABEL_PREFIX := "$P" || PROC_ID_BYTE

/* Change labels on PROC to ease understanding of debug information. */

BLOCK (*, 'STC6804 PROC_ID=' || BLOCK_NAME(CURRENT_NEST_LEVEL) ||
'" " DEBUG ID=X" || PROC_ID_BYTE || "")

else
PROC_ID_BYTE := '00'

fi

corp
81167. **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 is to be set either on or
81178. off. */
81179. int I /* List sub-operand index. */
81180. GLOBAL := false /* Has not yet been found. */
81181. PREVIOUS **DEBUG** VECTOR :=
81182. DEBUG BLOCKNAMES_REQD //
81183. DEBUG PROCNAMES_REQD //
81184. DEBUG LISTBLOCKS_REQD //
81185. DEBUG BLOCKCOUNTS_REQD //
81186. DEBUG PROCCOUNTS_REQD //
81187. DEBUG CORPVALUES_REQD //
81188. DEBUG SAVETRACE_REQD //
81189. /* Save current value of debug switches. */
81190. I := 1
81191. while I < *DEBUG
81192. do
81193. case **DEBUG**(I)
81194. of
81195. case "GLOBAL", "GBL"
81196. GLOBAL := true
81197. esac
81198. case "BLOCKNAMES", "BN"
81199. DEBUG BLOCKNAMES_REQD := true
81200. /* BLOCKNAMES causes the name of each block to be generated as an
81201. in-line character constant at the start of each block (of any type,
81202. not just BLOCK macros) for ease of locating code in dumps. */
81203. esac
81204. case "PROCNAMES", "PN"
81205. DEBUG PROCNAMES_REQD := true
81206. /* PROCNAMES causes the name of each PROC to be generated as an
81207. in-line character constant at the start of the PROC for ease of
81208. locating code in dumps. */
81209. esac
81210. case "LISTBLOCKS", "LB"
81211. DEBUG LISTBLOCKS_REQD := true
81212. /* LISTBLOCKS causes the name, number, and depth of each block to be
81213. generated in an mnote at the start and end of the block. */
81214. esac
81215. case "BLOCKCOUNTS", "BC"
81216. DEBUG BLOCKCOUNTS_REQD := false
81217. /* BLOCKCOUNTS causes counters to be kept on the number of executions
81218. of all blocks. */
81219. esac
81220. case "PROCVALUES", "PC"
81221. DEBUG PROCVALUES_REQD := false
81222. /* PROCVALUES causes counters to be kept on the number of executions
81223. of all PROC blocks. */
81224. esac
81225. case "WALLISTBLOCKS", "WLB"
81226. DEBUG LISTBLOCKS_REQD := false
81227. esac
81228. case "BLOCBLOCKS", "BCC"
81229. DEBUG BLOCKBLOCKS_REQD := false
81230. /* BLOCKBLOCKS causes counters to be kept on the number of executions
81231. of all blocks. */
81232. esac
81233. case "WALLISTPROC", "WPC"
81234. DEBUG LISTPROC_REQD := false
81235. /* LISTPROC causes counters to be kept on the number of executions
81236. of all PROC blocks. */
81237. esac
81238. esac
81239. esac
81240. esac
81241. DEBUG PROCVALUES_REQD := false
81242. esac
81243. C-74
/* PROC Macro -- 5 July 1973 */

```assembly
CASE ('PROCTRACE', 'PT')
  DEBUG_PROCTRACE_REQD := true
  /* PROCTRACE causes a trace vector to be generated and instructions to move the hex PROC number into the vector to show the order of PROC calls. */
CASE ('NOPROCTRACE', 'NPT')
  DEBUG_PROCTRACE_REQD := false
CASE ('CORPVALUES', 'CV')
  DEBUG_CORPVALUES_REQD := true
  /* CORPVALUES causes the value of the registers at CORP time (before restoring those saved at PROC entry) to be stored into an area for reference. */
CASE ('NOCORPVALUES', 'NCV')
  DEBUG_CORPVALUES_REQD := false
CASE ('SAVETRACE', 'ST')
  SAVETRACE_VALUE := true
  SAVETRACE_CHECK := true
  /* Note that savetrace has been specified. */
CASE ('NOSAVETRACE', 'NST')
  SAVETRACE_VALUE := false
  SAVETRACE_CHECK := true
CASE ('ALL', 'NONE')
  DEBUG_BLOCKNAMES_REQD, DEBUG_PROCNAMES_REQD, DEBUG_LISTBLOCKS_REQD, DEBUG_BLOCKCOUNTS_REQD, DEBUG_PROCCOUNTS_REQD, DEBUG_PROCTRACE_REQD, DEBUG_CORPVALUES_REQD, SAVETRACE_VALUE := (DEBUG(I) = 'ALL')
  /* Set (or reset) all main debug switches. */
CASE ('MACRONAMES', 'MN')
  DEBUG_MACRONAMES_REQD := true
  /* MACRONAMES causes the name of each structured macro (including inner macros) to be printed in an mnote whenever invoked. */
CASE ('NOMACRONAMES', 'MN')
  DEBUG_MACRONAMES_REQD := false
CASE ('DEBUGMACROS', 'DM')
  DEBUG_DEBUGMACROS_REQD := true
  /* DEBUGMACROS causes various intermediate values within the macros to be printed in notes for use in debugging the macros. */
CASE ('NODEBUGMACROS', 'NDM')
  DEBUG_DEBUGMACROS_REQD := false
CASE ('STRCDBG', 'DBG')
  STRCDBG := (I | DEBUG(I)) | 'INVALID-IGNORED'
```
"PROC" MACRO -- 5 July 1973

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.                                       else (4, 'STEC8103 WARNING--SAVETRACE REQUIRES "FINAL" MACRO')
81321.                                        else
81322.                                                            else
81323.                                                                 if SAVETRACE CN FIRST PROC
81324.                                                                 then
81325.                                                                         /* SAVETRACE is being resumed. */
81326.                                                                             then
81327.                                                                                            DEBUG_SAVETRACE_REQD := true
81328.                                                                                            else
81329.                                                                                                    else
81330.                                                                                                    else
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

*--------------------------------------------------------------------------*

C-76
"PROC" Macro  --  5 July 1973

8134.  "PROC" HAMABER
8135.  / * Generates a "CSECT" and "USING *,15" if LINKAGE=(**,**,CSECT)
8136.  specified. If LINKAGE=(OS,**,** specified, generate inline ID
8137.  similar to ID=SAVE macro; for non-OS linkage, do the same if
8138.  OS_LINKAGE=**,** specified, but omit ID-length-count field.
8139.  */
8140.  if OS_LINKAGE = **,** then
8141.  \* Return any label generated as branch target in LABEL. */
8142.  if LINKAGE=(**,**,** specified. If LIdKAGE=(OS,---) specified, generate inline ID
8143.  similar to £13
8144.  SAVE mazro; for non-OS linkage, do the same if
8145.  3B3'3
8146.  _?OCNAAS RE2D or ID= specified, but omit ID-length-count field.
8147.  Return
8148.  ina
8149.  Label generated as branch target in LABEL.
8150.  */
8151.  char SECT,
8152.  /*. name (or "$PRIVATE") for default TID constant.
8153.  */
8154.  TARGET
8155.  /*. Temporary. */
8156.  if ((OS_LINKAGE or ID= '')) and ID = 'NONE' or
8157.  BASE_PROGRAMSFILE
8158.  then
8159.  TARGET := BLOCK_LABEL_BLOCK of 'AA'
8160.  generate (LABEL (LABEL (LABEL (LABEL
8161.  if ID= '' we specified on PROC macro. */
8162.  else
8163.  SEC := USER_NAME
8164.  LENST4 := K'USER NAME
8165.  fi
8166.  if OS_LINKAGE
8167.  then
8168.  SEC := $PRIVATE
8169.  else
8170.  SEC := BLOCK_NAME(CURRENT_NEST_LEVEL)
8171.  fi
8172.  LEN3T4 := 8
8173.  else
8174.  SEC := USER
8175.  LENST4 := K'USER
8176.  fi
8177.  if OS_LINKAGE
8178.  then
8179.  LENGTH := ((K'ID/2)*2) + 1 /* Round up to odd number. */
8180.  generate (" DC AL(' | LENGTH | *)',CL' | LENGTH |
8181.  ": | SEC | *)"
8182.  else
8183.  generate PROC-name constant only. */
8184.  LENGTH := ((K'ID + 1)/2)*2 /* Round up to even number. */
8185.  generate (" DC CL' | LENGTH | " | SEC | "")
8186.  fi
8187.  else
8188.  if ID= specified. */
8189.  QUOTE := 1
8190.  LENGTH := 0
8191.  if ID= ''
8192.  then
8193.  QUOTE := 1
8194.  LENGTH := -2 /* Subtract 2 for the quotes. */
8195.  fi
8196.  if OS_LINKAGE
8197.  then
8198.  LENGTH := ((K'ID/2)*2) + 1 /* Round up to odd number. */
8199.  generate (" DC AL(' | LENGTH | *)',CL' | LENGTH |
8200.  " | ID | "QUOTE)
8201.  else
8202.  LENGTH := ((K'ID + 1)/2)*2 /* Round up to even number. */
8203.  generate (" DC CL' | LENGTH | " | ID | "QUOTE)
8204.  fi
8205.  fi
8206.  fi
PROC Macro -- 5 July 1973

If SAVE=NOE and doing SAVETRACE generate store of all registers.
If SAVE=NOE and no SAVETRACE, do nothing. For other than
SAVE=NOE, extract from SAVE=NOE's register list to be saved.
Give type and size of save area and put into SAVE(TYPE) and
SAVE.LENGTH. Generate instruction to store register.*/

FIRST, LAST := '
OFFSET, SAF, SAL, OFFSET_TO_GPRO := 0
if SAVE = 'NOE'
then
  if DEBUG_SAVETRACE := 0
  then
    SAVE_TYPES := 'FULL'
    SAVE.LENGTH := 15
  end
  FIRST := 'NOE
  LAST := 'NOE
  SAF := 0
  SAL := 0
  OFFSET_TO_GPRO := 0
  LOCAL_POINTER := 'NOE
  OS_POINTER := 'NOE
  LABEL := '
else
  call PROC_SAVETRACE_SAVE_INFO
  call PROC_SET_SAVE_TYPE
  call PROC_SECT_SAVE_INFO
  offset in bytes.
  */
  if SAVE_TYPE = 'FULL'
  then
    SAVETYPE (LABEL || 'STN 14, 12, ' || BLOCK_LABEL_PREFIX || 'SV+12')
    LABEL := '
else
  SAVETYPE (LABEL || 'STN ' || FIRST || LAST || COMMA2 || LOCAL_POINTER || OS_POINTER)
  else
    SAVETYPE (LABEL || 'STN ' || FIRST || LAST || COMMA2 || LOCAL_POINTER || OS_POINTER)
  fi
  LABEL := '
end

*-----*/

"PROC" Macro -- 5 July 1973

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

FIRST_VALUE_KNOWN := false
MULT := 'I'
COMMA2 := 
/* Assumed values. */
if T'SAVE(1) = 'O' /* At least first suboperand is omitted. */
then FIRST := '14' SAF := 14
LAST := '12' SAL := 12
/* Default is to save all registers 14 through 12. */
else if T'SAVE(1) = 'N' /* Self-defining term. */
then SAF := SAVE(1) /* Store it as a number. */
FIRST := SAF /* Convert it back to a string (done for non-decimal
self-defining terms). */
else /* It must be symbolic. */
FIRST := SAVE(1) /* Store it as a character string. */
SAF := 14 /* Just say first of save area is register 14. */
FIRST_VALUE_KNOWN := false
fi

if T'SAVE(2) = 'O' /* Second suboperand is omitted. */
then LAST, MULT, COMMA2 := 
if FIRST_VALUE_KNOWN
then SAL := SAF /* Last register is same as first. */
else SAL := 12 /* Last register is 12. */
fi
else
if T'SAVE(2) = 'N' /* Self-defined. */
then SAL := SAVE(2) /* Store it as a number. */
LAST := SAL /* Convert it back to a string. */
else
LAST := SAVE(2) /* Store it as a character string. */
SAL := 14 /* Just say last of save area is register 12. */
fi
fi
```
```
"PROC" Macro -- 5 July 1973

81530. DEOC PROC_DECIDE_SAVE_TYPE
81531. /* Set SAVE_TYPE with type of save area to be generated: NONE, OSSAVE,
81532. TRUNC, TRUNCDIR, NORM, NORMHDR, or FULL. Set SAVE_LENGTH with
81533. length (in words) of save area (except only length of register part
81534. for non-OS LINKAGE areas). Offset in save area (in words) of either
81535. FIRST (if FIRST_VALUE_KNOWN) or else of register 0 is put into OFFSET;
81536. the latter always is stored into OFFSET_TO_GPRO for COMP's reference.
81537. Also set LOCAL_POINTER to '1' and OS_POINTER to '13' if OS_LINKAGE,
81538. else LOCAL_POINTER to (BLOCK_LABEL_PREFIX || 'SV') and OS_POINTER to
81539. '1'. */
81540. int I /* Temporary. */
81542. if OS_LINKAGE
81543. begin
81544. OS_POINTER := '13'
81545. LOCAL_POINTER := '1'
81546. if SAVE(3) = 'NONE'
81547. then
81548. SAVE_TYPE := 'NONE'
81549. else
81550. SAVE_TYPE := 'OSSAVE' /* Standard OS save area. */
81551. OFFSET_TO_GPRO, OFFSET := 5 /* Put offset to reg 0 in both. */
81552. if SAVE(4) = ''
81553. then
81554. SAVE_LENGTH := '18' /* Standard OS save area is 18 words. */
81555. else
81556. SAVE_LENGTH := SAVE(4) /* Length specified. */
81557. fi
81558. fi
81559. else /* Not OS_LINKAGE. */
81560. OS_POINTER := -'
81561. LOCAL_POINTER := BLOCK_LABEL_PREFIX || 'SV'
81562. if DEBUG_SAVETRACE_REQD
81563. begin
81564. SAVE_TYPE := 'FULL' /* Full 18 word pseudo-OS save area. */
81565. SAVE_LENGTH := '15'
81566. /* Length of register part of full save area is 15 words. */
81567. OFFSET_TO_GPRO, OFFSET := 5
81568. end
81569. else
81570. SAVE := SAL - ((SAL+2)/16*16) - SAF + ((SAF+2)/16*16) + 1
81571. /* Convert calculated length to character string. */
81572. if SAF = 14
81573. then /* SAVE(1) was not symbolic, specified as 14, or symbolic. */
81574. else /* DEBUG_SAVETRACE_REQD */
81575. begin /* Header included for count. */
81576. SAVE_TYPE := 'NORMHDR' /* First register word is 14. */
81577. OFFSET_TO_GPRO, OFFSET := 3
81578. end
81579. else /* If DEBUG_SAVETRACE_REQD */
81580. begin /* Header included for count. */
81581. SAVE_TYPE := 'TRUNCHDR'
81582. OFFSET := 1 /* To SAVE(1). */
81583. end
81584. end /* DEBUG_SAVETRACE_REQD */
81585. if SAF > 13
81586. begin /* OFFSET_TO_GPRO := OFFSET - SAF */
81587. OFFSET_TO_GPRO := OFFSET_TO_GPRO + 16
81588. end
81589. else /* If FIRST_VALUE_KNOWN */
81590. begin /* SAVE(1) was not symbolic */
81591. SAVE_TYPE := 'TRUNC'
81592. OFFSET := 0 /* To SAVE(1). */
81593. end /* DEBUG_SAVETRACE_REQD */
81594. end /* DEBUG_SAVETRACE_REQD */
81595. fi
81596. fi
81597. else /* Not OS_LINKAGE. */
81598. OS_POINTER := -'
81599. LOCAL_POINTER := '1'
81600. if SAVE(3) = 'NONE'
81601. then
81602. /* Adjust OFFSET from giving offset to GPRO to give offset to SAF. */
81603. OFFSET := OFFSET_TO_GPRO - SAF
81604. if SAF > 13
81605. begin /* OFFSET_TO_GPRO := OFFSET - SAF */
81606. OFFSET_TO_GPRO := OFFSET_TO_GPRO + 16
81607. end
81608. end
81609. else
```
PROC PROC_ESTABLISH_BASE

/* Set up base register and issue USING where required. */

bit INLINSAVEA /* Indicates whether an inline save area is to be generated. */

char BASEREG /* Name of register loaded with base value. */

int I, J /* Temporaries. */
PROC Macro -- 5 July 1971

81621. if BASE # "NONE" and (OS_LINKAGE or BASE # "")
81622. then /* Generate a base register. */
81623. BASEREG := BASE(1)
81624. INLINESAVERARA := (SAVE # "NONE" and SAVE(3) = 1)
81625. if BASEREG = 13 and (INLINE_SAVERAREA and OS_LINKAGE)
81626. then
81627. /* BASE REG 13 */
81628. BASEREG := 12
81629. fi
81630. if BASEREG = 13 then /* BASE register specified. */
81631. if INLINE_SAVERAREA and OS_LINKAGE
81632. then
81633. BASEREG := WORKREG
81634. /* We will load the base value first into the work register, then
81635. copy the value to register 13 after we finish all linkage. */
81636. JSIG13 := true
81637. fi
81638. if OSLINKAGE then
81639. advance ("CNOP 0,4")
81640. /* Advance to fullword boundary; outstanding label can wait
81641. for next instruction. */
81642. fi
81643. TARGET := EQU_TOKEN(1)
81644. generate (LABEL II "BAL II BASE(I) II " II TARGET)
81645. LABEL := TARGET
81646. else
81647. fi
81648. if N'BASE > 1
81649. then
81650. if OS_LINKAGE
81651. then
81652. "LABEL (" CRAP 0,4")
81653. /* Advance to fullword boundary; outstanding label can wait next
81654. instruction. */
81655. fi
81656. TARGET := EQU_TOKEN(1)
81657. generate (LABEL II "BAL II BASEREG II " II TARGET)
81658. LABEL := TARGET
81659. else /* No inline save area. */
81660. generate (LABEL II "BALR II BASEREG II " II)
81661. LABEL := ""
81662. fi
81663. fi
81664. if INLINESAVERAREA
81665. then
81666. if OS_LINKAGE
81667. then
81668. generate ("USING *, 11 BASEREG"
81669. else
81670. if N'BASE > 1
81671. then
81672. generate ("USING *, 11 BASEREG"
81673. fi
81674. "MULTIBASE := " TARGE
81675. J := 4096
81676. I := 2
81677. while I < N'BASE
81678. do
81679. generate ("USING *, 11 J II " II BASE(I))
81680. I := I + 1
81681. J := J + 4096
81682. od /* (termination: Same proof as above.) */
81683. fi
81684. fi
81685. fi
81686. "JUMP 1"
81687. "BASE 1"
81688. "BASE 1"
81689. "BASE 1"
81690. "BASE 1"
81691. "BASE 1"
81692. "BASE 1"
81693. "BASE 1"
81694. "BASE 1"
81695. "BASE 1"
81696. "BASE 1"
81697. /*
SRSC PROC_SAVAREA

/\ Generate appropriate save area according to SAVE_TYPE, SAVE_LENGTH,

and the SAVE suboperands. */

if SAVE_TYPE = 'OSSAVE'

then

call PROC_SAV_OSSAVE_AREA

/* Generate OS save area and chain it up following OS linkage

conventions. Also link up static chain of local save areas if

this is the first proc and SAVETRACE requested. */

else

if (SAVE # 'NONE' and SAVE_TYPE # 'NONE') or

EBJG_SAVETRAC_REQD

then

call PROC_SAV_LOCAL_SAVEAREA

/* Generate local PROC save area according to SAVE_TYPE and SAVE_LENGTH

and, if SAVETRACE requested, provide static save area chaining. */

fi

fi

else

C-83
"PROC" macro -- 5 July 1973

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. SAVEREG requested. */
81724.
81725. call PROC_DEFTEST_NEW_OSSAVE
81726. /* generate linkage, out-of-line, or dynamic save area and point to
81727. it with BASEREG or WORKREG; put register name in SAVEREG. */
81728. if DEBUG_SAVERSAREA_REQD and FIRSTPROC
81729. then /* Static chain of local save areas must be linked to OS save areas. */
81730. generate (LABEL II 'ST 13,$FIRSTSV+4')
81731. LABEL := 'ST 13,$FIRSTSV+4'
81732. generate ('MVC 8(4,13),=A($FIRSTSV)')
81733. generate ('L 13,$LASTSAV')
81734. PREV_SAVETRACE_PTR := '$FIRSTSV'
81735. PREV_SAVETRACE_AREA := '0'
81736. fi
81737.
81738. if DEBUG_PROCTRACE_REQD and FIRSTPROC and SAVE(3)
81739. then /* Static pointer to PROC trace vector in word 1 of OS save area. */
81740. if WORK = 'NONE' and ~ WORKREG_USED
81741. then
81742. generate ('LR 0,WORKREG')
81743. if
81744. generate ('LA 1 WORKREG',STRACE)
81745. WORKREG_PTR :='start'
81756. fi
81757. else
81758. else
PROC_DEFINE
OSSAVE

/* Generate inline, out-of-line, or dynamic save area for
OS_LINsbAg and point to it with the WORKREG or BASEREG. Put base of
pointing register in SAVERSREG. */

char X /* Temporary */

X := "SAVEREG := 433REG /* Assumed */

if SAVE(3) = 'JUNAM'
then
generate (LABEL II ' LA 0,(' II SAVE_LENGTH II ')#1')
LABEL := '
if WORK = 'NONE'
then
generate (' GETMAIN i,LV=(0)')
generate (' LR 0, II WORKREG)
generate (' LR II WORKREG (',1')
if WORK = 'IUN'
then
generate (' LR 1,24(13)')
if FIRST # '!2' or LAST = '*2'
then
SAVEREG (4, 'STARC8107 REG 1 MUST BE AMONG THOSE SAVED')
fi
else
generate (' LR 1,0')
fi
WORKREG_JSG := true
go else
if SAVE(3) = '
then /* Ailine save area. */
if Label = '
then
if WORK = 'NONE' and WORKREG_USED
then
generate (' LR 0, II WORKREG)
fi
WORKREG_USED := true
generate (' CHOP 0,' '*')
LABEL := BLOCK_LABEL_PREFIX II 'CC'
BAL := WORKREG (',', II LABEL)
else
X := BLOCK_LABEL_PREFIX II 'ISV'
fi
then
SAVEREG := BASEREG
fi
if DBJ;_ROCTRACE_REQD and FIRST_PROC
then
generate (X II ' DC A(TRACE),(' II SAVE_LENGTH II ')#1') /* Generate inline save area with first word pointing to trace vector. */
else
generate (X II ' DC (' II SAVE_LENGTH II ')#1') /* Generate inline save area with first word pointing to trace vector. */
call PROC_MULTIBASE_GEN
/* Insert multiple base addons after save area. */
generate (LABEL II ' LA 0, II WORKREG)
LABEL := '
fi
WORKREG_USED := 3EAD
generate (LABEL II ' LA 0, II WORKREG)
fi
LABEL := '
fi
/* Area has been generated and address is in SAVERSREG register. */
PROC Macro -- 5 July 1973

81833. PROC JIN_LOCAL_SAVEAREA
81834.     /* Generate local PROC save area according to SAVE_TYPE and SAVE_LENGTH
81835.     and, if SAVE_TRACE requested, provide static save area chaining. */
81836.     char FWD_PTR */ Area used for next save area. */
81838.     if LABEL = "
81839.     then
81840.     LABEL := BLOCK_LABEL_PREFIX || 'DD'
81841.     generate ('B' || LABEL)
81842.     endif
81844.     if SAVE_TYPE = 'FULL' or SAVE_TYPE[6,3] = 'HDR'
81845.     then /* Word one should contain PROC count and ID byte. */
81846.     generate ('DC X'PF' || PROC_ID_BYTE '0000'
81847.     'FLAG (RETRIED, FINISHED), ID, COUNT')
81849.     endif
81851.     if SAVE_TYPE = 'FULL'
81852.     then
81853.     FWD_PTR := BLOCK_LABEL_PREFIX || 'NXT'
81854.     generate (BLOCK_LABEL_PREFIX || 'SV')
81855.     generate ('DC A' || PREV_SAVE_TRACE_AREA || ')')
81856.     PREV_SAVE_AREA_PTR := FWD_PTR
81857.     /* Save label used as forward pointer. */
81858.     PREV_SAVE_TRACE_AREA := BLOCK_LABEL_PREFIX || 'SV'
81859.     /* Save name of this save area. */
81861.     generate ('DC (' || SAVE_LENGTH || ').FO'"
81862.     call PROC_XLLIFRASE_28W
81863.     C-86
"PROC" Macro -- 5 July 1973

81865. 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. */

81870. if DEBUG PROCCTAACE_REQD then
81871. if TRACE VECTOR_GENNED then /* Previously generated TRACE vector must be updated. */
81872. if WORK = 'NONE' and WORKREG_USED then
81873. generate (LABEL || ' LR 0,' || WORKREG)
81874. LABEL := ' ""
81875. if LABEL = '' then /* Branch around trace vector. */
81876. LABEL := BLOCK_LABEL_PREFIX || 'EE'
81877. TRASALS (' ""
81878. if LABEL = '' then /* Branch around trace vector. */
81879. LABEL := BLOCK_LABEL_PREFIX || 'EE'
81880. TRASALS (' ""
81881. if LABEL = '' then /* Branch around trace vector. */
81882. LABEL := BLOCK_LABEL_PREFIX || 'EE'
81883. TRASALS (' ""
81884. if LABEL = '' then /* Branch around trace vector. */
81885. LABEL := BLOCK_LABEL_PREFIX || 'EE'
81886. TRASALS (' ""
81887. if LABEL = '' then /* Branch around trace vector. */
81888. LABEL := BLOCK_LABEL_PREFIX || 'EE'
81889. TRASALS (' ""
81890. if LABEL = '' then /* Branch around trace vector. */
81891. LABEL := BLOCK_LABEL_PREFIX || 'EE'
81892. TRASALS (' ""
81893. if LABEL = '' then /* Branch around trace vector. */
81894. LABEL := BLOCK_LABEL_PREFIX || 'EE'
81895. TRASALS (' ""
81896. if LABEL = '' then /* Branch around trace vector. */
81897. LABEL := BLOCK_LABEL_PREFIX || 'EE'
81898. TRASALS (' ""
81899. if LABEL = '' then /* Branch around trace vector. */
81900. LABEL := BLOCK_LABEL_PREFIX || 'EE'
81901. TRASALS (' ""
81902. if LABEL = '' then /* Branch around trace vector. */
81903. LABEL := BLOCK_LABEL_PREFIX || 'EE'
81904. TRASALS (' ""
81905. if LABEL = '' then /* Branch around trace vector. */
81906. LABEL := BLOCK_LABEL_PREFIX || 'EE'
81907. TRASALS (' ""
81908. if LABEL = '' then /* Branch around trace vector. */
81909. LABEL := BLOCK_LABEL_PREFIX || 'EE'
81910. TRASALS (' ""
81911. if LABEL = '' then /* Branch around trace vector. */
81912. LABEL := BLOCK_LABEL_PREFIX || 'EE'
81913. TRASALS (' ""
81914. if LABEL = '' then /* Branch around trace vector. */
81915. LABEL := BLOCK_LABEL_PREFIX || 'EE'
81916. TRASALS (' ""
81917. if LABEL = '' then /* Branch around trace vector. */
81918. LABEL := BLOCK_LABEL_PREFIX || 'EE'
81919. TRASALS (' ""
81920. if LABEL = '' then /* Branch around trace vector. */
81921. LABEL := BLOCK_LABEL_PREFIX || 'EE'
81922. TRASALS (' ""
81923. if LABEL = '' then /* Branch around trace vector. */
81924. LABEL := BLOCK_LABEL_PREFIX || 'EE'
81925. TRASALS (' ""
81926. if LABEL = '' then /* Branch around trace vector. */
81927. LABEL := BLOCK_LABEL_PREFIX || 'EE'
81928. TRASALS (' ""
81929. if LABEL = '' then /* Branch around trace vector. */
81930. LABEL := BLOCK_LABEL_PREFIX || 'EE'
81931. TRASALS (' ""
81932. if LABEL = '' then /* Branch around trace vector. */
81933. LABEL := BLOCK_LABEL_PREFIX || 'EE'
81934. TRASALS (' ""
81935. if LABEL = '' then /* Branch around trace vector. */
81936. LABEL := BLOCK_LABEL_PREFIX || 'EE'
81937. TRASALS (' ""
81938. if LABEL = '' then /* Branch around trace vector. */
81939. LABEL := BLOCK_LABEL_PREFIX || 'EE'
81940. TRASALS (' ""
81941. if LABEL = '' then /* Branch around trace vector. */
81942. LABEL := BLOCK_LABEL_PREFIX || 'EE'
81943. TRASALS (' ""
81944. \end{verbatim}
"PROC" Macro -- 5 July 1973

81946. **PROC** PROC_INFO_SAVE
81947. /* Save all information needed at CORP time. */
81948. char i /* OFFSET_TO_CORP, biased by 50 and converted to character format. */
81949.
81950. OPERAND1(CURRENT_NEST_LEVEL) := FIRST
81951. OPERAND2(CURRENT_NEST_LEVEL) := LAST
81952. OPERAND3(CURRENT_NEST_LEVEL) := SAVE_LENGTH
81953. OPERAND4(CURRENT_NEST_LEVEL) := PREVIOUS_DEBUG_VECTOR
81954. i := OFFSET_TO_CORP + 50
81955. /* Bias value by 50 and convert to two-digit character string. */
81956. INFORMATION(CURRENT_NEST_LEVEL) :=
81957. i || GS_LINEAGE || (SAVE(3) = 'DYNAN') || FIRST_VALUE_KNOW ||
81958. PROC_IDBYTE || SPECIAL_PREFIX
81959. **END**
PROC Macro
5 July 1973

81962. PROC PROC_MULTIBASE
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 /* Preparation. */
81967. call X
81968.
81969. if MULTIBASE
81970. then /* Multibase adcons required but not yet generated. */
81971. if LABEL = ""
81972. then /* Search 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
generate('B' || BLOCK_LABEL_PREFIX || 'MBR' || XI)
81981. X := X + 1
81982. I := I + 1
81983. if (termination: I is incremented during loop, N'BASE is
81984. fixed: I must eventually exceed N'BASE.)
81985. MULTIBASE := false
81986. end
81987. end
81988. end
81989. end
81990. end
81991.
81992.
/* Defines the end of a procedure block. The register or registers indicated by RESTORE= are restored with the exception of those listed (except those in the RETURN= list). The return code is set from the RC= operand and return is made to the address specified by the LINK= operand, unless LINK=NOF is specified. */

/* (Ground rule: No CORP proc modifies CURRENT_NEST_LEVEL. This can be shown via the cross-reference listing.) */

int FIRST_SAVED_REG, /* Register number which may be placed into the first word of the save area. This may be, for example, 14 when the first register saved is some symbolic of unknown value. */
OFFSET_TO_R0, /* Offset in save area (in words) to the storage place for GPRO. This may be positive or negative. */

bit ANY_REGS_SAVED, /* Indicates whether any registers were saved in this proc. */
DYNAMIC_SAVED, /* Indicates whether SAVE=(***,DYNAM,*** was coded on PROC. */
BACK_AREA_BEGIN, /* Indicates whether BCK save area is needed. */
FIRST_VALUE_KNOW, /* Indicates whether FIRST is other than a symbolic. */
FIRST_REGISTER_KNOW, /* Indicates whether first register to be restored (in REST1) is other than symbolic. */
OS_LINKAGE, /* Indicates whether LINKAGE=(OS,*** was coded on PROC. */

char LABEL, /* Any outstanding label waiting to be generated. */

GPRO_OFFSET_STRING, /* OFFSET TO GPRO is character form. */
FIRST_REG_SAVED, /* First and last registers saved at PROC time. */
MULT, CORR, /* Holds either a "M" and "," respectively or else nulls to allow generation of either a "LM" or "L" instruction. */
LOCAL_POINTERS, OS_POINTERS, RESTORE_AREA, /* Instruction segments to generate load instructions from proper save area. */

RC_REG, /* Register holding return code before restoring of registers. */

SAVE_LENGTH, /* Length of save area. */
FIRST_REG, LAST_REG, /* First and last register to be restored. */
PROC_TO_GPRO, /* One-byte hex number used as identifier of current proc in traces and the like. */
PREVIOUS_DEBUGVECTOR, /* Value of debug switches (packed) before encountering this PROC or [if DEBUG=(...,GLOBAL) specified] null. */

CALL TRACE_PRINTER ( ; 'Corp')

/* Print macro name "CORP" as note if tracing on. */

if CURRENT_NEST_LEVEL > NESTING_LIMIT then
LABEL := USER_NAME

CALL POP_OLD_BLOCK ( ; )

fi

CALL VERIFY_END ( ; 'PROC', PROC_NAME)

/* Verifies current block has the name specified in the PROC_NAME operand on the CORP macro (if any) and that it is in a PROC block. Various errors receive messages and either intermediate blocks are "bleded" as a fixup or ERROR_OCCURRED is set. */

ERROR_OCCURRED will be set false and CURRENT_NEST_LEVEL will not be modified. */

if ERROR_OCCURRED then
ERROROccurred
fi

LABEL := USER_NAME

/* Generate label at first opportunity. */
COP Macro -- 6 July 1973

83074. call CORP_GET_PIOC_INFO
83075. /* Get info saved at PIOC macro: FIRST_REG_SAVED, LAST_REG_SAVED,
83076. OS_LINKAGE, FIRST_SAVAREA_REG, FIRST_VALUEKNOWN, DYNAMIC_SAVAREA,
83077. first_OFFSET, FIRST_OFFSET_REG, Offset_to_OFFSET, Prevents_DEBUG_VECTOR, PROC_TO_BYTE, */
83078. GPRO_OFFSET_STRING, BLOCK_LABEL_PREFIX, and ANY_REGS_SAVED. */
83079. call CORP_SET_RESTORE_RANGES
83080. /* Set RESTORE_AREA to 'SV'. Set REST1 and REST2 to RESTORE-
83081. special or, if omitted, then to FIRST_ and LAST_REG_SAVED. */
83082. call CORP_GET_EXIT_LABEL
83083. /* If an XL label is required, put it into LABEL (generating
83084. any label already there). */
83085. if OS_LINKAGE
83086. then
83087. call CORP_RESTORE_REGS
83088. /* Move register 11 pointer to point to previous save area, saving
83089. pointer to current area in register 1 if it is dynamic. */
83090. if DYNAMIC_SAVAREA
83091. then
83092. call CORP_RESTORE_DYN_SAVAREA
83093. /* Issue FREEMAP for dynamic core. */
83094. else
83095. call CORP_DEBUGGING_STORES
83096. /* If DEBUG_CORPVALUES_REQD, copy registers into CIP save area. If
83097. CORVALRES and RETURN- (or RC-) specified, copy SV save area to BCK
83098. save area, set RESTORE_AREA to 'BCK', and set BCK_AREA_REQD. */
83099. fi
83100. call CORP_SET_RETURNCODE
83101. /* If RC-value (or implied zero), load it to GPR15, except that if it
83102. is in a register other than 15, leave it in that register. */
83103. if DYNAMIC_SAVAREA
83104. call CORP_FREE_DYN_SAVAREA
83105. /* Issue FREEMAP for dynamic core. */
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 LN instruction will be issued.
83110. Also set OS_POINTER, LOCAL_POINTER, and restore proper save
83111. area. */
83112. if OS_LINKAGE and DEBUG_SAVETRACE_REQD
83113. then
83114. generate (LABEL = "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
8320. then
8321. generate LABEL := "X'FF'")
8322. LABEL := ""
8323. fi
8324. if LINK != "NONE"
8325. then
8326. generate (LABEL = "38 " || LINK)
8327. LABEL := ""
8328. fi
8329. call CORP_SETairobi
8330. /* Set that register. */
8331. generate LABEL := ""
8332. if LABEL != ""
8333. generate LABEL := ""
8334. if LABEL := "38"
8335. fi
8336. call POP_OLD_BLOCK ( )
8337. /* Delete PROC block from the stack. [Lemma: POP_OLD_BLOCK
8338. decreases CURRENT_NEST_LEVEL by exactly one.] */
8339. call CORP_RESET_DEBUG_ENVIRONMENT
8340. /* Restore value of debugging switches from PREVIOUS_DEBUG_VECTOR. */
8341. if (LABEL := ""
8342. [Lemma: If CURRENT_NEST_LEVEL > 3 and
8343. PROC_NAME = "" or = BLOCK_NAME(CURRENT_NEST_LEVEL)] and
8344. BLOCK_PTR(CURRENT_NEST_LEVEL) != PROC, then
8345. CURRENT_NEST_LEVEL will be decremented by exactly one.] */

C-91
83148. proc corp_getproc_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_GPRO := x[1,2] - 50 /* Stored biased by 50. */
83156. if OFFSET_TO_GPRO < 0
83157. then
83158. GPRO_OFFSET_STRING := '-'+ OFFSET_TO_GPRO
83159. else /* In string conversion, absolute value is taken; restore sign. */
83160. GPRO_OFFSET_STRING := '+'+ OFFSET_TO_GPRO
83161. fi
83162. OS_LINKAGE := x[3,1]
83164. DYNAMIC_SAVAREA := x[4,1]
83165. FIRST_VALUE_KNOWN := x[5,1]
83166. SAFE_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[7,1]
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_SAVAREA_REG := FIRST_REG_SAVED /* Convert to integer. */
83179. else
83180. FIRST_SAVAREA_REG := 14
83181. fi
83182. /* FIRST_SAVAREA_REG is similar to the variable SAP of PROC. */
83183. else
```
03185. DEFINE CORP_S2P_REGOP$ANCE
03186. /* Set REST1 and REST2 to RESTORE= operands if present, else to FIRST_ 
03187. and LAST_REG_SAVED. Set RESTORE_AREA to "SV". Also set MULT and 
03188. COMMA2 to proper values. */
03190. /* Assume: */
03191. FIRST1 := FIRST_REG_SAVED
03192. REST1 := LAST_REG_SAVED
03193. MULT := 'N'
03194. COMMA2 := 'U'
03195. FIRST_REST_VALUE_KNOW := FIRST_VALUE_KNOW
03196. /* Now find out. */
03197. if - ANY_REGS_SAVED
03198. then
03199. if RESTORE "A"
03200. then
03201. stderr (9, 'STBC8301 NO REGISTERS SAVED--RESTORE IGNORED')
03202. else
03203. else
03204. if RESTORE "A"
03205. then
03206. REST1 := RESTORE(1)
03207. REST2 := RESTORE(2)
03208. FIRST_REST_VALUE_KNOW := (RESTORE(1) = 'N')
03209. /* True iff first suboperand is a self-defining term. */
03210. if
03211. if REST2 = ""
03212. then
03213. MULT, COMMA2 := ""
03214. RESTORE_AREA := "SV"
03215. else
03216. endif
03217. endif
```
"CPRD" MACRO -- 6 JULY 1973

3219. RECG CORP_embers_siors
3220. */ Given: - U2_LINKAL. If DEBUG_CORPVALUES_REQD, store copy of
3221. register values into CPR save area. If CORPVALUES and RETURN (or
3222. RETURN) specified, copy SY save area to BCK save area, set
3223. RESTORE_AREA to "BCK", and set BCK_AREA_REQD. */
3224. 
3225. CRP_BCK_OFFSET, /* Character string to be inserted to insure
3226. registers are stored at proper offset from CRP/BCK label. */
3227. SV_OFFSET, /* Character string to be inserted to reference proper
3228. offset from SY save area label. */

3229. IF DEBUG_CORPVALUES_REQD
3230. THEN
3231. IF DEBUG_SAVEMACB_REQD
3232. THEN
3233. SV_OFFSET := '1'12'
3234. CRP_BCK_OFFSET := '1'12'
3235. ELSE
3236. CRP_BCK_OFFSET := '1'
3237. IF DEBUG_SUGCDREQS_REQD
3238. THEN
3239. SV_OFFSET := '1'1'
3240. ELSE
3241. SV_OFFSET := '1'
3242. FI
3243. FI
3244. GENERATE (LABEL || ' TN   14,12,11 || BLOCK_LABEL_SYMBOL ||
3245. 'CAP' || CRP_BCK_OFFSET)
3246. LABEL := '1'
3247. IF RETURN #: ' & (RC #: && 'NOTS')
3248. THEN
3249. CRP_BCK_ADDR := $123
3250. GENERATE ('
3251. '17C || BLOCK_LABEL_SYMBOL || 'BCK' ||
3252. CRP_BCK_OFFSET || '1'1' || SAVE_LENGTH || '4'),' ||
3253. BLOCK_LABEL_SYMBOL || 'SV' || SV_OFFSET)
3254. RESTORE_AREA := '1BCK'
3255. FI
3256. FI
3257. END
**CORP** Macro -- 6 July 1973

83259. PROC CORP_RESTORE_ASG13
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**. */
83263. if DYNAMIC_SAVEAREA
83264. then
83265. generate (LABEL || ' LR 1,13')
83266. fi
83267. if SAVETRACE_OA_FLAGS_PROC and PROC_ID_BYTE = '01'
83268. then
83269. generate (LABEL || ' L 13,EPB575F+4*)
83270. fi
83271. label := **
83272. C-95
"CORP" Macro -- 6 July 1973

83278. proc CORP_SRT_RETURN_CODE
83279. /* If RC=VALUE (or implied zero), load value into GPR15, but nop if
83280. RC=REG). Note in SC_REG what register (if any) contains RC at
83281. exit. */
83282.
83283. if RC = '"
83284. fi
83285.
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.
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. fi
83305. label (LABEL || 'LA 15,' || RC)
83306. fi
83307. LABEL := '
83308. RC_REG := '15'
83309. fi
83310. fi
83311. fi
83312. end
83314. PROC CORP_SAVE_RETIRNING_REGS
83315. /* For the register containing the return code and all those listed
83316. in RETJN=, 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
83322. if ANY_REGS_SAVE:
83323. then
83324. OS_LINKAGE := 0
83325. LOCAL_POINTER := '(13)'
83326. else
83327. LOCAL_POINTER := '***'
83328. fi
83329. if RC_REG # 15
83330. then
83331. LABEL := '(' LOCAL_LABEL_PREFIX RESTORE_AREA '4')
83332. offset := (OFFSET_TO_GPRO - 1) * 4
83333. generate (LABEL 'ST' RETURN(I) '(', LOCAL_POINTER '16*16)')
83334. else
83335. generate (LABEL 'ST' RETURN(I) '(', LOCAL_POINTER '16*16)')
83336. fi
83337. if FIRST_SAVAREA_REG # 14 and N'RETURN > 0
83338. then
83339. write ('STRCB392 WARNING—NO CHECK MADE TO INSURE RETURNING
83400. ' REGISTERS ARE AMONG THOSE SAVED IN TRUNCATED SAVE AREA')
83401. fi
83402. I := 1
83403. while I < N'RETURN do
83404. if RETURN(I) = 'N' /* Self-defining term. */
83405. then
83406. offset := (OFFSET_TO_GPRO + RETURN(I) - (RETURN(I) + 2)/16*16) * 4
83407. generate (LABEL 'ST' RETURN(I) '16*16)'
83408. else
83409. generate (LABEL 'ST' RETURN(I) '16*16)'
83410. fi
83411. fi
83412. I := I + 1
83413. label := 1
83414. od /* termination: I is incremented, N'RETURN is fixed in loop;
83415. I must eventually exceed N'RETURN. */
"CORP" macro -- 6 July 1973

83381. PROC CORP_FREE_DYNAMIC_SAVEAREA
83382. /* Issue FREEMAIN for dynamic save area. */
83383. GENERATE (LABEL || ' LA 0,' || SAVE_LENGTH || '*')
83384. LABEL := '
83385. GENERATE (' FREEMAIN R,LV=(0),A=(1)')
83386. CORD
83369. PROC CORP_RESTORE_REGISTERS
83370. /* Restore registers REST1 through REST2 from proper save area if
83371. saved. */
83372.印度 OFFSET, I /* Temporaries. */
83373. EY ANY_REGS_SAVED
83374. then
83375. if FIRST_REGS_VALUE_KNOWN
83376. else
83377. I := REST1 /* Convert to integer. */
83378. OFFSET := (OFFSET_TO_GPRO • I - (I + 2)/16*16) • 4
83379. GPRO_TO_GPRO := (LABEL • ' L' • MOLT • ' ' • REST1 • ' ' • ' ' • REST2 • ' ' • GPRO_OFFSET_STRING • ' ' • ' ' • OS_POINTER)
83380. GPRO_OFFSET_STRING • ' ' • ' ' • OS_POINTER
83381. end
83382. if
83383. then
83384. GPRO_OFFSET_STRING • ' ' • ' ' • OS_POINTER
83385. REST2 • ' ' • LOCAL_POINTER • ' ' • ' ' • REST1 • ' ' • REST2 • ' ' • LOCAL_POINTER • ' ' • ' ' • REST1 • ' ' • REST2 • ' ' • LOCAL_POINTER • ' ' • ' ' • REST1 • ' ' • REST2 • ' ' • LOCAL_POINTER • ' ' • ' ' • REST1 • ' ' • ' ' • OS_POINTER
83386. else
83387. end
83388. REST1 • ' ' • LOCAL_POINTER • ' ' • ' ' • REST1 • ' ' • ' ' • OS_POINTER
83389. REST1 • ' ' • LOCAL_POINTER • ' ' • ' ' • REST1 • ' ' • ' ' • OS_POINTER
83390. REST1 • ' ' • LOCAL_POINTER • ' ' • ' ' • REST1 • ' ' • ' ' • OS_POINTER
83391. REST1 • ' ' • LOCAL_POINTER • ' ' • ' ' • REST1 • ' ' • ' ' • OS_POINTER
83392. end
83393. LABEL := ''
83394. C-99
/* CORP macro -- 6 July 1973 */

93418.  BLOC CORP_RESTORE_DEBUG_ENVIRONMENT
93419.  /* Restore debug flags which were in progress before the PROC (unless
93420.  inherited from a null raise to suppress restore). Values are packed
93421.  in PREVIOUS_DEBUG_VECTOR and need only be unpacked. */

93422.  if PREVIOUS_DEBUG_VECTOR ==
93423.  then
93424.  debug_block_vectors_req := PREVIOUS_DEBUG_VECTOR[1,1]
93425.  debug_program_vectors_req := PREVIOUS_DEBUG_VECTOR[2,1]
93426.  debug_lock_vectors_req := PREVIOUS_DEBUG_VECTOR[3,1]
93427.  debug_block_counts_req := PREVIOUS_DEBUG_VECTOR[4,1]
93428.  debug_proc_counts_req := PREVIOUS_DEBUG_VECTOR[5,1]
93429.  debug_process_vectors_req := PREVIOUS_DEBUG_VECTOR[6,1]
93430.  debug_cpu_values_req := PREVIOUS_DEBUG_VECTOR[7,1]
93431.  debug_save_vectors_req := PREVIOUS_DEBUG_VECTOR[8,1]
93432.  fi
93433.  see
`CORP` Macro -- 6 July 1973

```
83436. PROC CORP_3GCRPAREA
83437. /* if received, generate CRP and BCK save areas. */
83438. LAST_AREA, /* label of CRP or BCK area, whichever is generated
83439. LAST_AREA. */
83440. FWD_PTR, /* label generated as forward pointer in last area. */
83441. TARGET /* temporary. */
83442.
83443. if OS_LINKAGE and DEBUG_CAREAREQD
83444. then /* we need a CRP save area. */
83445. if LINK = '033Z'
83446. then /* we need a CRP save area. */
83447. if OS_LINKAGE and DEBUG_CAREAREQD
83448. then /* we need a CRP save area. */
83449. then /* we need a CRP save area. */
83450. then /* we need a CRP save area. */
83451. then /* we need a CRP save area. */
83452. then /* we need a CRP save area. */
83453. then /* we need a CRP save area. */
83454. then /* we need a CRP save area. */
83455. then /* we need a CRP save area. */
83456. then /* we need a CRP save area. */
83457. then /* we need a CRP save area. */
83458. then /* we need a CRP save area. */
83459. then /* we need a CRP save area. */
83460. then /* we need a CRP save area. */
83461. then /* we need a CRP save area. */
83462. then /* we need a CRP save area. */
83463. then /* we need a CRP save area. */
83464. then /* we need a CRP save area. */
83465. then /* we need a CRP save area. */
83466. then /* we need a CRP save area. */
83467. then /* we need a CRP save area. */
83468. then /* we need a CRP save area. */
83469. then /* we need a CRP save area. */
83470. then /* we need a CRP save area. */
83471. then /* we need a CRP save area. */
83472. then /* we need a CRP save area. */
83473. then /* we need a CRP save area. */
83474. then /* we need a CRP save area. */
83475. then /* we need a CRP save area. */
83476. then /* we need a CRP save area. */
83477. then /* we need a CRP save area. */
83478. then /* we need a CRP save area. */
83479. then /* we need a CRP save area. */
83480. then /* we need a CRP save area. */
83481. then /* we need a CRP save area. */
83482. then /* we need a CRP save area. */
83483. then /* we need a CRP save area. */
83484. then /* we need a CRP save area. */
83485. then /* we need a CRP save area. */
83486. then /* we need a CRP save area. */
83487. then /* we need a CRP save area. */
83488. then /* we need a CRP save area. */
83489. then /* we need a CRP save area. */
83490. then /* we need a CRP save area. */
83491. then /* we need a CRP save area. */
83492. then /* we need a CRP save area. */
83493. then /* we need a CRP save area. */
83494. then /* we need a CRP save area. */
83495. then /* we need a CRP save area. */
83496. then /* we need a CRP save area. */
83497. then /* we need a CRP save area. */
83498. then /* we need a CRP save area. */
83499. then /* we need a CRP save area. */
83500. then /* we need a CRP save area. */
83501. then /* we need a CRP save area. */
83502. then /* we need a CRP save area. */
83503. then /* we need a CRP save area. */
83504. then /* we need a CRP save area. */
83505. then /* we need a CRP save area. */
83506. then /* we need a CRP save area. */
83507. then /* we need a CRP save area. */
83508. then /* we need a CRP save area. */
83509. then /* we need a CRP save area. */
83510. then /* we need a CRP save area. */
83511. then /* we need a CRP save area. */
83512. then /* we need a CRP save area. */
83513. then /* we need a CRP save area. */
83514. then /* we need a CRP save area. */
83515. then /* we need a CRP save area. */
83516. then /* we need a CRP save area. */
83517. then /* we need a CRP save area. */
83518. then /* we need a CRP save area. */
83519. then /* we need a CRP save area. */
83520. then /* we need a CRP save area. */
83521. then /* we need a CRP save area. */
83522. then /* we need a CRP save area. */
83523. then /* we need a CRP save area. */
83524. then /* we need a CRP save area. */
83525. then /* we need a CRP save area. */
83526. then /* we need a CRP save area. */
83527. then /* we need a CRP save area. */
83528. then /* we need a CRP save area. */
83529. then /* we need a CRP save area. */
83530. then /* we need a CRP save area. */
83531. then /* we need a CRP save area. */
83532. then /* we need a CRP save area. */
83533. then /* we need a CRP save area. */
83534. then /* we need a CRP save area. */
83535. then /* we need a CRP save area. */
83536. then /* we need a CRP save area. */
83537. then /* we need a CRP save area. */
83538. then /* we need a CRP save area. */
83539. then /* we need a CRP save area. */
83540. then /* we need a CRP save area. */
83541. then /* we need a CRP save area. */
83542. then /* we need a CRP save area. */
83543. then /* we need a CRP save area. */
83544. then /* we need a CRP save area. */
83545. then /* we need a CRP save area. */
83546. then /* we need a CRP save area. */
83547. then /* we need a CRP save area. */
83548. then /* we need a CRP save area. */
83549. then /* we need a CRP save area. */
83550. then /* we need a CRP save area. */
83551. then /* we need a CRP save area. */
83552. then /* we need a CRP save area. */
83553. then /* we need a CRP save area. */
83554. then /* we need a CRP save area. */
83555. then /* we need a CRP save area. */
83556. then /* we need a CRP save area. */
83557. then /* we need a CRP save area. */
83558. then /* we need a CRP save area. */
83559. then /* we need a CRP save area. */
83560. then /* we need a CRP save area. */
83561. then /* we need a CRP save area. */
83562. then /* we need a CRP save area. */
83563. then /* we need a CRP save area. */
83564. then /* we need a CRP save area. */
83565. then /* we need a CRP save area. */
83566. then /* we need a CRP save area. */
83567. then /* we need a CRP save area. */
83568. then /* we need a CRP save area. */
83569. then /* we need a CRP save area. */
83570. then /* we need a CRP save area. */
83571. then /* we need a CRP save area. */
83572. then /* we need a CRP save area. */
83573. then /* we need a CRP save area. */
83574. then /* we need a CRP save area. */
83575. then /* we need a CRP save area. */
83576. then /* we need a CRP save area. */
83577. then /* we need a CRP save area. */
83578. then /* we need a CRP save area. */
83579. then /* we need a CRP save area. */
83580. then /* we need a CRP save area. */
83581. then /* we need a CRP save area. */
83582. then /* we need a CRP save area. */
83583. then /* we need a CRP save area. */
83584. then /* we need a CRP save area. */
83585. then /* we need a CRP save area. */
83586. then /* we need a CRP save area. */
```

C-101
"CORR" Macro -- 6 July 1973

83498.  PIES COMP_1M_EXIT_LABEL
83499.  /* If an EXIT label is required, put it into LABEL (generating
83500.  any label already there). Issue a note regarding EXIT references. */
83501.  if EXIT_LABEL_1DD(CURRENT_NEST_LEVEL)
83502.      then
83503.          if LABEL # ""
83504.              THEN
83505.                  (LABEL || ' DS 0H')
83506.              fi
83507.              if EXIT_SEVERITY = '4'
83508.                  THEN
83509.                  EXIT_SEVERITY := '4'
83510.                  fi
83511.              fi
83512.          fi
83513.          LABEL := BLCK_LABEL_PREFIX || 'EXIT'
83514.          fi
83515.      fi
83516.  endif
83517.  endif
83518.  endif
83519.  endif
83520.  endif
83521.  endif
83522.  endif
83523.  endif
83524.  endif
83525.  endif
"EXIT_FIND" Macro -- 10 July 1973

91001. RACES 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 EXIT label. */
91006. int I /* Temporary */
91007.
91008. call track_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 # '*' then /* We must search for the right block. */
91014. while I > 0 and REQD_NAME # BLOCK_NAME(I) and
91015. BLOCK_TYPE(I) = 'PROC'
91016. do
91017. I := I - 1
91018. od /* (Termination: I is decremented and would eventually become
91019. < 0 even if other tests never occurred.) */
91020. if I = 0 or
91021. (REQD_NAME = BLOCK_NAME(I) and # ''' and # '*')
91022. then /* Not found in search. */
91023. error_occurred := true
91024. if REQD_NAME = '' or = '''
91025. and BLOCK_NAME(I) = 'DO' and
91026. INFORMATION(I)[6,1] or INFORMATION(I)[7,1]
91027. then /* Must use special DO prefix form. */
91028. error_occurred := true
91029. else if BLOCK_NAME(I) = 'DO' and INFORMATION(I)[6,1]
91030. then /* Just use special PROC prefix form. */
91031. ULTIMATE_BRANCH_LABEL := '$P' || INFORMATION(I)[6,2] || 'EXIT'
91032. else
91033. ULTIMATE_BRANCH_LABEL := '' || BLOCK_NUMBER(I) || 'EXIT'
91034. fi
91035. fi
91036. fi
91037. fi
91038. fi
91039. if BLOCK_TYPE(I) = 'DO' and
91040. INFORMATION(I)[6,1] or INFORMATION(I)[7,1]
91041. then /* Stuck in a loop. */
91042. error_occurred := true
91043. else
91044. ULTIMATE_BRANCH_LABEL := '$P' || INFORMATION(I)[6,2] || 'EXIT'
91045. fi
91046. error_occurred := true
91047. if BLOCK_TYPE(I) = 'DO' and INFORMATION(I)[6,1]
91048. then /* Just use special PROC prefix form. */
91049. ULTIMATE_BRANCH_LABEL := '$P' || INFORMATION(I)[6,2] || 'EXIT'
91050. else
91051. ULTIMATE_BRANCH_LABEL := '' || BLOCK_NUMBER(I) || 'EXIT'
91052. fi
91053. fi
91054. fi
91055. fi
91056. fi
91057. fi
91058. fi
91059. return
"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. */
92004. call TRACE_PRINTER ( ; "POP")
92005. /* Call macro name "POP" in note if tracing on. */
92006. if CURRENT_NEST_LEVEL < NESTING_LIMIT
92007. then
92008. if END_LABEL_REQD(CURRENT_NEST_LEVEL)
92009. then
92010. generate ('$' || BLOCK_NUMBER(CURRENT_NEST_LEVEL) || 'END DS OH')
92011. fi
92012. if EXIT_LABEL_REQD(CURRENT_NEST_LEVEL) or OLD_EXIT #
92013. then
92014. then
92015. if EXIT_SEVERITY = *
92016. then
92017. EXIT_SEVERITY := '0'
92018. fi
92019. if EXIT_SEVERITY,
92020. "STC9201 ONE OR MORE EXIT'S REFERENCE THIS POINT"
92021. fi
92022. if EXIT_LABEL_REQD(CURRENT_NEST_LEVEL)
92023. then
92024. generate ('$' || BLOCK_NUMBER(CURRENT_NEST_LEVEL) || 'XIT DS OH')
92025. fi
92026. if OLD_EXIT #
92027. then
92028. generate (OLD_EXIT || ' DS OH')
92029. fi
92030. if DEBUG_LIST_BLOCKS_REQD
92031. then
92032. notes (*, "STC9203 END OF BLOCK '|| BLOCK_NUMBER(CURRENT_NEST_LEVEL)
92033. || '(' || BLOCK_NAME(CURRENT_NEST_LEVEL) || ') AT DEPTH '||
92034. CURRENT_NEST_LEVEL)
92035. notes (*, "********************************************************************************")
92036. fi
92037. fi
92038. CURRENT_NEST_LEVEL := CURRENT_NEST_LEVEL - 1
92039. /* (Lemma: Execution of POP_OLD_BLOCK always decrements
92040. CURRENT_NEST_LEVEL by exactly one.) */

*/
PUSH_BLOCK
Macro -- 10 July 1973

93001. MACRO PUSH_BLOCK (BLOCK_NAME_VALUE;)
93002. BLOCK_TYPE_VALUE="", OPERAND1_VALUE="",
93003. OPERAND2_VALUE="", OPERAND3_VALUE="",
93004. INFORMATION_VALUE="", END_LABEL_VALUE=false)
93005. /* Define new block; add to stack. Save block specifications.
93006. All macro operands unspecified default to null string except
93007. END_LABEL_VALUE defaults to false. */
93008. CALL_TRACEPRINTER ("PUSH")
93009. /* Print macro name "PUSH" in mnote if tracing on. */
93010. NESTING_LIMIT := 100
93011. /* Insure maximum depth of stack is set in variable. Note that stack
93012. depth and this variable must match, but may be changed to any
93013. value. */
93014. SCASE_NESTING_LIMIT := 9
93015. /* Same for general CASE stack. */
93016. ERROR_OCCURRED := false
93017. CURRENT_NEST_LEVEL := CURRENT_NEST_LEVEL + 1
93018. if CURRENT_NEST_LEVEL > NESTING_LIMIT
93019. then
93020. "STACK NESTING LIMIT OF \$NESTING_LIMIT\$ EXCEEDED--MACROS
93021. MUST BE MODIFIED"
93022. ERROR_OCCURRED := true
93023. else
93024. LAST BLOCK NUMBER := LAST BLOCK NUMBER + 1
93025. /* Set block number for this block. */
93026. BLOCK_NAME(CURRENT_NEST_LEVEL) := BLOCK_NAME_VALUE = ""
93027. if BLOCK_NAME_VALUE = ""
93028. then
93029. BLOCK_NAME(CURRENT_NEST_LEVEL) := 'BLK' \$LAST BLOCK NUMBER\$
93030. Block_label_prefix := 'B' \$LAST BLOCK NUMBER\$
93031. if CURRENT_NEST_LEVEL > 1 and
93032. BLOCK_TYPE(CURRENT_NEST_LEVEL-1) = 'DOCASE' and
93033. BLOCK_TYPE_VALUE = 'CASE'
93034. then
93035. "ERROR (A, 'STACK9301 BLOCK NESTING LIMIT OF \$NESTING_LIMIT \$
93036. ' EXCEEDED--MACROS MUST BE MODIFIED')"
93037. ERROR_OCCURRED := true
93038. else
93039. LAST_BLOCK_NUMBER := LAST BLOCK NUMBER + 1
93040. /* Set block number for this block. */
93041. BLOCK_NAME(CURRENT_NEST_LEVEL) := BLOCK_NAME_VALUE = ""
93042. if BLOCK_NAME_VALUE = ""
93043. then
93044. BLOCK_NAME(CURRENT_NEST_LEVEL) := 'BLK' \$LAST BLOCK NUMBER\$
93045. Block_label_prefix := 'B' \$LAST BLOCK NUMBER\$
93046. if CURRENT_NEST_LEVEL > 1 and
93047. BLOCK_TYPE(CURRENT_NEST_LEVEL-1) = 'DOCASE' and
93048. BLOCK_TYPE_VALUE = 'CASE'
93049. then
93050. /* Stack non-case block immediately surrounded by DOCASE invalid */
93051. ERROR_OCCURRED := true
93052. if DEBUG_LISTBLOCKS_HAVE
93053. then
93054. "ERROR (A, 'CUSTOMHOUSE NESTING LIMIT EXCEEDED--MACROS
93055. MUST BE MODIFIED')"
93056. ERROR_OCCURRED := true
93057. BLOCK_NAME(CURRENT_NEST_LEVEL) := BLOCK_NAME_VALUE = ""
93058. if BLOCK_NAME_VALUE = ""
93059. then
93060. "ERROR (A, 'STACK9301 BLOCK NESTING LIMIT OF \$NESTING_LIMIT \$
93061. ' EXCEEDED--MACROS MUST BE MODIFIED')"
93062. ERROR_OCCURRED := true
"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_COND) 
94010. /* Separate indicated instruction followed by appropriate conditional 
94011. branch to indicated label. */ 
94012. 
94013. char LOCAL_MASK, LOCAL_REL, /* Holds mask or relation for branch. */ 
94014. BCTAG /* Label to go on bc instruction. */ 
94015. 
94016. call TRACE_PRINTER (; "SIMPCOND") 
94017. /* Print macro name "SIMPCOND" in note if tracing on. */ 
94018. call SIMPCOND_ONE_MASK OR REL 
94019. /* Extract LOCAL_MASK or LOCAL_REL from OPER's. If LOCAL_REL is an 
94020. external value (GT, GE, EQ, LT, or LE), replace it with the 
94021. proper value (H, AL, E, L, or NH). */ 
94022. 
94023. case OP_COUNT 
94024. case 0 /* Mask or relation only. */ 
94025. BCTAG := LABEL 
94026. esac 
94027. case 1 /* Mask or relation only. */ 
94028. BCTAG := LABEL 
94029. esac 
94030. case 2 /* Mask or relation only. */ 
94031. BCTAG := LABEL 
94032. esac 
94033. case 3 /* Mask or relation only. */ 
94034. BCTAG := LABEL 
94035. esac 
94036. case 4 /* Mask or relation only. */ 
94037. BCTAG := LABEL 
94038. esac 
94039. case 5 /* Mask or relation only. */ 
94040. BCTAG := LABEL 
94041. esac 
94042. case misc /* Mask or relation only. */ 
94043. mnote(3, 'STRING01 INSUFFICIENT OPERANDS FOR TEST "" || OP_CODE || 
94044. "") 
94045. esac 
94046. if LOCAL_MASK == 
94047. then 
94048. if FALLTHRU_CONDITION /* is true: */ 
94049. then /* Invert relation. */ 
94050. if LOCAL_REL[1,1] == 'N' 
94051. then 
94052. LOCAL_REL := LOCAL_REL[2,7] 
94053. esac 
94054. LOCAL_REL := 'H' || LOCAL_REL 
94055. fi 
94056. esac 
94057. else 
94058. generate (BC_TAG || 'H' || LOCAL_REL || ' ' || BRANCH_LABEL) 
94059. esac 
94060. endif 
94061. endif /* Invert mask. */ 
94062. generate (BC_TAG || 'E' || LOCAL_MASK || ' ' || BRANCH_LABEL) 
94063. endif /* Invert mask. */ 
94064. generate (BC_TAG || 'E' || LOCAL_MASK || ' ' || BRANCH_LABEL) 
94065. fi 
94066. fi 
94067. endif /* Invert mask. */ 
94068. endif /* Invert mask. */ 
```
"SIMPLE_CONDITIONAL" macro -- 10 July 1973

94070. DOES SIMPCOND_GET_MASK_REL
94071. /* Extract LOCAL_ASK or LOCAL_REL from OPER's. If LOCAL_REL is a
94072.   external value (GT, GE, EQ, LT, or LE), replace it with the
94073.   proper value (H, NL, E, L, or NH). */
94074. 
94075.   if OP_COUNT = 0
94076.     error (8, 'STRC9403 NO CONDITION SPECIFIED—"MASK-0" ASSUMED')
94077.     LOCAL_ASK := '0'
94078.   else
94079.     if SYSLIST(OP_COUNT)[1,5] = 'MASK-
94080.       LOCAL_ASK := SYSLIST(OP_COUNT)[6,8]
94081.     else
94082.       LOCAL_ASK := SYSLIST(OP_COUNT)[1,4] - 'REL-
94083.         LOCAL_REL := SYSLIST(OP_COUNT)[5,8]
94084.       else
94085.         LOCAL_ASK := SYSLIST(OP_COUNT)
94086.         fi
94087.         fi
94088.         fi
94089.
94090.         fi
94091.
94092.         fi
94093.         fi
94094.         fi
94095.         fi
94096.         fi
94097.         fi
94098.         fi
94099.         fi
94100.         fi
94101.         fi
94102.         fi
94103.         fi
94104.         fi
94105.         fi
94106.         fi
94107.         fi
94108.         fi
94109.         fi
94110.         fi
94111.  fi
94112.  fi
94113.  fi
94114.  fi
94115.  fi
94116.  fi
94117.  fi
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. endif
95006. endif (*, 'SZRC9500 ' || MACRO_NAME)
95007. endif
95008. endif
macro VERIFY END ( : REQD_TYPE, REQD_NAME)

/* Verifies current block has name specified by REQD_NAME operand, if any, and that it is of type REQD_TYPE. Various errors receive messages and either intermediate blocks are BLENDed as a fixup or ERROR_OCCURRED is set. */

# Temporary.
VERIFY MACRO -- 10 July 1973

96008. call TRACE_PRINTER ( ; 'VERIFY')
96009. /* Print macro name "VERIFY" in anote if tracing on. */
96010. ERROR_OCCURRED := false /* Assumed. */
96011. if REQ_NAME = ''
96012. then
96013. if CURRENT_NEST_LEVEL < 0
96014. then
96015. note (6, 'STRC9604 NO BLOCKS ACTIVE--MACRO IGNORED')
96016. ERROR_OCCURRED := true
96017. else
96018. if CURRENT_NEST_LEVEL < 1
96019. then
96020. BLOCK_TYPE(CURRENT_NEST_LEVEL) # REQD_TYPE
96021. then
96022. note (6, 'STRC9601 ONE BLEND ASSUMED TO GET TO "' || REQD_TYPE || '" BLOCK')
96023. call BLEND ( ; )
96024. else
96025. note (6, 'STRC9602 TWO BLENDS ASSUMED TO GET TO "' || REQD_TYPE || '" BLOCK')
96026. call BLEND ( ; )
96027. call BLEND ( ; )
96028. if CURRENT_NEST_LEVEL < 3
96029. then
96030. BLOCK_TYPE(CURRENT_NEST_LEVEL - 2) = REQD_TYPE
96031. then
96032. note (6, 'STRC9603 CURRENT BLOCK NOT "' || REQD_TYPE || '" BLOCK--MACRO IGNORED')
96033. ERROR_OCCURRED := true
96034. else
96035. note (6, 'STRC9605 BLOCK "' || REQD_NAME || '" IS NOT A ' || REQD_TYPE || ' BLOCK')
96036. ERROR_OCCURRED := true
96037. else
96038. if REQD_TYPE # BLOCK_TYPE(I)
96039. then
96040. note (6, 'STRC9606 NO ACTIVE BLOCK NAMED "' || REQD_NAME || '"')
96041. ERROR_OCCURRED := true
96042. else
96043. /* A block name was specified. */
96044. I := CURRENT_NEST_LEVEL
96045. while I > 0 and REQ_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. note (6, '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. note (6, 'STRC9605 BLOCK "' || REQD_NAME || '" IS NOT A ' || REQD_TYPE || ' BLOCK--MACRO IGNORED')
96058. ERROR_OCCURRED := true
96059. else
96060. while CURRENT_NEST_LEVEL > I
96061. do /* BLEND any intermediate blocks. */
96062. note (6, 'STRC9606 END OF BLOCK "' || REQD_NAME || '"
96063. IMPLIES END OF BLOCK "' || BLOCK_NAME(CURRENT_NEST_LEVEL) || '"
96064. BLOCK_NAME(CURRENT_NEST_LEVEL) || '"
96065. call BLEND ( ; ) /* (Lemma: If CURRENT_NEST_LEVEL > 0 and no BLEND operands
96066. specified, BLEND will decrement CURRENT_NEST_LEVEL
96067. by exactly one.) */
96068. od /* (Termination: On all iterations, I is fixed and 
96069. CURRENT_NEST_LEVEL > I > 0. But BLEND decrements 
96070. CURRENT_NEST_LEVEL. Therefore, CURRENT_NEST_LEVEL must 
96071. eventually become = (actually =) I. */) */
96072. fi
96073. fi
96074. fi
96075. fi
96076. fi
96077. /* (Lemma: If CURRENT_NEST_LEVEL > 0 and 
96078. [REQ_NAME = ' ' or BLOCK_TYPE(CURRENT_NEST_LEVEL) # REQD_TYPE = BLOCK_TYPE(CURRENT_NEST_LEVEL), then 
96079. ERROR_OCCURRED will always be set false and CURRENT_NEST_LEVEL 
96080. will be unmodified. Proof: If the hypothesized conditions are 
96081. true, the module calls TRACE_PRINTER and sets ERROR_OCCURRED to 
96082. false as its only actions. TRACE_PRINTER modifies no 
96083. globals.) */

C-110
"CONDITIONAL_EXPRESSION_PROCESSOR" Macro -- 9 July 1973

```
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. 
97014. 
97015. 
97016. 
97017. 
97018. 
97019. 
97020. 
97021. 
97022. 
97023. 
97024. 
97025. 
97026. 
97027. 
97028. 
97029. 
97030. 
97031. 
97032. 
97033. 
97034. 
97035. 
97036. 
97037. 
97038. 
97039. 

97040. C-111
```
"CONDITIONAL_EXPRESSION_PROCESSOR" Macro -- 9 July 1973

97041.  call TRACE_PRINTER ( ; 'CEP')
97042.  */ Print macro name "CEP" in note if tracing on. */
97043.  LABEL := FIRST_INDEX
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.    */ Step INDEX up to next simple conditional increasing DEPTH for any
97052.    */ which found and setting ERROR_OCCURRED on any syntax error. Increments
97053.    */ COND_COUNT for the condition found. */
97054.    if ERROR_OCCURRED
97055.    then
97056.    BEXIT
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_LABELUSED. 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.  BEXIT
97069.  fi
97070.  if LOCAL_LABELS_REQ(COND_COUNT)
97071.  then
97072.  LABEL := BLOCK_LABEL_PREFIX || UNIQUE_LABEL_ID || COND_COUNT
97073.  fi
97074.  OP_COUNT := w'SYSTABLE(INDEX)
97075.  call SIMPLE_CONDITIONAL (LABEL);
97076.  SYSTABLE(INDEX,1)
97077.  SYSTABLE(INDEX,2)
97078.  SYSTABLE(INDEX,3)
97079.  SYSTABLE(INDEX,4)
97080.  SYSTABLE(INDEX,5)
97081.  LOCAL_BRANCH_LABEL
97082.  LOCAL_FALLTHRU_CONDITION,
97083.  OP_COUNT)
97084.  LABEL := w'T
97085.  INDEX := NEXT_INDEX
97086.  DEPTH := NEXT_DEPTH
97087.  go /* [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.  message (0, "STRC9701 INSUFFICIENT OPERANDS")
97093.  fi
97094.  if DEPTH ≠ 0
97095.  then
97096.  message (0, "STRC9702 INSUFFICIENT BRACKETS")
97097.  fi
97098.  end
"CONDITIONAL_EXPRESSION_PROCESSOR" Macro -- 9 July 1973

57100. proc CEP_FIND_NEXT_CONDITION
57101. /* Step up to next simple conditional, incrementing DEPTH for any "<"
57102. (or "=") found. Start with SYSLIST(INDEX) and advance INDEX till
57103. found or LAST_INDEX. Set ERROR_OCCURED on any syntax error,
57104. giving message. Increment COND_COUNT by 1. */
57105.
57106. ERROR_OCCURED := false
57107. while INDEX ≤ LAST_INDEX and (SYSLIST(INDEX) = '<' or = '==')
57108. do
57109. DEPTH := DEPTH + 1
57110. INDEX := INDEX + 1
57111. od
57112. /* (Termination: INDEX is incremented, LAST_INDEX is fixed in
57113. loop: INDEX would eventually exceed LAST_INDEX even if other
57114. tests never occur.) */
57115. if SYSLIST(INDEX)[1,1] = '('
57116. then
57117. mnote (8, 'STRC9706 SYNTAX ERROR--LOOKING FOR SIMPLE '<'
57118. 'CONDITIONAL, FOUND ' || SYSLIST(INDEX) || '==')
57119. ERROR_OCCURED := true
57120. fi
57121. COND_COUNT := COND_COUNT + 1
PROC CEP_LOOKAHEAD

57123. /* Search operands beyond current simple conditional. If AND/OR found,
57124. label found or generated for that spot into LOCAL_BRANCH_LABEL. If
57125. case as ULTIMATE_FALLTHRU_LABEL, set FALLTHRU_LABEL_USED. Decide
57126. whether this test is to fallthru on true or false and set
57127. LOCAL_FALLTHRU_CONDITION. If syntax error found during lookahead,
57128. give message and set ERROR_OCCURRED. Also set NEXT_INDEX and
57129. NEXT_DEPTH with the index/depth of the AND/OR. */
57130. 
57131. ERROR_OCCURRED := false
57132. LA_DEPTH := DEPTH
57133. I := INDEX + 1
57134. while I ≤ LAST_INDEX and (SYSLIST(I) = '>' or I = '/')
57135. do
57136. I := I + 1
57137. if LA_DEPTH > 0
57138. then
57139. LA_DEPTH := LA_DEPTH - 1
57140. else
57141. next INDEX := I + 1
57142. fi
57143. /* (Termination: I is incremented; LAST_INDEX is fixed in
57144. loop; INDEX would eventually exceed LAST_INDEX even if other
57145. tests never occur.) */
57146. NEXT_INDEX := I + 1
57147. /* (Lemma: NEXT_INDEX > I > INDEX.) */
57148. NEXT_DEPTH := LA_DEPTH
57149. LOCAL_BRANCH_LABEL := ULTIMATE_BRANCH_LABEL
57150. LOCAL_FALLTHRU_CONDITION := ULTIMATE_FALLTHRU_CONDITION
57151. else
57152. if SYSLIST(I) = 'AND' or I = 'OR'
57153. then
57154. CEPSCAN_FCR_BRANCH
57155. /* Search ahead for branch target. Set AND_OR_OUTSTANDING, LOCAL_
57156. FALLTHRU_CONDITION, and LOCAL_SEARCH_POINT. */
57157. else
57158. ERROR_OCCURRED := true
57159. /* (Lemma: The value of NEXT_INDEX returned is greater than the
57160. value of INDEX entered.) */
57161. fi
57162. fi
57163. */
CONDITIONAL_EXPRESSION_PROCESSOR Macro -- 9 July 1973

97171.  BEGN 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.BECD 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.   LOCKFOR /* Operation ("AND" or "OR") opposite of SYSLIST(I). */
97182.   int MAX_DEPTH, /* The maximum depth at which the LOOKFOR'ed
97183.   operation is a possible branch target. */
97184.   TARGET /* Simple conditional number which is the target for
97185.   the branch. */
AND_OB_OUTSTANDING := false
if SYSLIST(I) = 'AND'
then
LOCAL_FALLTHRU_CONDITION := false
MAX_DEPTH := LA_DEPTH
else /* Operation is OR */
LOCAL_FALLTHRU_CONDITION := false
if LA_DEPTH = 0
then
I := LAST_INDEX + 1
*/ Advance I to force skip of unnecessary search. */
else
LOCPOS := 'AND'
MAX_DEPTH := LA_DEPTH - 1
fi
I := I + 1
TARGET := COND_COUNT + 1
while I <= LAST_INDEX and
(SYSLIST(I) ≠ LOCPOS OR LA_DEPTH > MAX_DEPTH)
do if SYSLIST(I) = '<' OR = '*'
then
LA_DEPTH := LA_DEPTH + 1
else
if SYSLIST(I) = '>' or = '/'
then
if LA_DEPTH > 0
then
LA_DEPTH := LA_DEPTH - 1
else
if SYSLIST(I)[1,1] = '('
then
TARGET := TARGET + 1
else
if SYSLIST(I) ≠ 'AND' and ≠ 'OR'
then
Note (8, 'STRC9704 SYNTAX ERROR-"" || SYSLIST(I) ||
ERROR_OCCURRED := true
I := LAST_INDEX + 1
/* Force break out of loop. */
else
I := I + 1
od /* (Termination: I is either incremented or set to LAST_INDEX +
1; LAST_INDEX is fixed; I would eventually exceed LAST_INDEX even
if other tests never occurred. */
if I > LAST_INDEX
then
(LCOPOR = 'OR' /* Operand was AND. */ and
~ ULTIMATE_FALLTHRU_CONDITION) OR
(LCOPOR = 'AND' /* Operand was OR. */ and
ULTIMATE_FALLTHRU_CONDITION ∧ is true)
then
LOCAL_BRANCH_LABEL := ULTIMATE_FALLTHRU_LABEL
FALLTHRU_LABEL_USED := true
else
LOCAL_BRANCH_LABEL := ULTIMATE_BRANCH_LABEL
else
LOCAL_BRANCH_LABEL := BLOCK_LABEL_PREFIX || UNIQUE_LABEL_ID || TARGET
LOCAL_LABEL_REQD(TARGET) := true
/* Note that we are relying on the automatic initialization of the
LOCAL_LABEL_REQD array to false at start of every invocation of
CONDITIONAL_EXPRESSION_PROCESSOR. */
else
COND_C-116
TERMINATE DO LOOP macro

Called by DD macro to terminate the current DO block loop by generating the necessary loop-terminating branches. If control can fall out of the bottom of the code at loop termination, set TD_FALLTHRU_OCCURS to true; else set false.

Bit LB_LABEL_REQ /* indicates whether label required on looping branch instruction. */

char LB_OPCODE, OPER1, OPER2 /* Opcode and operands of looping branch to be generated. */

CALL TRACE_PRINTER( ; 'TDL') /* Extract the following stored by DO macro: */

X := INFORMATION(CURRENT_NEST_LEVEL) /* INFORMATION of blocks involved in current DO loop. */

BRANCH8 := X[1,1] /* One character code for flow point which is target for flow point 8. */

BRANCH10 := X[2,1] /* One character code for flow point 10. */

BRANCH11 := X[3,1] /* One character code for flow point 11. */

LB_OPCODE := OPERAND3(CURRENT_NEST_LEVEL) /* Operands of looping branch to be generated. */

LB_LABEL_PREFIX := '$' BLOCK_NUMBER(CURRENT_NEST_LEVEL) /* Prefix for labels. */

if LB_LABEL_REQ then LABEL := LBLABEL_PREFIX 'LPB' fi /* Generate all looping instructions. */

docase BRANCH8 only
case ('W', 'U') generate (' B ' LB_LABEL_PREFIX ' ' BRANCH8 '1') esac
case ('B') generate (' B ' LB_LABEL_PREFIX ' ' '1') esac

case ('L') /* Nothing to generate; fall through to looping branch. */
enddocase

if LB_OPCODE = 'BCT' then generate (LABEL ' ' LB_OPCODE ' ' OPER1 ' ' OPER2 ' ' '1') fi

if BRANCH11 = 'N' then /* It must be 'W' or 'U'; looping branch fall-through is not to end of loop; generate branch to proper alternative conditional test. */
generate (LABEL ' ' '1') TD_FALLTHRU_OCCURS := false /* Branch defeats fall through. */
enddocase

/* Lemma: TERMINATE_DO_LOOP does not modify CURRENT_NEST_LEVEL */
"XHEX" Macro -- 19 December 1973

59001. MACRO XHEX ( : )
59002. /* Converts the integer in HEX_IN to a two-character hex string */
59003. HEX_OUT. */
59004. INT I, J /* Temporaries. */
59006. CALL Trace_Printer ( ; 'XHEX')
59007. /* Print macro name "XHEX" in anote if tracing on. */
59008. IF HEX(1) ≠ '0'
59009. THEN /* Hex array must be initialized. */
59100. HEX(1) := "O" HEX(2) := "1" HEX(3) := "2" HEX(4) := "3"
59101. HEX(5) := "4" HEX(6) := "5" HEX(7) := "6" HEX(8) := "7"
59102. HEX(9) := "8" HEX(10) := "9" HEX(11) := "A" HEX(12) := "B"
59103. HEX(13) := "C" HEX(14) := "D" HEX(15) := "E" HEX(16) := "F"
59104. END
59105. I := HEX_IN/16
59106. J := HEX_IN - I*16 + 1
59107. HEX_OUT := HEX(I+1) || HEX(J)
59108. END
AND OR OUTSTANDING
ANY COMP CASES
ANY ecology cases
AYEND
AYEND GENERATED
CASE
CASE_TAG
EOC AREA REQD
ERED
BLOCK
BLOCK LABEL PREFIX
BLOCK NAME
BLOCK NAME VALUE
BLOCK NUMBER
BLOCK TRACK COUNTERS
BLOCK TYPE
BLOCK TYPE VALUE
BRANCH LABEL
BRANCH TO CASE
BRANCH TO
BRANCH TO CASE
BRANCH
BRANCH CHANGED
B10
B11
B12
CASE
CASE ASSUMED VECTOR CASE
CASE ACT GEN
CASE COUNTRY
CASE COUNTRY
CASE FORMAT
CASE_GEN_COMPARE
CASE GET DOCASE INFO
CASE LABEL REQD
CASE_MISC_PROCESS
CASE OCCURS
CASE ECCTION CHECK
CASE PROCESS COMPARE OPERANDS
CASE PROCESS VECTOR OPERANDS
CASE SIT NAMES
CASE_TRACE COUNTER

**Parameter**

*Parameter*
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>11621.</td>
<td>*11128.</td>
<td>11139.</td>
<td>11140.</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>21036.</td>
<td>21183.</td>
<td>21175.</td>
<td>21197.</td>
<td>21231.</td>
</tr>
<tr>
<td>21191.</td>
<td>21194.</td>
<td>21198.</td>
<td>21251.</td>
<td>21259.</td>
<td>21288.</td>
</tr>
<tr>
<td>21246.</td>
<td>21299.</td>
<td>21250.</td>
<td>21320.</td>
<td>21333.</td>
<td>21336.</td>
</tr>
<tr>
<td>21313.</td>
<td>21316.</td>
<td>21319.</td>
<td>21320.</td>
<td>21333.</td>
<td>21336.</td>
</tr>
<tr>
<td>21342.</td>
<td>21345.</td>
<td>31139.</td>
<td>31029.</td>
<td>31012.</td>
<td>31014.</td>
</tr>
<tr>
<td>31320.</td>
<td>*31329.</td>
<td>31340.</td>
<td>31341.</td>
<td>31342.</td>
<td>31343.</td>
</tr>
<tr>
<td>31329.</td>
<td>33302.</td>
<td>33310.</td>
<td>33314.</td>
<td>33316.</td>
<td>33327.</td>
</tr>
<tr>
<td>33335.</td>
<td>33337.</td>
<td>33341.</td>
<td>33348.</td>
<td>33350.</td>
<td>33371.</td>
</tr>
<tr>
<td>33376.</td>
<td>33381.</td>
<td>33383.</td>
<td>33388.</td>
<td>33393.</td>
<td>33423.</td>
</tr>
<tr>
<td>33565.</td>
<td>33470.</td>
<td>33485.</td>
<td>33487.</td>
<td>33488.</td>
<td>33489.</td>
</tr>
<tr>
<td>37195.</td>
<td>37197.</td>
<td>37238.</td>
<td>*37259.</td>
<td>37260.</td>
<td>37266.</td>
</tr>
<tr>
<td>37264.</td>
<td>*37272.</td>
<td>37277.</td>
<td>37287.</td>
<td>37288.</td>
<td>37292.</td>
</tr>
<tr>
<td>45024.</td>
<td>43024.</td>
<td>63027.</td>
<td>63034.</td>
<td>63116.</td>
<td>63139.</td>
</tr>
<tr>
<td>81196.</td>
<td>81278.</td>
<td>81303.</td>
<td>*81306.</td>
<td>81308.</td>
<td>81422.</td>
</tr>
<tr>
<td>81652.</td>
<td>81655.</td>
<td>81660.</td>
<td>81666.</td>
<td>81668.</td>
<td>81669.</td>
</tr>
<tr>
<td>81692.</td>
<td>81692.</td>
<td>81896.</td>
<td>81995.</td>
<td>81998.</td>
<td>81999.</td>
</tr>
<tr>
<td>83359.</td>
<td>83360.</td>
<td>83365.</td>
<td>83366.</td>
<td>83367.</td>
<td>83371.</td>
</tr>
<tr>
<td>83374.</td>
<td>83379.</td>
<td>83391.</td>
<td>83397.</td>
<td>83398.</td>
<td>83400.</td>
</tr>
<tr>
<td>91012.</td>
<td>91015.</td>
<td>91015.</td>
<td>91016.</td>
<td>91018.</td>
<td>91019.</td>
</tr>
<tr>
<td>91023.</td>
<td>91035.</td>
<td>91037.</td>
<td>91042.</td>
<td>91049.</td>
<td>91051.</td>
</tr>
<tr>
<td>91053.</td>
<td>91055.</td>
<td>91062.</td>
<td>91064.</td>
<td>91065.</td>
<td>91084.</td>
</tr>
<tr>
<td>96017.</td>
<td>96050.</td>
<td>96055.</td>
<td>96061.</td>
<td>97025.</td>
<td>97135.</td>
</tr>
<tr>
<td>97136.</td>
<td>97138.</td>
<td>97138.</td>
<td>97148.</td>
<td>97151.</td>
<td>97156.</td>
</tr>
<tr>
<td>97189.</td>
<td>*97198.</td>
<td>97198.</td>
<td>97204.</td>
<td>97206.</td>
<td>97207.</td>
</tr>
<tr>
<td>97213.</td>
<td>97220.</td>
<td>97225.</td>
<td>97226.</td>
<td>97229.</td>
<td>97235.</td>
</tr>
<tr>
<td>97239.</td>
<td>*99004.</td>
<td>99015.</td>
<td>99016.</td>
<td>99017.</td>
<td>99017.</td>
</tr>
<tr>
<td>ID</td>
<td>*11001.</td>
<td>81362.</td>
<td>81369.</td>
<td>81396.</td>
<td>81403.</td>
</tr>
<tr>
<td>IF</td>
<td>11001.</td>
<td>11076.</td>
<td>11206.</td>
<td>11210.</td>
<td>11212.</td>
</tr>
<tr>
<td>IF_ASMCH_BRANCH</td>
<td>11001.</td>
<td>11002.</td>
<td>11003.</td>
<td>11004.</td>
<td>11005.</td>
</tr>
<tr>
<td>IF_BLOCK_COUNT</td>
<td>11057.</td>
<td>11163.</td>
<td>11206.</td>
<td>11207.</td>
<td>11208.</td>
</tr>
<tr>
<td>IF_CONDITIONAL_GENERATOR</td>
<td>11301.</td>
<td>11304.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IF_EXIT_LABEL</td>
<td>11305.</td>
<td>11306.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IF_EXIT_SPEC</td>
<td>11502.</td>
<td>11116.</td>
<td>11044.</td>
<td>11046.</td>
<td>11090.</td>
</tr>
<tr>
<td>INDEX</td>
<td>37238.</td>
<td>*37248.</td>
<td>37250.</td>
<td>37260.</td>
<td>37272.</td>
</tr>
<tr>
<td>INDEX_ADDR</td>
<td>31001.</td>
<td>31100.</td>
<td>31105.</td>
<td>31107.</td>
<td>31108.</td>
</tr>
<tr>
<td>INDEX_LENGTH</td>
<td>31344.</td>
<td>31166.</td>
<td>31170.</td>
<td>31176.</td>
<td>31180.</td>
</tr>
<tr>
<td>INDEX_ADDRESS</td>
<td>81193.</td>
<td>81194.</td>
<td>81195.</td>
<td>81196.</td>
<td>81197.</td>
</tr>
<tr>
<td>INDEX_RANGE_ADDR</td>
<td>33472.</td>
<td>*33473.</td>
<td>33474.</td>
<td>33475.</td>
<td>33476.</td>
</tr>
<tr>
<td>INDEX_REG</td>
<td>33561.</td>
<td>33562.</td>
<td>33563.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INDEX_TYPE</td>
<td>33564.</td>
<td>33565.</td>
<td>33566.</td>
<td>33567.</td>
<td>33568.</td>
</tr>
<tr>
<td>INFO</td>
<td>23017.</td>
<td>23034.</td>
<td>23035.</td>
<td>23043.</td>
<td>23045.</td>
</tr>
<tr>
<td>INFORMATION</td>
<td>23017.</td>
<td>23024.</td>
<td>23025.</td>
<td>23029.</td>
<td>23030.</td>
</tr>
<tr>
<td>INFORMATION_VALUE</td>
<td>33561.</td>
<td>33562.</td>
<td>33563.</td>
<td>33564.</td>
<td>33565.</td>
</tr>
<tr>
<td>INLINREFSAVEAREA</td>
<td>21019.</td>
<td>21210.</td>
<td>21212.</td>
<td>21222.</td>
<td>21224.</td>
</tr>
<tr>
<td>J</td>
<td>21408.</td>
<td>21409.</td>
<td>21410.</td>
<td>21411.</td>
<td>21412.</td>
</tr>
<tr>
<td>LA_DEPTH</td>
<td>21569.</td>
<td>21573.</td>
<td>21574.</td>
<td>21578.</td>
<td>21579.</td>
</tr>
<tr>
<td>LABEL</td>
<td>21711.</td>
<td>21714.</td>
<td>21717.</td>
<td>21760.</td>
<td>21763.</td>
</tr>
</tbody>
</table>

.Modified Parameter
<table>
<thead>
<tr>
<th>TARGET</th>
<th>(cont.)</th>
<th>03449</th>
<th>83450</th>
<th>977105</th>
<th>977205</th>
<th>977222</th>
<th>977222</th>
<th>977252</th>
<th>977253</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDL_FALLTHRU_OCCURS</td>
<td>a</td>
<td>146</td>
<td>25055</td>
<td>98045</td>
<td>98048</td>
<td>98051</td>
<td>25055</td>
<td>25058</td>
<td>27005</td>
</tr>
<tr>
<td>IDL_Genned</td>
<td>a23011</td>
<td>23044</td>
<td>23047</td>
<td>23051</td>
<td>23055</td>
<td>23058</td>
<td>23064</td>
<td>23068</td>
<td>23073</td>
</tr>
<tr>
<td>TERMINATE_GO_LOOP</td>
<td>a</td>
<td>23049</td>
<td>25054</td>
<td>27035</td>
<td></td>
<td>98001</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>THIS_CONDITIONAL_EQQ</td>
<td>a21052</td>
<td>21600</td>
<td>21634</td>
<td>21680</td>
<td>21684</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRACe_PREnTer</td>
<td>11020</td>
<td>13014</td>
<td>15005</td>
<td>21075</td>
<td>23019</td>
<td>25019</td>
<td>27012</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRACE VECTOR GennED</td>
<td>a153</td>
<td>81873</td>
<td>81894</td>
<td>81944</td>
<td>81950</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ULTIMATE_BRANCH_LABEL</td>
<td>a260</td>
<td>11706</td>
<td>11726</td>
<td>11730</td>
<td>11739</td>
<td>11754</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ULTIMATE_FALLTHRU_CONDITION</td>
<td>a263</td>
<td>11709</td>
<td>11746</td>
<td>11760</td>
<td>11762</td>
<td>11767</td>
<td>12659</td>
<td>12664</td>
<td>12689</td>
</tr>
<tr>
<td>ULTIMATE_FALLTHRU_LABEL</td>
<td>a266</td>
<td>12669</td>
<td>12687</td>
<td>12748</td>
<td>13500</td>
<td>97265</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNEXPECTED_OPERANDS_FOUND</td>
<td>a33034</td>
<td>33177</td>
<td>33226</td>
<td>33410</td>
<td>33451</td>
<td>33611</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNIQUE_LABEL_ID</td>
<td>a265</td>
<td>11717</td>
<td>11755</td>
<td>11765</td>
<td>11769</td>
<td>11772</td>
<td>11775</td>
<td>11777</td>
<td></td>
</tr>
<tr>
<td>UNTIL_COND_TEST</td>
<td>a21050</td>
<td>21319</td>
<td>21324</td>
<td>21328</td>
<td>21412</td>
<td>21415</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNTIL_END_INDEX</td>
<td>a21035</td>
<td>21231</td>
<td>21243</td>
<td>21246</td>
<td>21265</td>
<td>21268</td>
<td>21319</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNTIL_INDEX</td>
<td>a21035</td>
<td>21216</td>
<td>21280</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNTIL_PRESENT</td>
<td>a21045</td>
<td>21214</td>
<td>21217</td>
<td>21234</td>
<td>21238</td>
<td>21270</td>
<td>21272</td>
<td>21273</td>
<td>21275</td>
</tr>
<tr>
<td>USER_NAME</td>
<td>*11001</td>
<td>11022</td>
<td>11038</td>
<td>13001</td>
<td>13034</td>
<td>13054</td>
<td>13065</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WORK</td>
<td>*81001</td>
<td>81002</td>
<td>81003</td>
<td>81004</td>
<td>81006</td>
<td>81010</td>
<td></td>
<td></td>
<td>81804</td>
</tr>
<tr>
<td>WORKREG</td>
<td>a81041</td>
<td>81080</td>
<td>81124</td>
<td>81125</td>
<td>81131</td>
<td>81133</td>
<td>81134</td>
<td>81140</td>
<td>81631</td>
</tr>
<tr>
<td>WORKREG_USED</td>
<td>a81040</td>
<td>81073</td>
<td>81754</td>
<td>81766</td>
<td>81773</td>
<td>81776</td>
<td>81804</td>
<td>81805</td>
<td>81806</td>
</tr>
<tr>
<td>X</td>
<td>*81093</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XHEX</td>
<td>33582</td>
<td>81156</td>
<td>89001</td>
<td>99001</td>
<td>99002</td>
<td>99004</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| DEFINED | Modified | Parameter |
APPENDIX D

DIAGNOSTIC MESSAGES

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

<table>
<thead>
<tr>
<th>Severity</th>
<th>Message-Number</th>
<th>Message-Text</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>STRC1101</td>
<td>EXIT= IGNORED WITH ASYNCH</td>
<td>Both the EXIT= and ASYNCH operands were specified, but are mutually exclusive; the EXIT= operand has been ignored.</td>
</tr>
<tr>
<td>8</td>
<td>STRC1102</td>
<td>REL= OR MASK= NOT IN PARENTHESES—IGNORED</td>
<td>The REL= and MASK= operands must be part of a simple conditional and thus must be inside parentheses. The keyword has been ignored.</td>
</tr>
<tr>
<td>8</td>
<td>STRC1103</td>
<td>EXIT= IGNORED WITH ELSE=</td>
<td>The EXIT= and ELSE= operands were both specified but are mutually exclusive. The EXIT= operand has been ignored.</td>
</tr>
<tr>
<td>8</td>
<td>STRC1301</td>
<td>ELSE= name1 SPECIFIED ON IF BLOCK name2</td>
<td>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.</td>
</tr>
<tr>
<td>8</td>
<td>STRC1302</td>
<td>ELSE HAS ALREADY BEEN GENERATED FOR CURRENT IF</td>
<td>An ELSE macro has already occurred in the current IF block. The macro is ignored.</td>
</tr>
<tr>
<td>Severity</td>
<td>Message-Number</td>
<td>Message-Text</td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td>----------------</td>
<td>---------------------------------------------------</td>
<td></td>
</tr>
</tbody>
</table>
| 8        | STRC1501       | ELSE BLOCK  
elsename  NOT FOUND                               |
<p>|          |                | The operand ELSE=elsename was coded on the IF      |
|          |                | macro, but the FI has occurred before the ELSE     |
|          |                | occurred.                                          |
| 8        | STRC2101       | OPERANDS AFTER SECOND &quot;WHILE&quot; IGNORED             |
|          |                | The keyword &quot;WHILE&quot; appears more than once in the  |
|          |                | DO's operands.                                     |
| 8        | STRC2102       | OPERANDS AFTER SECOND &quot;UNTIL&quot; IGNORED             |
|          |                | The keyword &quot;UNTIL&quot; appears more than once in the  |
|          |                | DO's operands.                                     |
| 4        | STRC2103       | WARNING—&quot;WHILE,(BCT,...&quot; 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 &quot;OR&quot;. 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 &quot;DO&quot;—&quot;WHILE&quot;       |
|          |                | IGNORED                                          |
|          |                | Both the WHILE and UNTIL looping groups contain    |
|          |                | looping branches (BCTs, BXHs, or BXLE,); the       |
|          |                | WHILE looping group has been ignored.              |</p>
<table>
<thead>
<tr>
<th>Severity</th>
<th>Message-Number</th>
<th>Message-Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>STRC2106</td>
<td>INVALID NUMBER OF OPERANDS FOR opcode</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The looping branch <code>opcode</code> has the wrong number of operands; BCT should have one, BXLE or BXH should have two.</td>
</tr>
<tr>
<td>8</td>
<td>STRC2107</td>
<td>xxx INVALID AFTER LOOPS BRANCH—&quot;AND&quot; INSERTED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The operand following the looping branch must be &quot;AND&quot; or &quot;OR&quot;; xxx was found.</td>
</tr>
<tr>
<td>8</td>
<td>STRC2108</td>
<td>FIRST OPERAND MUST BE &quot;WHILE&quot;, &quot;UNTIL&quot;, &quot;FOREVER&quot;, OR OMITTED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The first operand of the DO macro is invalid. Either WHILE or UNTIL has been inserted, depending on the remaining operands.</td>
</tr>
<tr>
<td>8</td>
<td>STRC2109</td>
<td>WHILE TEST IS VOID—IGNORED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No looping group follows the keyword &quot;WHILE&quot;; the keyword has been discarded.</td>
</tr>
<tr>
<td>8</td>
<td>STRC2110</td>
<td>LOGIC OPERATOR BETWEEN &quot;WHILE&quot; AND &quot;UNTIL&quot; OMITTED—&quot;AND&quot; ASSUMED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No logic operator occurs between the WHILE and UNTIL looping groups. An &quot;AND&quot; has been inserted.</td>
</tr>
<tr>
<td>8</td>
<td>STRC2111</td>
<td>UNTIL TEST IS VOID—IGNORED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No looping group follows the keyword &quot;UNTIL&quot;; the keyword has been discarded.</td>
</tr>
<tr>
<td>8</td>
<td>STRC2112</td>
<td>PARENTHESES OMITTED AROUND <code>opcode</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>The looping branch <code>opcode</code> was not specified as a sublist.</td>
</tr>
<tr>
<td>Severity</td>
<td>Message-Number</td>
<td>Message-Text</td>
</tr>
<tr>
<td>----------</td>
<td>----------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>8</td>
<td>STRC2113</td>
<td>REL= OR MASK= NOT IN PARENTHESES—IGNORED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The REL= and MASK= operands must be a part of a simple conditional and thus must be inside parentheses. The keyword has been ignored.</td>
</tr>
<tr>
<td>8</td>
<td>STRC2114</td>
<td>SUPERFLUOUS LOOPING GROUP IGNORED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Both WHILE and UNTIL are present, but other operands precede both. Such operands have been ignored.</td>
</tr>
<tr>
<td>8</td>
<td>STRC2301</td>
<td>MORE THAN ONE &quot;ATEND&quot; IN BLOCK</td>
</tr>
<tr>
<td></td>
<td></td>
<td>More than one ATEND macro has been found for the same DO. Only the first is processed.</td>
</tr>
<tr>
<td>8</td>
<td>STRC2501</td>
<td>MORE THAN ONE &quot;ONEXIT&quot; IN BLOCK</td>
</tr>
<tr>
<td></td>
<td></td>
<td>More than one ONEXIT macro has been found for the same DO. Only the first is processed.</td>
</tr>
<tr>
<td>8</td>
<td>STRC2502</td>
<td>NO EXIT FOR THIS &quot;DO&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No EXIT has occurred specifying the DO block for which an ONEXIT is being generated. The segment is dead code.</td>
</tr>
<tr>
<td>4</td>
<td>STRC3101</td>
<td>WARNING—xxx ASSUMED AS INDEX: USE &quot;DOCASE,xxx&quot; FOR RANGE SPEC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>xxx is either &quot;IFANY&quot; or &quot;ONLY&quot; 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.</td>
</tr>
<tr>
<td>8</td>
<td>STRC3102</td>
<td>xxx INVALID SECOND OPERAND—IGNORED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The second operand of DOCASE may only be &quot;SPARSE&quot;, &quot;SIMPLE&quot;, &quot;IFANY&quot;, &quot;ONLY&quot;, or omitted. xxx was found.</td>
</tr>
<tr>
<td>Severity</td>
<td>Message-Number</td>
<td>Message-Text</td>
</tr>
<tr>
<td>----------</td>
<td>----------------</td>
<td>---------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>8</td>
<td>STRC3103</td>
<td>xxx INVALID THIRD OPERAND—IGNORED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The third operand of DOCASE may only be &quot;IFANY&quot;, &quot;ONLY&quot;, or omitted. xxx was found.</td>
</tr>
<tr>
<td>12</td>
<td>STRC3104</td>
<td>GENERAL/SPARSE/CHARCOMP DOCASE NESTING LEVEL nestlev EXCEEDS MAXIMUM OF maxlev—MACROS MUST BE MODIFIED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>8</td>
<td>STRC3301</td>
<td>&quot;CASE&quot; NOT IMMEDIATE DAUGHTER OF &quot;DOCASE&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>8</td>
<td>STRC3302</td>
<td>ASSUMING &quot;BLEND&quot; OMITTED—INSERTED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Preceded by message 3301. Since the second containing block is a DOCASE, one BLEND is inserted to get to it.</td>
</tr>
<tr>
<td>8</td>
<td>STRC3303</td>
<td>ASSUMING TWO &quot;BLENDs&quot; OMITTED—INSERTED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Preceded by message 3301. Since the third containing block is a DOCASE, two BLENDs are inserted to get to it.</td>
</tr>
<tr>
<td>8</td>
<td>STRC3304</td>
<td>&quot;CASE&quot; TREATED AS &quot;BLOCK&quot; MACRO</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Preceded by message 3301. No fix-up was possible. The CASE is converted to a BLOCK.</td>
</tr>
<tr>
<td>Severity</td>
<td>Message-Number</td>
<td>Message-Text</td>
</tr>
<tr>
<td>----------</td>
<td>---------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>8</td>
<td>STRC3305</td>
<td>xxx INVALID—yyy ASSUMED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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 &quot;(15, 10)&quot;). The operand yyy replaces xxx.</td>
</tr>
<tr>
<td>4</td>
<td>STRC3306</td>
<td>EARLIER UNEXPECTED OPERAND IMPLIES THIS TO BE CASE xxx</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>8</td>
<td>STRC3307</td>
<td>OPERAND INVALID VALUE ON SIMPLE CASE xxx</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>8</td>
<td>STRC3308</td>
<td>&quot;DOCASE...,ONLY&quot; INVALID WITH MISC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A CASE MISC has been found in a DOCASE with the ONLY range option. Since these are mutually exclusive, the ONLY has been ignored.</td>
</tr>
<tr>
<td>8</td>
<td>STRC3309</td>
<td>OPERAND MUST BE SELF-DEFINING TERM OR OMITTED ON SIMPLE CASE xxx</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>8</td>
<td>STRC3310</td>
<td>REL= OR MASK= NOT IN PARENTHESES—IGNORED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The REL= and MASK= operands must be a part of a simple conditional and thus must be inside parentheses. The keyword has been ignored.</td>
</tr>
<tr>
<td>Severity</td>
<td>Message-Number</td>
<td>Message-Text</td>
</tr>
<tr>
<td>----------</td>
<td>----------------</td>
<td>--------------</td>
</tr>
<tr>
<td>8</td>
<td>STRC3311</td>
<td>MULTIPLE MISC CASES IN THIS DOCASE—THIS BLOCK IS DEAD CODE</td>
</tr>
<tr>
<td>8</td>
<td>STRC3312</td>
<td>xxx INVALID—ONLY FIRST TWO SUBOPERANDS PROCESSED</td>
</tr>
<tr>
<td>*</td>
<td>STRC3313</td>
<td>CASE DEBUG ID=X'hh'</td>
</tr>
<tr>
<td>8</td>
<td>STRC3701</td>
<td>DOCASE CONTAINS NO VALID CASES</td>
</tr>
<tr>
<td>8</td>
<td>STRC4301</td>
<td>NO BLOCKS ACTIVE—&quot;BLEND&quot; IGNORED</td>
</tr>
<tr>
<td>8</td>
<td>STRC4302</td>
<td>NO BLOCK ACTIVE NAMED xxx—&quot;BLEND&quot; IGNORED</td>
</tr>
<tr>
<td>8</td>
<td>STRC5301</td>
<td>BLEND OF biname ASSUMED</td>
</tr>
<tr>
<td>Severity</td>
<td>Message-Number</td>
<td>Message-Text</td>
</tr>
<tr>
<td>----------</td>
<td>----------------</td>
<td>---------------------------------------------------</td>
</tr>
<tr>
<td>8</td>
<td>STRC8101</td>
<td>LINKAGE=*** INVALID—&quot;OS&quot; ASSUMED</td>
</tr>
<tr>
<td>8</td>
<td>STRC8102</td>
<td>SECOND LINKAGE OPERAND IGNORED</td>
</tr>
<tr>
<td>4</td>
<td>STRC8103</td>
<td>WARNING—SAVETRACE REQUIRES &quot;FINAL&quot; MACRO</td>
</tr>
<tr>
<td>8</td>
<td>STRC8104</td>
<td>DEBUG=*** INVALID—IGNORED</td>
</tr>
<tr>
<td>8</td>
<td>STRC8105</td>
<td>SAVETRACE MUST BE SPECIFIED ON FIRST PROC</td>
</tr>
<tr>
<td>8</td>
<td>STRC8106</td>
<td>SAVETRACE REQUIRES FIRST PROC TO BE LINKAGE=OS</td>
</tr>
<tr>
<td>4</td>
<td>STRC8107</td>
<td>REG 1 MUST BE AMONG THOSE SAVED</td>
</tr>
<tr>
<td>Severity</td>
<td>Message-Number</td>
<td>Message-Text</td>
</tr>
<tr>
<td>----------</td>
<td>----------------</td>
<td>--------------</td>
</tr>
<tr>
<td>4</td>
<td>STRC8107</td>
<td>but the restore has been issued. If register 1 was not among those saved, its value will be undefined. (cont.)</td>
</tr>
<tr>
<td></td>
<td>STRC8108</td>
<td>PROC <code>proc-name</code>, DEBUG ID=X'hh' The save-trace, proc-trace, or proc-count debug option has been specified for this proc. The hex id byte <code>hh</code> 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 &quot;PRINT NOGEN&quot; has been specified.</td>
</tr>
<tr>
<td>8</td>
<td>STRC8109</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>STRC8301</td>
<td>NO REGISTERS SAVED—RESTORE IGNORED RESTORE= was coded on the CORP macro, but SAVE=NONE was coded on the PROC. The operand is ignored.</td>
</tr>
<tr>
<td>4</td>
<td>STRC8302</td>
<td>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.</td>
</tr>
<tr>
<td>0</td>
<td>STRC8303</td>
<td>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.</td>
</tr>
<tr>
<td>Severity</td>
<td>Message-Number</td>
<td>Message-Text</td>
</tr>
<tr>
<td>----------</td>
<td>----------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>8</td>
<td>STRC9103</td>
<td>EXIT TO IMMEDIATELY SURROUNDING BLOCK INVALID</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The block name specified as the EXIT target is the block immediately surrounding the EXIT.</td>
</tr>
<tr>
<td>8</td>
<td>STRC9104</td>
<td>EXIT TO DO BLOCK INVALID WITHIN ATEND OR ONEXIT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>0</td>
<td>STRC9201</td>
<td>ONE OR MORE EXIT'S REFERENCE THIS POINT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>12</td>
<td>STRC9301</td>
<td>BLOCK NESTING LIMIT OF limit EXCEEDED—MACROS MUST BE MODIFIED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The current static nesting level has exceeded the stack limit in the macros of limit. Either the program or macros must be modified.</td>
</tr>
<tr>
<td>8</td>
<td>STRC9302</td>
<td>NON-CASE BLOCK IMMEDIATELY SURROUNDED BY DOCASE INVALID</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The block being defined is not a CASE block but is immediately surrounded by a DOCASE block. The result is undefined.</td>
</tr>
<tr>
<td>8</td>
<td>STRC9401</td>
<td>INSUFFICIENT OPERANDS FOR TEST &quot;opcode&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The parenthesized list which is supposed to be a simple conditional contains two items, the operation code opcode and a bc-spec.</td>
</tr>
</tbody>
</table>
Severity   Message-Number   Message-Text

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.
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 (">", "/") 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 (">", "+").
<table>
<thead>
<tr>
<th>Severity</th>
<th>Message-Number</th>
<th>Message-Text</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>STRC9706</td>
<td>SYNTAX ERROR—LOOKING FOR SIMPLE CONDITIONAL, FOUND &quot;xxx&quot;</td>
<td>The operand xxx was found in a conditional expression where a simple conditional sublist was expected.</td>
</tr>
<tr>
<td></td>
<td>STRC9902</td>
<td>START OF BLOCK nn (blname) AT DEPTH level</td>
<td>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 BLKnn. The current static nest level is level.</td>
</tr>
<tr>
<td></td>
<td>STRC9903</td>
<td>END OF BLOCK nn (blname) AT DEPTH level</td>
<td>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 BLKnn. The current static nest level is level.</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Data Set</th>
<th>BLKSIZE</th>
<th>LRECL</th>
<th>RECFM</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) PROSE</td>
<td>2000</td>
<td>80</td>
<td>FB</td>
</tr>
<tr>
<td>(2) STRCMACS</td>
<td>2000</td>
<td>80</td>
<td>FB</td>
</tr>
<tr>
<td>(3) LISTINGS</td>
<td>2000</td>
<td>137</td>
<td>VBA</td>
</tr>
<tr>
<td>(4) RUFDRAFT</td>
<td>2000</td>
<td>80</td>
<td>FB</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Term</th>
<th>Page</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>abnormal block exit</td>
<td>27</td>
<td>CORP</td>
</tr>
<tr>
<td>abstract programming language</td>
<td>8</td>
<td>CSECT</td>
</tr>
<tr>
<td>addressability</td>
<td>42</td>
<td>current block</td>
</tr>
<tr>
<td>angle brackets</td>
<td>19, A-2</td>
<td>current nest level</td>
</tr>
<tr>
<td>asynchronous IF block</td>
<td>20</td>
<td>debug counts</td>
</tr>
<tr>
<td>ATEND</td>
<td>29, A-3</td>
<td>debugging aids</td>
</tr>
<tr>
<td>ATEXIT</td>
<td>30, A-4</td>
<td>decision making</td>
</tr>
<tr>
<td>base register</td>
<td>34, A-32</td>
<td>defining modules</td>
</tr>
<tr>
<td>bc-spec</td>
<td>A-1</td>
<td>diagnostic messages</td>
</tr>
<tr>
<td>BCT</td>
<td>21</td>
<td>distribution medium</td>
</tr>
<tr>
<td>BLEND</td>
<td>16, 19, 23, 27, 38, A-5</td>
<td>STRCMACS</td>
</tr>
<tr>
<td>BLISS</td>
<td>6, 7, 10</td>
<td>DO</td>
</tr>
<tr>
<td>BLOCK</td>
<td>16, A-6</td>
<td>DO decision table</td>
</tr>
<tr>
<td>block counts</td>
<td>40</td>
<td>DO operand format</td>
</tr>
<tr>
<td>block exit</td>
<td>27</td>
<td>DOCASE</td>
</tr>
<tr>
<td>blocks</td>
<td>16</td>
<td>character compare</td>
</tr>
<tr>
<td>BXH</td>
<td>21</td>
<td>type</td>
</tr>
<tr>
<td>BXLE</td>
<td>21</td>
<td>conditional test type</td>
</tr>
<tr>
<td>CASE</td>
<td>7, 23, A-7</td>
<td>simple type</td>
</tr>
<tr>
<td>case ranges</td>
<td>24</td>
<td>sparse type</td>
</tr>
<tr>
<td>CASEEND</td>
<td>27, A-9</td>
<td>DOCASEND</td>
</tr>
<tr>
<td>COBOL</td>
<td>10</td>
<td>DOEND</td>
</tr>
<tr>
<td>conditional expression</td>
<td>19, 21, 26, A-1</td>
<td>DROP</td>
</tr>
<tr>
<td>conditional relations</td>
<td>19</td>
<td>efficiency</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ELSE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>error messages</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ESAC</td>
</tr>
</tbody>
</table>

\( \mathcal{E} - 3 \)
INDEX (continued)

<table>
<thead>
<tr>
<th>Term</th>
<th>Page</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESACOD</td>
<td>23, A-20</td>
<td>looping branches</td>
</tr>
<tr>
<td>evoking procs</td>
<td>36-38</td>
<td>looping group</td>
</tr>
<tr>
<td>EXIT</td>
<td>27, A-21</td>
<td>messages</td>
</tr>
<tr>
<td>exit severity</td>
<td>38</td>
<td>MISC</td>
</tr>
<tr>
<td>FI</td>
<td>18, A-22</td>
<td>miscellaneous case</td>
</tr>
<tr>
<td>FINAL</td>
<td>38, A-23</td>
<td>modifying structured programs</td>
</tr>
<tr>
<td>FOREVER</td>
<td>28, A-12</td>
<td>modularity</td>
</tr>
<tr>
<td>FORTRAN</td>
<td>10</td>
<td>modules</td>
</tr>
<tr>
<td>GEDANKEN</td>
<td>10</td>
<td>multiple decisions</td>
</tr>
<tr>
<td>goto controversy</td>
<td>3</td>
<td>nest level</td>
</tr>
<tr>
<td>goto-less programming</td>
<td>1,2-10</td>
<td>nesting blocks</td>
</tr>
<tr>
<td>IF</td>
<td>18, 28, A-24</td>
<td>OD</td>
</tr>
<tr>
<td>IFANY</td>
<td>27, A-15</td>
<td>ONEND</td>
</tr>
<tr>
<td>IFEND</td>
<td>19, A-26</td>
<td>ONEXIT</td>
</tr>
<tr>
<td>immediately</td>
<td>17</td>
<td>ONLY</td>
</tr>
<tr>
<td>surrounding</td>
<td>17</td>
<td>operand format for DO</td>
</tr>
<tr>
<td>in-line identifier</td>
<td>34, A-31</td>
<td>OREGANO</td>
</tr>
<tr>
<td>infinite loop</td>
<td>28</td>
<td>PL/I</td>
</tr>
<tr>
<td>installing STRCMACS</td>
<td>E-1</td>
<td>PROC</td>
</tr>
<tr>
<td>ISWIM</td>
<td>10</td>
<td>OS vs non-OS</td>
</tr>
<tr>
<td>iteration</td>
<td>21</td>
<td>procs</td>
</tr>
<tr>
<td>labels</td>
<td>42</td>
<td>PROCEND</td>
</tr>
<tr>
<td>link register</td>
<td>36, A-11</td>
<td>properly nested</td>
</tr>
<tr>
<td>linkage of procs</td>
<td>36-38</td>
<td>proving program correctness</td>
</tr>
<tr>
<td>LISP</td>
<td>10</td>
<td>quasi-goto-less code</td>
</tr>
<tr>
<td>Term</td>
<td>Page</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-------</td>
<td>--------------</td>
</tr>
<tr>
<td>range option</td>
<td>27</td>
<td>SIMPL-M (con't)</td>
</tr>
<tr>
<td>recursive code</td>
<td>34, 42</td>
<td>generate</td>
</tr>
<tr>
<td>reentrant code</td>
<td>34, 42</td>
<td>implied relations</td>
</tr>
<tr>
<td>register save areas</td>
<td>32, A-31</td>
<td>initialization</td>
</tr>
<tr>
<td>user supplied</td>
<td>33</td>
<td>B-2</td>
</tr>
<tr>
<td>dynamic</td>
<td>33</td>
<td>B-4</td>
</tr>
<tr>
<td>length of</td>
<td>33</td>
<td>B-1</td>
</tr>
<tr>
<td>CORPVALUES</td>
<td>41</td>
<td>B-4</td>
</tr>
<tr>
<td>BACK</td>
<td>41</td>
<td>B-4</td>
</tr>
<tr>
<td>restoring registers</td>
<td>32, A-10</td>
<td>B-1</td>
</tr>
<tr>
<td>return address register</td>
<td>36, A-10</td>
<td>B-2</td>
</tr>
<tr>
<td>return code</td>
<td>35, A-10</td>
<td>B-1</td>
</tr>
<tr>
<td>returning values</td>
<td>32, A-10</td>
<td>B-2</td>
</tr>
<tr>
<td>RUFDRAFT</td>
<td>E-1</td>
<td>SIMPL-X</td>
</tr>
<tr>
<td>save areas</td>
<td>32, A-31</td>
<td>simple conditionals</td>
</tr>
<tr>
<td>user supplied</td>
<td>33</td>
<td>18</td>
</tr>
<tr>
<td>dynamic</td>
<td>33</td>
<td>1</td>
</tr>
<tr>
<td>length of</td>
<td>33</td>
<td>11-13</td>
</tr>
<tr>
<td>CORPVALUES</td>
<td>41</td>
<td>10, B-1</td>
</tr>
<tr>
<td>BACK</td>
<td>41</td>
<td>18</td>
</tr>
<tr>
<td>SIMPL-M</td>
<td>8, B-1</td>
<td>stepwise refinement</td>
</tr>
<tr>
<td>arrays</td>
<td>B-3</td>
<td>1,</td>
</tr>
<tr>
<td>assignments</td>
<td>B-3</td>
<td>11-13</td>
</tr>
<tr>
<td>comments</td>
<td>B-4</td>
<td>18</td>
</tr>
<tr>
<td>data types</td>
<td>B-2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>21</td>
</tr>
<tr>
<td></td>
<td></td>
<td>35</td>
</tr>
<tr>
<td></td>
<td></td>
<td>21</td>
</tr>
<tr>
<td></td>
<td></td>
<td>35, A-33</td>
</tr>
</tbody>
</table>