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
NOTICE

THIS DOCUMENT HAS BEEN REPRODUCED FROM THE BEST COPY FURNISHED US BY THE SPONSORING AGENCY. ALTHOUGH IT IS RECOGNIZED 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>Topic</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>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>
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. 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 \textit{if} and \textit{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 conventional programming.

GOTO-LESS PROGRAMMING

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

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

\[
\text{if } I < 5 \\
\text{then} \\
J := 5; \\
\text{else} \\
J := I;
\]
he would use:

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

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

At the 1972 National Conference of the ACM, a debate was held on whether the goto should even be a part of future programming languages. The interesting thing about the debate was that even those who were trying to justify the retention of the goto did not do so on the grounds that it was required for good programming. In fact, one debater stated: "In my opinion, there have been far too many gotos in most programs .... The no goto rule....does improve the code produced by most programmers.... If I were teaching a beginning programming class, I would not teach the goto." 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
goto-less programming

IF (q) GO TO 20
IF (r) GO TO 10
\(\text{IF (r) GO TO 10}\)
\(<\text{A}>\)
GO TO 60
\(\text{GO TO 60}\)

10  \(\text{GO TO 60}\)

20 IF (s) GO TO 80
\(\text{IF (s) GO TO 80}\)
\(<\text{C}>\)
\(<\text{D}>\)
\(<\text{E}>\)

30 IF (.NOT.(t)) GO TO 40
\(\text{IF (.NOT.(t)) GO TO 40}\)
\(<\text{F}>\)
GO TO 30
\(\text{GO TO 30}\)

40 IF (u) GO TO 70
\(\text{IF (u) GO TO 70}\)
\(<\text{G}>\)

50 IF (.NOT.(v)) GO TO 60
\(\text{IF (.NOT.(v)) GO TO 60}\)
\(<\text{H}>\)

60 RETURN
\(\text{RETURN}\)

70 \(<\text{J}>\)
GO TO 50
\(\text{GO TO 50}\)

80 IF (w) GO TO 90
\(\text{IF (w) GO TO 90}\)
\(<\text{C}>\)
\(<\text{D}>\)
GO TO 110
\(\text{GO TO 110}\)

90 \(<\text{C}>\)
\(<\text{D}>\)
\(<\text{K}>\)

100 IF (.NOT.(x)) GO TO 110
\(\text{IF (.NOT.(x)) GO TO 110}\)
\(<\text{L}>\)
GO TO 100
\(\text{GO TO 100}\)

110 IF (.NOT.(w)) GO TO 120
\(\text{IF (.NOT.(w)) GO TO 120}\)
\(<\text{M}>\)

120 IF (.NOT.(v)) GO TO 60
\(\text{IF (.NOT.(v)) GO TO 60}\)
\(<\text{H}>\)
GO TO 60
\(\text{GO TO 60}\)

IF q
\(\text{IF q}\)
THEN
\(\text{THEN}\)
\(<\text{C}>\)
\(<\text{D}>\)

IF s
\(\text{IF s}\)
THEN
\(\text{THEN}\)
\(<\text{B}>\)
\(<\text{D}>\)

IF w
\(\text{IF w}\)
THEN
\(\text{THEN}\)
\(<\text{K}>\)
\(<\text{B}>\)

WHILE x
\(\text{WHILE x}\)
DO
\(\text{DO}\)
\(<\text{L}>\)

DO
\(\text{DO}\)
\(<\text{M}>\)

FI
\(\text{FI}\)

ELSE
\(\text{ELSE}\)
\(<\text{E}>\)
\(<\text{G}>\)

WHILE t
\(\text{WHILE t}\)
DO
\(\text{DO}\)
\(<\text{F}>\)

DO
\(\text{DO}\)
\(<\text{M}>\)

FI
\(\text{FI}\)

IF v
\(\text{IF v}\)
THEN
\(\text{THEN}\)
\(<\text{H}>\)
\(\text{RETURN}\)

ELSE
\(\text{ELSE}\)
\(<\text{K}>\)
\(<\text{L}>\)

\(\text{RETURN}\)

FI
\(\text{FI}\)

FI
\(\text{FI}\)

\(<\text{I}>\)
in goto-less form. Alan Perlis has referred to similar situations as the 'Turing Tarpit' in which everything is possible, but nothing is easy."¹

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

The main arguments for goto-less programming are:

- goto-less programs are easier to understand, debug, and modify.

- It is easier to prove assertions (in particular, to prove program correctness) about goto-less programs.

- Goto-less programs are less likely to contain bugs due to their intellectual manageability.

- Compilers are able to understand, and therefore to optimize, goto-less programs to a larger extent.

- Languages which contain the goto construct invite its misuse to make a "rat's nest" of control flow.

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

²ibid, 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 if's. The program:

```
OUTER: begin;  
  INNER: begin;  
    [The BLISS language has been simplified somewhat.]  
    if I = 0  
      then  
        leave OUTER;  
        \alpha  
      end;  
    end;  
  \beta  
end;  
\gamma
```
has the same effect as:

```
OUTER: begin;

INNER: begin;
    if I \neq 0 then
        begin;
            \alpha
        end;
    if I \neq 0 then
        begin;
            \beta
        end;
end;
```

Notice, however, that by using the \texttt{leave} statement, the immediate predecessors of $\gamma$ are not quite as obvious; as a result, the compiler should give appropriate warning messages to flag the targets of \texttt{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. 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.

1Hopkins, 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" \( B_1 \) for some abstract super-machine with arbitrarily complex instructions. Most of the instructions of \( B_1 \) are not intelligible to our real computer. But the number of instructions are few (maybe 50 or so), so we can feel that, if there were a machine which could understand \( B_1 \), it would surely accomplish the task A. We can now take the instructions of \( B_1 \) (call them the \( A_{2,i} \)) and for whichever are not understandable to our real computer repeat the problem-solving process producing a program \( B_{2,i} \) —the "how" for each \( A_{2,i} \) in more primitive terms. This process is continued until eventually all instructions are in terms intelligible to our computer.

At this point, we have the program B written entirely in some machine-understandable language and all the intermediate "super-instructions" may be discarded. However, for the purpose of documentation and maintenance, it is probably desirable to save these intermediate programs. This may be accomplished in the following ways. (1) The name of the super-instruction can appear as comment cards surrounding the final instructions defining the super-instruction. (2) The super-instruction can be replaced by a \texttt{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

---

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 manageability of structured programs may lead to the use of better algorithms. The ease of modifiability opens the door for replacing entire algorithms within working programs with a minimum of problems. Techniques such as these can make major increases in program efficiency and can make up for a myriad of redundant tests or occurrences of branch-to-a-branch.

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

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

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

Defining blocks

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

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

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

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

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

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

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

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

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

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

The structured macros are used to define properly nested programs. Error messages occur if a block terminating macro is issued for other than the current block. If no block name is coded as the operand of a block terminating
macro, the current block is assumed. Blocks may be nested up to some depth which is built into the macros. As distributed, this depth is 100.

Decision making

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

\[(\text{LTR},3,3,Z)\]

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:

\[
\text{TRY IF (LTR, 3, 3, Z)} \\
\text{L 1,WORD} \\
\text{LA 1,1(1)} \\
\text{ST 1,WORD} \\
\text{FI TRY}
\]

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

\[(\text{LTR},3,3,Z)\]
\[(\text{LTR},3,3,\text{REL}=Z)\]
\[(\text{LTR},3,3,\text{MASK}=8)\]  \hspace{1cm} \text{[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:

\[
\text{LTR 3,3} \\
\text{BNZ end-of-block or LTR 3,3} \\
\text{BC 7, end-of-block}
\]

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

\[
\text{IF (TS, SPOT, MASK=8)} \\
\text{IF (CR, 3, 4, E)} \\
\text{IF (CLM, 3, X'C', BYTE, REL=E)}
\]
The following relations may be used:

- H or GT (High)
- L or LT (Low)
- E or EQ (Equal)
- O (Ones or Overflow)
- P (Plus)
- M (Minus)
- Z (Zero or Zeros)

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

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

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

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

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

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

```
QTEST IF (LTR,5,5,Z), OR, (CH,3,HWORD,NE)
    L 7,SPOT
MORECHK IF <,(CR,7,5,E), OR, (SR,3,1,Z),>, AND, (LTR,1,1,
    MASK=SYMMASK)
    L 1,WORD
    LA 1,1(l)
    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', II), 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
    FI
  OI FLAG, EOF
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.

\[
\text{LA 1,1} \quad \text{FILL ARRAY} \\
\text{LA 3,ARRAY} \quad \text{WITH 1's.} \\
\text{LA 4,4} \\
\text{LA 5,ARRAYEND} \\
\text{FILL1S DO UNTIL,(BXLE,3,4)} \\
\text{ST 1,0(3)} \\
\text{OD}
\]

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.

\[
\text{LA 3},5 \\
\text{DO WHILE,(BCT,3)} \\
\quad \cdot \\
\quad \cdot \\
\text{OD}
\]

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).

\[
\text{X DO UNTIL,(BCT,5),OR,(LTR,4,4,Z),AND,(TM,FLAG,X'80',Z)} \\
\text{OD}
\]

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:

\[
\text{X B } \alpha \\
\gamma \text{ LTR 4,4} \\
\text{BNZ } \alpha \\
\text{TM FLAG,X'80'} \\
\text{BZ } \beta \\
\alpha \text{ DS 0H} \\
\text{block code} \\
\text{BCT 5,}\gamma \\
\beta \text{ DS 0H}
\]

22
Appendix B shows the code generated for all possible combinations of DO operands.

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

Multiple decisions

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

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

Example:

```
UPDATE DOCASE REQWORD
ADD CASE
\_\_\_\_
ESAC ADD
REPL CASE
\_\_\_\_
ESAC
CHANGE CASE
\_\_\_\_
ESAC CASE
\_\_\_\_
ESAC CASE
\_\_\_\_
ESAC
ESACOD
```

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

One of the CASE macros (usually the last of the list) may have the operand "MISC" to indicate that it is to be executed only if no other block is appropriate.
(that is, if the index is less than one or greater than $n$, in the form we have discussed so far). This miscellaneous block is not counted in locating the $i$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 I
A CASE 3,7
     .
     .
     ESAC
B CASE 0,2,8
     .
     .
     ESAC
C CASE 4,(9,13),X'1C'
     .
     .
     ESAC
D CASE FIVE,(FOURTEEN,SIXTEEN)
     .
     .
     ESAC
E CASE
     .
     .
     ESAC
ESACEND
```
Case A will be executed if I contains either 3 or 7; case C for I of 4, 9, 10, 11, 12, 13, or 28 (=X'1C'). As indicated, values may be specified symbolically (although slower code is generated). All values must be in the range 0-4095. (Again, slower code is generated for values greater than 255.)

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

```
DOCASE I  
   or Fullword

DOCASE (I,W)  
   or Halfword

DOCASE (I,B)  
   or One byte

DOCASE (3)  
   or 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  ($, 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〉
SCAN DO WHILE,(TM,FLAG,ENDOCARD,Z)  
:  
:  
WHOOPS IF (CLI,DELIMITR,C',',NE)  
〈Print 'BAD DELIMITER' message〉
EXIT  
FI  
:  
:  
OD  
OD  DOINFILE
In this code segment, a delimiter other than the comma will cause abnormal termination of the control card scan loop after printing a message. Since the EXIT macro has no operand, the exit is to the end of the block containing the block containing the EXIT (the surrounding block whose nest level is one less than the current nest level; in our example, the block SCAN). Any surrounding block's name may be specified as the EXIT's operand to cause transfer to the end of that block. In our example, adding the operand DOINFILE to the EXIT would skip the rest of the input when the error occurred.

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

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

or

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

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

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

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

```
DO FOREVER
```

or just

```
DO
```

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

<table>
<thead>
<tr>
<th>UPDATE BLOCK</th>
</tr>
</thead>
<tbody>
<tr>
<td>LA 1,1</td>
</tr>
<tr>
<td>L 2,IDTAB</td>
</tr>
<tr>
<td>SEARCH DO</td>
</tr>
<tr>
<td>WHILE (CLI, 0(2), X'00', NE)</td>
</tr>
<tr>
<td>IF (CLC, ARG(8), 0(2), EQ)</td>
</tr>
<tr>
<td>A 1,8(2)</td>
</tr>
<tr>
<td>ST 1,8(2)</td>
</tr>
<tr>
<td>EXIT UPDATE</td>
</tr>
<tr>
<td>FI</td>
</tr>
<tr>
<td>L 2,12(2)</td>
</tr>
<tr>
<td>OD</td>
</tr>
<tr>
<td>MVC 0(8,2), ARG</td>
</tr>
<tr>
<td>ST 1,8(2)</td>
</tr>
<tr>
<td>BLEND UPDATE</td>
</tr>
</tbody>
</table>

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.

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

\texttt{SAVE=3}

or

\texttt{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:

\begin{verbatim}
PROC SAVE=(R5, LAST)
.
.
R5 EQU 5
LAST EQU 9
\end{verbatim}

The range must be a sub-sequence of 14 through 12 (that is, specifications such as \texttt{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:

\begin{verbatim}
CORP RESTORE=(first,last)
\end{verbatim}

or limited to a single saved register by coding

\begin{verbatim}
CORP RESTORE=reg
\end{verbatim}

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.

\begin{verbatim}
X PROC
.
.
CORP RETURN=(2, 7, 9)
\end{verbatim}
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. Reg-
ister 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 address-
ability and have a valid outstanding USING instruction.

The STRCMACS, like any macros, must use certain registers as work reg-
isters. 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 reg-
ister. 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 re-
stored 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)
  ...
  ...
CORP  B
C PROC DEBUG=(NoproCTRACE, PROCCOUNTS)
  ...
  ...
CORP  C
D PROC DEBUG=(ALL, NOSAVETRACE)
  ...
  ...
CORP

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

We will now discuss each of the options in turn.

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

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

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

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

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

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

and is generated in the first proc which requests PROCTRACE.

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

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

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

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

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

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

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

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

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

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

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

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

Addressability, labels, and reentrant code

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

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

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

PROG PROC LINKAGE=(DS, GSECT), DEBUG=(ALL, SDLIST BLOCKS, GLOBAL)

LA 5, P
DO UNTIL, (RCT, 3)
   BAL 14, SJBX
   BAL 14, SJBZ
   OD
   *
   *
   CORP EJECT
   SJBX PROC SAVE=(3, 5)
   *
   *
   L 3, XID
   *
   *
   CORP SUBX EJECT
   SUBX PROC DEBUG=NO DEBUG VALUES
   *
   *
   L 3, YID
   CALL NEXTPROG
   *
   *
   CORP RETURN=3 EJECT
   SUBZ PROC
   *
   *
   L 3, ZID
   LP 6, 3
   IF (C, 5, '=F', '1', EQ)
      BAL 14, SUBY
   FI
   *
   *
   CORP RETURN=6 EJECT
   FINAL
   DS OF
   XID DC C'XX
   YID DC C'YY
   ZID DC C'ZZ
   LTORG
   SPACE 3
   END

43
### Example of Debug Facilities

**LOC** | **OBJECT CODE** | **ADDR1** | **ADDR2** | **STAT** | **SOURCE STATEMENT**
--- | --- | --- | --- | --- | ---
000000 | 47F0 PO0C | 0000C | 6+ | USING *15 | 2 QURPB RG PROC
000001 | 07DB64927D906C7 | 00000C | 7+ | B $P01AA | 5+ QURPB RG CSECT
000002 | 90DC 00C | 0000C | 8+ | DC AL1(7), CL7'QURPB RG' | 9+ QP01AA STM 14, 12, 12(13)
000003 | 4520 FO0C | 0000C | 9+ | CROP 0, 4 | 10+ CROP 0, 4
000004 | 2870 0002 0094 | 00004 | 11+ | DC A($TRACE),(18-1)F'O' | 12+ SP01ISV D0203 0094 4520 F05Z
000005 | 2870 0002 0094 | 00004 | 13+ | DC $STRACE 'FF', | 14+$P01BB 00005000
000006 | 5000 PO0C | 0000C | 15+ | L 13,'A($LASTSAV)' | 16+ ST 13,$14(2)
000007 | 4000 PO0C | 0000C | 17+ | DC A($TRACE),(18-1)F'O' | 18+ LB 13,2
000008 | 4000 PO0C | 0000C | 19+ | DC C'STRACE' | 20+ DC C'STRACE'
000009 | 4000 PO0C | 0000C | 21+ | DC 2,$P01BB | 22+ DC 256,$TRACE+1
00000A | 4000 PO0C | 0000C | 23+ | DC 2,$P01BB | 24+ DC 256,$TRACE+1

### Note

- **PROC** LINKAGE=(OSCECT), DEBUG=(ALL, NOISTBLOCS, GLOBAL)
- **3** 4, **STICKY WARNING**—SAVETRACE REQUIRES "FINAL" MACRO
- **4** 4, **STICKY PROC QURPB RG, DEBUG ID=1'01'
### 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>0001E6</td>
<td>47FC</td>
<td>D1DA</td>
<td>001E8</td>
<td>61</td>
<td>SUB1 PROC SAVE=(3,5)</td>
</tr>
<tr>
<td>0001EE</td>
<td>90BC</td>
<td>D180</td>
<td>30204</td>
<td>62</td>
<td>* STS08108 PROC SUBX, DEBUG ID=X'02'</td>
</tr>
<tr>
<td>0001F2</td>
<td>47FC</td>
<td>D22C</td>
<td>00240</td>
<td>63</td>
<td>SUBX B $P02AA</td>
</tr>
<tr>
<td>0001F8</td>
<td>PP020000</td>
<td>001F8</td>
<td>66</td>
<td>DC X'02D'</td>
<td></td>
</tr>
<tr>
<td>0001FC</td>
<td>00000000000000270</td>
<td>00240</td>
<td>67</td>
<td>DC A(0,$P02NXT)</td>
<td></td>
</tr>
<tr>
<td>000240</td>
<td>0000000000000000</td>
<td>00240</td>
<td>68</td>
<td>DC (15)'0'</td>
<td></td>
</tr>
<tr>
<td>00024A</td>
<td>9200</td>
<td>1100</td>
<td>00100</td>
<td>69</td>
<td>MVC 256(1),X'02t'</td>
</tr>
<tr>
<td>00024E</td>
<td>4810</td>
<td>D155</td>
<td>001FA</td>
<td>70</td>
<td>MVC 0(256,1),1(1)</td>
</tr>
<tr>
<td>000252</td>
<td>4111</td>
<td>0001</td>
<td>00001</td>
<td>71</td>
<td>MVC 1,1(1)</td>
</tr>
<tr>
<td>000256</td>
<td>4010</td>
<td>D185</td>
<td>001FA</td>
<td>72</td>
<td>MVC 1,$P02SV+2</td>
</tr>
<tr>
<td>00025A</td>
<td>92FF</td>
<td>D1E4</td>
<td>00498</td>
<td>73</td>
<td>MVC $P02SV,X'FE'</td>
</tr>
<tr>
<td>00025B</td>
<td>5830</td>
<td>D484</td>
<td>00498</td>
<td>74</td>
<td>MVC 0(256,1),1(1)</td>
</tr>
<tr>
<td>00026A</td>
<td>9B35</td>
<td>D204</td>
<td>00218</td>
<td>75</td>
<td>MVC 3,5,$P02SV+32</td>
</tr>
<tr>
<td>000266</td>
<td>92FC</td>
<td>D260</td>
<td>0027C</td>
<td>76</td>
<td>STM 14,12,$P02SV+12</td>
</tr>
<tr>
<td>000269</td>
<td>07FF</td>
<td>0027E</td>
<td>87</td>
<td>88</td>
<td>BR 14</td>
</tr>
<tr>
<td>000270</td>
<td>FC020000</td>
<td>00270</td>
<td>89</td>
<td>MVC $P02CRP, X'FC020000'</td>
<td></td>
</tr>
<tr>
<td>000274</td>
<td>00010F000002C9</td>
<td>0027C</td>
<td>90</td>
<td>MVC $P02CRP, X'FC020000'</td>
<td></td>
</tr>
<tr>
<td>00027C</td>
<td>0000000000000000</td>
<td>0027C</td>
<td>91</td>
<td>MVC $P02FND</td>
<td></td>
</tr>
</tbody>
</table>

---

**Page 3**

**23 Jan 74**

**00016000**

**00017000**

**00018000**

**00019000**

**00020000**

**00021000**

**00022000**
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>00002BC 47F0 D2AC</td>
<td>002C0</td>
<td>97*SUBY</td>
<td>B</td>
<td>$P03AA</td>
<td></td>
</tr>
<tr>
<td>00002C0 907C D2C0</td>
<td>032A4</td>
<td>99*P03AA</td>
<td>STM</td>
<td>14,12,$P03SV+12</td>
<td></td>
</tr>
<tr>
<td>00002C4 47F0 D2FC</td>
<td>00310</td>
<td>100*</td>
<td>B</td>
<td>$P03DD</td>
<td></td>
</tr>
<tr>
<td>00002C8 00000000</td>
<td>0101+P03SV</td>
<td>DS</td>
<td>OP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00002C8 PF030000</td>
<td>0102+</td>
<td>DC</td>
<td>X'FF030000'</td>
<td>FLAG (FF=FINISHED,FE=ENTERED),</td>
<td>ID, COUN</td>
</tr>
<tr>
<td>00002CC 0000000270000000360</td>
<td>0103+P02PWO</td>
<td>EQU</td>
<td>$P03SV</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00002D4 0000000000000000</td>
<td>0104+</td>
<td>DC</td>
<td>A($P03CRP,$P03NXT)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000310 5A10 D49C</td>
<td>004B0</td>
<td>0105+</td>
<td>DC</td>
<td>(15)F'0'</td>
<td></td>
</tr>
<tr>
<td>0000314 B2FF 1000</td>
<td>0000000000000000000000</td>
<td>0106*P03DD</td>
<td>L</td>
<td>1,=A($TRACE)</td>
<td></td>
</tr>
<tr>
<td>000031A 9203 1100</td>
<td>0013C</td>
<td>0107*</td>
<td>MVC</td>
<td>0(256,1),1(1)</td>
<td></td>
</tr>
<tr>
<td>000031E 4810 D285</td>
<td>002CA</td>
<td>0108*</td>
<td>TFI</td>
<td>256(1),X'03'</td>
<td></td>
</tr>
<tr>
<td>0000322 4111 0001</td>
<td>00001</td>
<td>0109*</td>
<td>LN</td>
<td>1,$P03SV+2</td>
<td></td>
</tr>
<tr>
<td>0000326 4010 D286</td>
<td>002CA</td>
<td>0110*</td>
<td>LA</td>
<td>1,1(1)</td>
<td></td>
</tr>
<tr>
<td>000032A 920E D248</td>
<td>0049C</td>
<td>0111*</td>
<td>STH</td>
<td>1,$P03SV+2</td>
<td></td>
</tr>
<tr>
<td>000032E 5630 D489</td>
<td>0049C</td>
<td>0112*</td>
<td>MVI</td>
<td>$P03SV,X'0F'</td>
<td></td>
</tr>
<tr>
<td>0000332 0700</td>
<td></td>
<td>0113*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000334 47F0 D328</td>
<td>003JC</td>
<td>0114*</td>
<td>CNOP</td>
<td>0,4</td>
<td></td>
</tr>
<tr>
<td>0000339 00000000</td>
<td>0117+</td>
<td>B</td>
<td>*8 BRANCH AROUND VCON</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000033C 58F0 D324</td>
<td>00338</td>
<td>0118+</td>
<td>ISB0040B</td>
<td>DC</td>
<td>V(NEXTPROG)</td>
</tr>
<tr>
<td>0000340 058F</td>
<td>00338</td>
<td>0119+</td>
<td>L</td>
<td>15,ISB0040B</td>
<td>LOAD</td>
</tr>
<tr>
<td>0000342 5030 D2D4</td>
<td>002A8</td>
<td>0121*</td>
<td>BALB</td>
<td>14,15</td>
<td>BRANCH</td>
</tr>
<tr>
<td>0000346 92FF D284</td>
<td>002C3</td>
<td>0122*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000034A 98EC D2C0</td>
<td>002D4</td>
<td>0123*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000034E 07FE</td>
<td>002D4</td>
<td>0124*</td>
<td>CORP</td>
<td>RETURN=3</td>
<td></td>
</tr>
<tr>
<td>0000354 5030 D2D4</td>
<td>002A8</td>
<td>0125*</td>
<td>ST</td>
<td>3,$P03SV+32</td>
<td></td>
</tr>
<tr>
<td>0000356 92FF D284</td>
<td>002C3</td>
<td>0126*</td>
<td>MVI</td>
<td>$P03SV,X'FF'</td>
<td></td>
</tr>
<tr>
<td>000035A 98EC D2C0</td>
<td>002D4</td>
<td>0127*</td>
<td>LHI</td>
<td>14,12,$P03SV+12</td>
<td></td>
</tr>
<tr>
<td>000035E 07FE</td>
<td>002D4</td>
<td>0128*</td>
<td>BR</td>
<td>14</td>
<td></td>
</tr>
</tbody>
</table>

PROC DEBUG=NOCORPVALUES
95 SUBY
* STRC8108 PROC SUBY, DEBUG ID='X'03'
96 B $P03AA
97*SUBY
98+ DC CLW'SUBY'
99+P03AA
STM 14,12,$P03SV+12
100+ B $P03DD
101+P03SV
DS OP
102+ DC X'FF030000' | FLAG (FF=FINISHED,FE=ENTERED), | ID, COUN
103+P02PWO
EQU $P03SV
104+ DC A($P03CRP,$P03NXT)
105+ DC (15)F'0'
106*P03DD
L 1,=A($TRACE)
107* MVC 0(256,1),1(1)
108+ TFI 256(1),X'03'
109+ LN 1,$P03SV+2
110+ LA 1,1(1)
111+ STH 1,$P03SV+2
112+ MVI $P03SV,X'0F'
113*   
114*   
115+ L 3,YID
116+ CALL NEXTPROG
117+ CNOP 0,4
118+ B *8 BRANCH AROUND VCON
119+ISB0040B DC V(NEXTPROG) ENTRY POINT ADDRESS
120+ L 15,ISB0040B LOAD 15 WITH ENTRY ADR
121+ BALB 14,15 BRANCH TO ENTRY POINT
122*   
123*   
124+ CORP RETURN=3
125* ST 3,$P03SV+32
126+ MVI $P03SV,X'FF'
127+ LHI 14,12,$P03SV+12
128+ BR 14
### 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 D344</td>
<td>0035A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000354</td>
<td>82E4C259</td>
<td>0035C</td>
<td>0035B</td>
<td>130</td>
<td>SUBZ PROC *</td>
</tr>
<tr>
<td>000359</td>
<td>03EC D359</td>
<td>0035C</td>
<td></td>
<td>131</td>
<td>STLC0106 PROC SUBZ, DEBUG ID='X'04'</td>
</tr>
<tr>
<td>00035C</td>
<td>47F0 D354</td>
<td>0035A</td>
<td>132</td>
<td>B 'PO4AA'</td>
<td></td>
</tr>
<tr>
<td>000360</td>
<td></td>
<td></td>
<td></td>
<td>133</td>
<td>DC CL4'SUBZ'</td>
</tr>
<tr>
<td>000364</td>
<td>PP040000</td>
<td></td>
<td></td>
<td>134</td>
<td>STM 14,12,'PO4SV+12</td>
</tr>
<tr>
<td>000368</td>
<td>9111 0001</td>
<td>00362</td>
<td>0036C</td>
<td>135</td>
<td>B 'PO4DD'</td>
</tr>
<tr>
<td>000370</td>
<td>9204 1100</td>
<td>00100</td>
<td></td>
<td>136</td>
<td>DC (15) 'PO4SV+2</td>
</tr>
<tr>
<td>00038C</td>
<td>9111 0001</td>
<td>00382</td>
<td>0038C</td>
<td>137</td>
<td>LA 1, 'PO4SV+2'</td>
</tr>
<tr>
<td>000390</td>
<td>9111 0001</td>
<td>00390</td>
<td>0039A</td>
<td>138</td>
<td>STM 14,12,'PO4SV+12</td>
</tr>
<tr>
<td>000394</td>
<td>8F040000</td>
<td>0039A</td>
<td>0039C</td>
<td>139</td>
<td>B 'PO4DD'</td>
</tr>
<tr>
<td>0003A0</td>
<td>9111 0001</td>
<td>003A0</td>
<td>003A4</td>
<td>140</td>
<td>DC (15) 'PO4SV'</td>
</tr>
<tr>
<td>0003A8</td>
<td>9111 0001</td>
<td>003A8</td>
<td>003B0</td>
<td>141</td>
<td>DC X'PO400000' FLAG (FF=FINISHED,FE=ENTERED), ID, COUNT</td>
</tr>
<tr>
<td>0003B8</td>
<td>9111 0001</td>
<td>003B8</td>
<td>003C0</td>
<td>142</td>
<td>DC X'PO4SV'+12</td>
</tr>
<tr>
<td>0003C0</td>
<td>9111 0001</td>
<td>003C0</td>
<td>003C4</td>
<td>143</td>
<td>DC X'PO4SV,PO4SVX'</td>
</tr>
<tr>
<td>0003C8</td>
<td>9111 0001</td>
<td>003C8</td>
<td>003D0</td>
<td>144</td>
<td>DC X'PO4SV,PO4SVX'</td>
</tr>
<tr>
<td>0003D0</td>
<td>9111 0001</td>
<td>003D0</td>
<td>003D4</td>
<td>145</td>
<td>DC X'PO4SV,PO4SVX'</td>
</tr>
<tr>
<td>0003E0</td>
<td>9111 0001</td>
<td>003E0</td>
<td>003E4</td>
<td>146</td>
<td>DC X'PO4SV,PO4SVX'</td>
</tr>
<tr>
<td>0003F0</td>
<td>9111 0001</td>
<td>003F0</td>
<td>003F4</td>
<td>147</td>
<td>DC X'PO4SV,PO4SVX'</td>
</tr>
<tr>
<td>000400</td>
<td>9111 0001</td>
<td>00400</td>
<td>00404</td>
<td>148</td>
<td>DC X'PO4SV,PO4SVX'</td>
</tr>
<tr>
<td>000410</td>
<td>9111 0001</td>
<td>00410</td>
<td>00414</td>
<td>149</td>
<td>DC X'PO4SV,PO4SVX'</td>
</tr>
<tr>
<td>000420</td>
<td>9111 0001</td>
<td>00420</td>
<td>00424</td>
<td>150</td>
<td>DC X'PO4SV,PO4SVX'</td>
</tr>
<tr>
<td>000430</td>
<td>9111 0001</td>
<td>00430</td>
<td>00434</td>
<td>151</td>
<td>DC X'PO4SV,PO4SVX'</td>
</tr>
<tr>
<td>000440</td>
<td>9111 0001</td>
<td>00440</td>
<td>00444</td>
<td>152</td>
<td>DC X'PO4SV,PO4SVX'</td>
</tr>
<tr>
<td>000450</td>
<td>9111 0001</td>
<td>00450</td>
<td>00454</td>
<td>153</td>
<td>DC X'PO4SV,PO4SVX'</td>
</tr>
<tr>
<td>000460</td>
<td>9111 0001</td>
<td>00460</td>
<td>00464</td>
<td>154</td>
<td>DC X'PO4SV,PO4SVX'</td>
</tr>
<tr>
<td>000470</td>
<td>9111 0001</td>
<td>00470</td>
<td>00474</td>
<td>155</td>
<td>DC X'PO4SV,PO4SVX'</td>
</tr>
<tr>
<td>000480</td>
<td>9111 0001</td>
<td>00480</td>
<td>00484</td>
<td>156</td>
<td>DC X'PO4SV,PO4SVX'</td>
</tr>
<tr>
<td>000490</td>
<td>9111 0001</td>
<td>00490</td>
<td>00494</td>
<td>157</td>
<td>DC X'PO4SV,PO4SVX'</td>
</tr>
<tr>
<td>0004A0</td>
<td>9111 0001</td>
<td>004A0</td>
<td>004A4</td>
<td>158</td>
<td>DC X'PO4SV,PO4SVX'</td>
</tr>
<tr>
<td>0004B0</td>
<td>9111 0001</td>
<td>004B0</td>
<td>004B4</td>
<td>159</td>
<td>DC X'PO4SV,PO4SVX'</td>
</tr>
<tr>
<td>0004C0</td>
<td>9111 0001</td>
<td>004C0</td>
<td>004C4</td>
<td>160</td>
<td>DC X'PO4SV,PO4SVX'</td>
</tr>
<tr>
<td>0004D0</td>
<td>9111 0001</td>
<td>004D0</td>
<td>004D4</td>
<td>161</td>
<td>DC X'PO4SV,PO4SVX'</td>
</tr>
<tr>
<td>0004E0</td>
<td>9111 0001</td>
<td>004E0</td>
<td>004E4</td>
<td>162</td>
<td>DC X'PO4SV,PO4SVX'</td>
</tr>
<tr>
<td>0004F0</td>
<td>9111 0001</td>
<td>004F0</td>
<td>004F4</td>
<td>163</td>
<td>DC X'PO4SV,PO4SVX'</td>
</tr>
</tbody>
</table>

* 23 JAN 74
### 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>000000</td>
<td>000496</td>
<td>000498</td>
<td>FINAL</td>
<td>$LASTSAV EQU $P04BCK</td>
</tr>
<tr>
<td>000496</td>
<td>E7E7E7E7</td>
<td>000498</td>
<td>000498</td>
<td>DS OF</td>
<td>187 $P04WD EQU 0</td>
</tr>
<tr>
<td>000498</td>
<td>E7E7E7E7</td>
<td>000498</td>
<td>000496</td>
<td>DS OF</td>
<td>186 XID DC C'XXX'</td>
</tr>
<tr>
<td>000498</td>
<td>E7E7E7E7</td>
<td>000498</td>
<td>000496</td>
<td>DS OF</td>
<td>185 YID DC C'YYY'</td>
</tr>
<tr>
<td>000498</td>
<td>E7E7E7E7</td>
<td>000498</td>
<td>000496</td>
<td>DS OF</td>
<td>184 ZID DC C'ZZZ'</td>
</tr>
<tr>
<td>000498</td>
<td>E7E7E7E7</td>
<td>000498</td>
<td>000496</td>
<td>DS OF</td>
<td>183 LTORG</td>
</tr>
</tbody>
</table>

197 END
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

A-1
in macro syntax). In addition, angle brackets "<" and ">" may be specified as operands for the grouping of subexpressions. For example:

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

(*)

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

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

is the same as

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

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

Only as many of the simple conditions are evaluated as are required to determine the value of the entire conditional expression. In the example (*) above, if the value of \( scond1 \) is false, the expression must be false so the remaining two simple conditionals are not evaluated.

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

A-2
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.

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

**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.

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

**block-name**

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

**other-ops**

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

*BLOCK—Define a Simple Block of Code*

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

```
[blname]   BLOCK
```

*blname*  sym

The name associated with this BLOCK block and to be defined on the first instruction generated.
CASE—Define a DOCASE Alternative

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

<table>
<thead>
<tr>
<th>blname</th>
<th>CASE</th>
</tr>
</thead>
<tbody>
<tr>
<td>index-list</td>
<td></td>
</tr>
<tr>
<td>char-index-list</td>
<td></td>
</tr>
<tr>
<td>conditional-test</td>
<td></td>
</tr>
</tbody>
</table>

blname

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

MISC

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

index-list

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

char-index-list

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

\[
\begin{array}{|c|c|}
\hline
[\text{label}] & \text{CORP} \\
\hline
 & [\text{proc-name}] \\
 & \{\text{RESTORE}=(\text{first} [,\text{last}])\} \\
 & \{\text{RETURN}=\text{reg-list}\} \\
 & \{\text{RC}=[\text{value}(\text{reg})]\} \\
 & \{\text{LINK}=[\text{NONE}(\text{linkreg})]\} \\
\hline
\end{array}
\]

\text{label} \quad \text{sym}

If present, \text{label} will appear on the first instruction generated.

\text{proc-name} \quad \text{sym}

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

\text{RESTORE}=(\text{first}, \text{last}) \quad \text{dec dig, sym}

Indicates the first and last registers to be restored. These must be a sub-sequence of those saved. If \text{last} is not specified, only register \text{first} will be restored. If the entire operand is omitted, all registers saved will be restored.

\text{RETURN}=\text{reg-list} \quad \text{dec dig, sym}

One or more registers which would otherwise be restored but which are to be exceptions. The registers in the \text{RETURN=} list may be thought of as output values being returned to the caller. Used mainly for non-OS procs.
RC=None

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

RC=value

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

RC=(reg)

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

If RC= is not coded the defaults are:

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

LINK=None

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

LINK=linkreg

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

If LINK= is omitted, LINK=14 is assumed.
DO – Define Iterative Block

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

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

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

looping-branch

One of the special looping instructions specified as:

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

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, \((\text{BCT}, \text{regl})\) will loop one time less than the initial value in \text{regl}.

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

biname

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

index-word

RX-type

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

(index-reg)

dec dig, sym

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

(index,W)

RX-type

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

(index,H)

RX-type

Indicates the DOCASE index is located in the half-word at address index.
(index, B)

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

(index, length)

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

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

SIMPLE

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

SPARSE

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

IFANY

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

ONLY

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

|       | DOCASEND | \[block-name\] |
DOEND—Terminate Iteration Block

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

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

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

`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></th>
<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.

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

**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

*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* sym

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

**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.

<table>
<thead>
<tr>
<th>FINAL</th>
</tr>
</thead>
</table>

A-23
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.

<table>
<thead>
<tr>
<th>bname</th>
<th>IF</th>
<th>{ ASYNCH }</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>{ cond-test }</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[ EXIT= { exit-block } ]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[ ELSE= else-block ]</td>
</tr>
</tbody>
</table>

bname

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

ASYNCH

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

cond-test

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

EXIT=exit-block

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

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

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

ELSE= else-block

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

ONEND—Define Normal Loop Termination Code

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

| ONEND | [block-name] |
ONEXIT—Define Abnormal Loop Termination Code

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

\[
\begin{array}{|c|c|}
\hline
\text{ONEXIT} & \text{[block-name]} \\
\hline
\end{array}
\]

\textit{block-name}

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

### PROCS

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

No in-line identifier is to be generated.

ID=id-string

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

ID=*  

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

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

For OS procs, ID=*
For non-OS procs, ID=NONE

SAVE=None

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

SAVE=(first, last)

dec dig, sym

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

SAVE=(, , DYNAM)

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

A-31
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 )

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
PROC

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

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

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

DEBUG=options-list

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

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

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

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

BLOCKNAMES[BN]—Each block’s name is to be generated as an in-line character constant.
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., "NOPROCTRACE" or "NPT"). When the CORP is generated, options will revert to their status before the PROC macro. To avoid the restoring of the options' status at CORP time, include "GLOBAL" (or "GBL"). "ALL" or "NONE" may be specified to turn on or off all options; either may be followed by exceptions. (e.g., "DEBUG=(ALL,NST)" turns on all options except the save-trace.)
PROCEND—Terminate a Proc

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

```
[label]  PROCEND  [proc-name]
  [RC=  \{NONE\}]
  (value {reg})
  [RESTORE=(first [,last])]
  [RETURN=reglist]
  [LINK=  \{NONE\}]
```
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),\textsuperscript{1} OS macro assembly language, and the STRCMACS themselves. After the code was written in SIMPL-M, it was translated by hand to OS macro assembly language. The SIMPL-M program is considered the "source" code and all updates are performed in it. It is much easier to read than the macro assembly language source. The SIMPL-M source for the macros is listed in Appendix C. In this appendix, we will give a brief description of the SIMPL-M language. In addition, a decision table for the DO macro formats is included in this appendix to complete the source documentation.

Introduction to SIMPL-M

SIMPL-M is a high-level language for the specifying of assembly language macros. In some ways it resembles ALGOL or PL/I; it provides for 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 &PROCNAME, &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 → false; all else to true
- **int** to **char**: the absolute value of the integer is expressed as characters without leading sign or zeros
- **bit** to **int**: false → 0; true → 1
bit to char: false → '0'; true → '1'

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

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

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

```
int X(20)
```

and are referred to as

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

The first element of the array has the index 1.

Macro operands are either positional (determined by order) or keyword (determined by the fixed term 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:

```
if I = 1 or I = 19
```

is the same as

```
if I = 1 or I = 19
```

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

```
if J ≠ 0 and I / J = 4
```

to be evaluated without an underflow occurring.
The body of a *macro* is terminated by a *mend* instruction. The *mexit* instruction causes immediate exit from the *macro* definition. Character strings are concatenated by using the "\|" operator.

\[
X := 'ABC'
\]

\[
Y := X \| 'DEF'
\]

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

\[
X := 'ABC'
\]

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

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

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

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

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

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

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

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

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

\[
/\ast\{\ldots\}\ast/
\]
**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 (con-
trol falls through to the indicated node). Boxes shown cross-hatched do not
occur for that operand combination. An example follows the table.
## FOREVER

<table>
<thead>
<tr>
<th>WHILE</th>
<th>looping-branch</th>
<th>AND/OR</th>
<th>cond-test A</th>
<th>AND/OR</th>
<th>cond-test B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## WHILE

<table>
<thead>
<tr>
<th>looping-branch</th>
<th>AND/OR</th>
<th>cond-test A</th>
<th>AND/OR</th>
<th>cond-test B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## UNTIL

<table>
<thead>
<tr>
<th>looping-branch</th>
<th>AND/OR</th>
<th>cond-test A</th>
<th>AND/OR</th>
<th>cond-test B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Entry point - 0

<table>
<thead>
<tr>
<th>Condition</th>
<th>To node</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 7 1 1</td>
</tr>
</tbody>
</table>

### Branch' leg 2

<table>
<thead>
<tr>
<th>Condition</th>
<th>To node</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>12 7 1 1</td>
</tr>
</tbody>
</table>

### 'Fall thru' leg 3

<table>
<thead>
<tr>
<th>Condition</th>
<th>To node</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>12 7 1 1</td>
</tr>
</tbody>
</table>

### Branch' leg 5

<table>
<thead>
<tr>
<th>Condition</th>
<th>To node</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>12 7 1 1</td>
</tr>
</tbody>
</table>

### 'Fall thru' leg 5

<table>
<thead>
<tr>
<th>Condition</th>
<th>To node</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>12 7 1 1</td>
</tr>
</tbody>
</table>

### After Code - 8

<table>
<thead>
<tr>
<th>Condition</th>
<th>To node</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>12 7 1 1</td>
</tr>
</tbody>
</table>

### Looping Branch

<table>
<thead>
<tr>
<th>Branch' leg 10</th>
<th>To node</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>12 7 1 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>'Fall thru' leg 11</th>
<th>To node</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>12 7 1 1</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, (CLI,SPOT,C'X',E)

Operand format number 6.

Code generated:

B  $1BW1
$1BU1 CLI SPOT,C'X'
BE $1BEND
$1BW1 LTR 3,3
BNP $1BEND

<Block's code>

BCT 5,$1BU1
$1BEND DS OH

START
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 "$Pep". The
137. SP is PROC_COUNTER in hex. */
138. LAST_BLOCK_NUMBER, /* Number used in labels of most recently generated
139. block. */
140. HEX_IN /* Input value to XHEX macro. */
141. /* GENERAL PURPOSE GLOBALS. */
142. int
143. PROC_COUNTER, /* Number for last special proc label "$Pep". The
144. Ep is PROC_COUNTER in hex. */
145. LAST_BLOCK_NUMBER, /* Number used in labels of most recently generated
146. block. */
147. HEX_IN /* Input value to XHEX macro. */
148. bit
149. BLOCK_LABEL_PREFIX, /* Unique character string for each block
150. for use in generating labels. */
151. EXIT_SEVERITY, /* Mnote severity for EXIT target message. Can be
152. set by proc macros. */
153. HEX(16), /* Constants used in converting decimal to hex by XHEX. */
154. HEX_OUT, /* Output value from XHEX macro. */
155. PREV_SAVETRACE_AREA, /* Holds label generated on last local PROC
156. save area to be used in producing the static chain for SAVETRACE. */
157. /* DEBUG FLAGS. */
158. bit
159. DEBUG_BLOCKCOUNTS_REQD, /* Causes code and counters to keep execution
160. counts on all blocks. */
161. DEBUG_BLOCKNAMES_REQD, /* Causes block names to be generated as
162. inline character constants to aid in locating within dumps. */
163. DEBUG_CORPVALUES_REQD, /* Causes register values at end of proc (at
164. CORP macro start) to be saved in inline save areas for reference. */
165. DEBUG_MACRONAMES_REQD, /* Causes macro names to be generated whenever any
166. macros are entered (including inner macros) which list the macro's
167. name; for debugging the macros. */
168. DEBUG_PROCCOUNTS_REQD, /* Causes code and counters to keep execution
169. counts on PROC blocks only. */
170. DEBUG_PROCTRACE_REQD, /* Causes a trace vector to be generated and
171. code to be generated to keep track of the last 257 PROCs entered. */
172. DEBUG_SAVETRACE_REQD /* Causes all local save areas to be statically
173. chained together and code to be generated to link the chain to the
174. OS save area to provide OS formatting within ABEND dumps. */
Global Definitions -- 14 July 1973

197. /* MAIN STACK. Dimensioned to 100. */
198. int CURRENT_NEST_LEVEL, /* Current depth of static nesting of
199. blocks; stack pointer. */
200. NESTING_LIMIT, /* Holds dimension of main stack. */
201. BLOCK_NUMBER(100) /* Block number of the Ith block. */
202. END_LABEL_REQD(100), /* Indicates whether Ith block needs an END label
203. generated during POP_OLD_BLOCK. */
204. EXIT_LABEL_REQD(100) /* Indicates whether Ith block needs an EXIT label
205. generated during POP_OLD_BLOCK. */
211. char BLOCK_NAME(100), /* Block name of Ith block, either USER_NAME specified
212. in macro label field or generated name "BLKnn" where nn is the
213. sequential block number. */
214. BLOCK_TYPE(100), /* Macro name which generated the Ith block
215. (IF, DO, DOCASE, CASE, BLOCK, or PROC). */
216. OPERAND1(100), OPERAND2(100), OPERAND3(100), OPERAND4(100),
217. /* These hold various data which are needed to close the blocks
218. generated. Specific contents vary according to the type of
219. block generated. See individual macros. */
220. INFORMATION(100) /* Similar to the OPERAND stacks above, the
221. INFORMATION stack holds information for the closing of the block.
222. Often the individual characters within the variables are used for
223. different values, packed together into INFORMATION. */
227. /* GCASE STACK. Holds data for general DOCASEs. Dimensioned to 9. */
229. int MAX_CASE_VALUE(9), /* Maximum branch vector value found. */
230. NEXT_CMP_LABEL_NO(9), /* Case number for next comparison case label
231. to be generated. */
232. GCASE_NEST_LEVEL, /* Current depth of stacking in the GCASE stack;
233. number of nested DOCASES with either GENERAL, SPARSE, or CHARCOMP
234. operand formats. */
235. GCASE_NEST_LIMIT /* Maximum depth of nesting of GCASE stack; must
236. be equal to stack dimension. */
244. /* CONDITIONAL_EXPRESSION_PROCESSOR PSEUDO-PARAMETERS. */
247. int FIRST_INDEX,
248. LAST_INDEX /* Pseudo-parameters to CONDITIONAL_EXPRESSION_PROCESSOR.
249. Indicates indexes within SYSLIST of first and last parameter to be
250. processed. */
253. ULTIMATE_FALLTHRU_CONDITION, /* Logical value upon which conditional
254. expression is to pass control (or fall through) to the ULTIMATE_ 
255. FAILTHRU_LABEL. */
256. FAILTHRU_LABEL_USED /* CEP sets this true if a branch is generated
257. to the ULTIMATE_FAILTHRU_LABEL (else no change occurs). */
259. char ULTIMATE_BRANCH_LABEL, /* Indicates label to be used as branch target
260. when conditional test does not have logical value stored in 
261. ULTIMATE_FALLTHRU_CONDITION. */
263. ULTIMATE_FAILTHRU_LABEL, /* Indicates label available as branch target
264. when conditional test has logical value stored in ULTIMATE_ 
265. FAILTHRU_CONDITION. If used as a branch target, FAILTHRU_ 
266. LABEL_USED must be set true (by CEP) to insure next sequential 
267. instruction following conditional expression receives label 
268. definition. */
269. UNIQUE_LABEL_ID /* One character unique to this call of CEP used to
270. insure labels generated by this call will differ from all other 
271. labels, even others within the same macro (particularly for DO). */
**IF** macro — 21 June 1973

```
11001. BASED **IF** (USER_NAME; REL=, MASK=, EXIT=, ELSE=)
11003. /* Initiate a block in the structure. Save any information needed
11004. by ELSE or FI. For ASYNCH type, generate branch around block.
11005. For normal IF (EXIT= not specified), generate conditional expression
11006. tests with branch around block (or to ELSE) for false and fall
11007. through for true; if EXIT= specified, then generate branch to
11008. proper block end for true, fall through for false, and delete
11009. IF block from structure. Put USER_NAME on first executable
11010. instruction if one specified. */
11011. bit VALID_EXIT
11012. /* VALID_EXIT is true if EXIT= was specified and no errors have
11013. been found to cause the EXIT to be ignored. */
11014. char EXIT_LABEL,
11016. /* Label for EXIT= branch, when deferred until
11017. after block count has been incremented. */
11018. LABEL /* Outstanding label, waiting to be generated. */
11020. call TRACE_PRINTER ( ; "IF") /* Prints macro name "IF" in mnote if tracing ca. */
11021. call PUSH_NEW_BLOCK(USER_NAME;
11023. BLOCK_TYPE_VALUE='IF',
11024. OPERAND1 VALUE=ELSE,
11025. END_LABEL_VALUE=true)
11026. /* Define new block; add to stack. Initialize block specifications. Assume block will require an END label. Note block type and save name of ELSE block if one specified here. Set up unique BLOCK_LABEL_PREFIX for use in generating unique labels. */
11027. if ERROR_OCCURRED
11028. then
11029. mexit
11030. fi
11032. fi
11034. if REL "" OR MASK ""
11035. then
11036. PRINT (B, "STRC1102 REL= OR MASK= NOT IN PARENTHESES—IGNORED")
11038. LABEL := USER_NAME
11039. /* Generate USER_NAME at first opportunity. */
11040. VALID_EXIT := (EXIT "")
11041. /* Set VALID_EXIT to the truth of whether EXIT= was specified. */
11042. if SYSLIST(1,1) = "ASYNCH"
11043. then /* Either "IF ASYNCH" or "IF (ASYNCH)" was entered. */
11044. call IF ASYNCH BRANCH /* Generate branch around block. */
11045. else
11046. call IF_SET_CONDITIONAL_TEST_SPECS
11047. /* Set all conditional test specifications in globals required to
11048. define the action to be performed by the conditional test
11049. generators. */
11050. if VALID_EXIT /* i.e., if EXIT specified and still valid... */
11051. then
11052. call IF_EXIT_SPECS
11053. /* Reset conditional test specs according to EXIT target provided
11054. no conflicting parameters exist (in that case, set VALID_EXIT to
11055. false). */
11056. fi
11057. call IF_CONDITIONAL_GENERATOR
11058. /* Generate conditional tests according to IF macro operands and
11059. the current conditional test specs. */
11060. fi
11061. call IF_BLOCK_COUNT
11063. /* Now that we're into the block's execution, do any debugging counting
11064. that is required. */
11065. if LABEL ""
11066. then
11067. GENERATE (LABEL |1 ' DS ON*)
11068. fi
11069. call IF VALID_EXIT
11070. /* No IF block remains after completion of "IF EXIT..." macro;
11071. simulate presence of FI macro. */
11072. call PI ( ; )
11074. end
```
11076. 

11077. BLOCK IF_ASYNCH_BRANCH
11078. /* Give error message if EXIT specified. Generate branch to
11079. end of IF block. */
11080. if VALID_EXIT
11081. then
11082. note (S, 'STSC1101 EXIT- IGNORED WITH "ASYNCH"')
11083. VALID_EXIT := false
11084. end
11085. generate (LABEL || ' B ' || BLOCK_LABEL_PREFIX || '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 logical value which is the one upon which the overall test is
11099. to fall through
11100. FALLTHRU_LABEL_USED false until a branch is required within
11101. the testing structure to the fall-through label.
11102. All of the above are global variables. */
11103. /* Set the normal conditional test specs. */
11104. ULTIMATE.Branch_LABEL := BLOCK_LABEL_PREFIX || 'END'
11105. /* Branch target for false result is END label—end of IF or
11106. start of ELSE. */
11107. ULTIMATE_FALLTHRU_LABEL := BLOCK_LABEL_PREFIX || 'BEG'
11108. /* Fall-through label to be used is BEG. */
11109. ULTIMATE.FALLTHRU_CONDITION := true
11110. /* Fall through if conditional test yields true result. */
11111. FALLTHRU_LABEL_USED := false /* Assume not required. */
11112. CORP
In the beginning of the code, there is a comment that explains the purpose:

```
1116. PROC IF_EXIT_SPECS
1117. /* An EXIT operand has been specified; insure ELSE not also
   specified. If valid, change conditional test specs to standard
   for EXIT-type IF including the assigning of the branch target
   as the XIT label of the block specified by the EXIT keyword. */
1118. char HOLD /* Temporary. */
1119.
1120. if ELSE " /* If ELSE was not omitted... */
1121. then
1122. msg ( "STRC1103 EXIT= IGNORED WITH ELSE=")
1123. VALID_EXIT := false
1124. else
1125. HOLD := ULTIMATE_BRANCH_LABEL
1126. /* Save old branch label, we may need it yet. */
1127. else
1128. call EXIT_FIND ( ; EXIT)
1129. /* Sets ULTIMATE_BRANCH_LABEL to XIT label of block whose name
   is specified in the argument; if none specified ("EXIT="),
   use block surrounding IF macro; if no such block, issue message,
   leave ULTIMATE_BRANCH_LABEL unmodified, and set ERROR_OCCURRED to
   true. Mark target block as requiring XIT label. */
1130. if DEBUG_BLOCKCOUNTS_REQD
1131. then
1132. EXIT_LABEL := ULTIMATE_BRANCH_LABEL
1133. ULTIMATE_BRANCH_LABEL := HOLD
1134. /* Make EXIT-type IF act like regular IF (i.e., fall through on true)
   so we can count the number of times the exit is taken; save the
   EXIT_LABEL for a branch after the count is made and make the
   ULTIMATE_BRANCH_LABEL whatever it would have been had this been
   a regular IF. */
1135. else
1136. ULTIMATE_FALLTHRU_LABEL := BLOCK_LABEL_PREFIX
1137. ULTIMATE_FALLTHRU_CONDITION := false /* Fall through on false. */
1138. END_LABEL_REQD(CURRENT_NEST_LEVEL) := false
1139. if ERROR_OCCURRED /* on EXIT_FIND... */
1140. then
1141. /* Exit point not found and message has been issued. Make branch
   point same as fall-through point and clear error (i.e., fix up
   and continue). */
1142. ULTIMATE_BRANCH_LABEL := BLOCK_LABEL_PREFIX
1143. FALLTHRU_LABEL_USED := true
1144. /* Since it's also the branch label. */
1145. ERROR_OCCURRED := false
1146. fi
1147. fi
1148. fi
1149. fi
1150. fi
1151. fi
1152. fi
1153. fi
1154. fi
1155. fi
1156. fi
1157. fi
1158. fi
1159. fi
1160. fi
1161. fi
```
"IF" Macro -- 21 June 1973

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.         LABEL := ULTIMATE_FALLTHRU_LABEL
11200.     ELSE
11201.         LABEL := ""
11202.     FI
11203.     CORP
11206. PROC IF_BLOCK_COUNT
11207. /* If debugging in progress, generate block name and/or count of block execution. Note that ASYNCH blocks cannot be counted. */
11208. IF TARGET /* Temporary. */
11209. IF SYSLIST(1,1) = 'ASYNCH'
11210. then
11211. IF DEBUG_BLOCKNAMES_REQD
11212. then
11213. generate ("DC C'" || BLOCK_NAME(CURRENT_NEST_LEVEL) || 
11214. '*O0')
11215. /* Asynch branch has already occurred; only name required. */
11216. else /* Not ASYNCH. */
11217. if DEBUG_BLOCKCOUNTS_REQD or DEBUG_BLOCKNAMES_REQD
11218. then
11219. if DEBUG_BLOCKCOUNTS_REQD
11220. then
11221. /* Generate code to increment block execution count. */
11222. generate (LABEL II '*LH'I'BLCCK_LABEL_PREFIX II 'IFC')
11223. LABEL := '
11224. generate ("LA'I'BLCCK_LABEL_PREFIX II 'IFC')
11225. if EXIT_LABEL = '
11226. TARGET := BLOCK_LABEL_PREFIX II 'GO'
11227. /* Branch directly around block name/count. */
11228. else
11229. TARGET := EXIT_LABEL
11230. /* Branch to end of EXIT= block, postponed to here so we could do the counting. */
11231. if EXIT_LABEL = '
11232. generate (LABEL II 'B'I'TARGET)
11233. if EXIT_LABEL = '
11234. then
11235. LABEL := TARGET
11236. /* Label for branch-around must be defined. */
11237. else
11238. LABEL := '
11239. if DEBUG_BLOCKNAMES_REQD
11240. then
11241. generate ("DC C'" || BLOCK_NAME(CURRENT_NEST_LEVEL) || 
11242. '*O0')
11243. if DEBUG_BLOCKCOUNTS_REQD
11244. then
11245. generate (BLOCK_LABEL_PREFIX II 'IFC DC H'O' IF COUNT')
11246. fi
11247. fi
11248. return
11249. else
11250. generate ("DC C'" || BLOCK_NAME(CURRENT_NEST_LEVEL) || 
11251. '*O0')
11252. if DEBUG_BLOCKCOUNTS_REQD
11253. then
11254. generate (BLOCK_LABEL_PREFIX II 'IFC DC H'O' IF COUNT')
11255. fi
11256. fi
11257. fi
11258. end
11259. */
**ELSE** Macro -- 26 June 1973

13001. **BEGIN** 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. call TRACE_PRINTER ( ; 'ELSE') /* Print macro name "ELSE" in mnote if tracing on. */
13012. if CURRENT_NEST_LEVEL > NESTING_LIMIT
13013. then
13014. mexit
13015. call VERIFY_END ( ; 'IF', BLEND)
13016. /* Verifies current block has the name specified
13017. by the BLEND= operand on the ELSE macro (if any) and that it is an IF block.
13018. Various errors receive messages and either intermediate blocks are
13019. BLENDed as a fixup or ERROR_OCCURRED is set. */
13020. if ERROR_OCCURRED
13021. then
13022. mexit
13023. fi
13024. if OPERAND2(CURRBENT_NEST_ LEVEL) = 'ELSE'
13025. then
13026. nnote(8, 'STRC1302 ELSE HAS ALREADY BEEN GENERATED FOR CURRENT IF')
13027. mexit
13028. fi
13029. else
13030. if OPERAND1(CURRENTNEST_LEVEL) # USER_NAME
13031. then
13032. /* ELSE=elsename specified on IF macro but different (or no) label
13033. field on ELSE macro. */
13034. mnote(8, 'STRC1301 ELSE=' OPERAND1(CURRENTNEST_LEVEL) 'SPECIFIED ON IF BLOCK' 'BLOCK_NAME(CURRENT_NEST_LEVEL)')
13035. mexit
13036. fi
13037. ELSE BLOCK NO := LAST BLOCK NUMBER + 1
13038. /* Generate block number of upcoming ELSE block. */
13039. if OPERAND1(CURRENT_NEST_LEVEL) # 'ELSE'
13040. then
13041. /* ELSE=elsename specified on IF macro but different (or no) label
13042. field on ELSE macro. */
13043. else
13044. GENERATE ('B ' ELSE_BLOCK_NO 'END') /* Generate branch to end of ELSE block. */
13045. fi
13046. if EXIT_LABEL_REQD(CURRENT_NEST_LEVEL)
13047. then
13048. IF_EXIT_LABEL := $ BLOCK_NUMBER(CURRENT_NEST_LEVEL) 'XIT'
13049. EXIT_LABEL_REQD(CURRENT_NEST_LEVEL) := false
13050. /* Exit label has been postponed to FI. */
13051. fi
13052. call POP OLD BLOCK ( ; ) /* Remove IF block from stack. */
13053. call PUSH NEW BLOCK (USER_NAME; BLOCK_TYPE_VALUE='IF', /* Must be marked as "IF" block for FI's
13054. END_LABEL_VALUE=none, call to VERIFY_END. */
13055. OPERAND2_VALUE='ELSE',
13056. OPERAND1_VALUE=IF_EXIT_LABEL)
13057. /* Generate new block in structure. It is marked as an IF block
13058. to simplify FI checking; "ELSE" is stored in OPERAND2, however,
13059. to indicate that an ELSE has been generated for this IF. An END
13060. label will be required for the block and, if the IF block was
13061. marked as needing an XLT label, save it in OPERAND3. */
13062. if USER_NAME # ''
13063. then
13064. GENERATE (USER_NAME || ' DS OH') /* Define outstanding label. */
13065. fi
13066. fi
13067. if IF_EXIT_LABEL ||
13068. /* Define outstanding label. */
13069. fi
13070. mend
**FI** Macro -- 26 June 1973

15001. **MACRO** FI ( USER_NAME)
15002. /* Generates end to match IF (or ELSE) block. Standard block closing
15003.  occurs. */
15005.  **call** TRACE_PRINTER ( FI)
15006.  /* Prints macro name "FI" in mnote if tracing on. */
15007.  **if** CURRENT_NEST_LEVEL > NESTING_LIMIT
15008.  **then**
15009.  **call** POP_OLD_BLOCK ( )
15010.  **begin**
15011.  **fi**
15012.  **call** VERIFY_END ( IF, USER_NAME)
15013.  /* Verifies current block has the name specified by the USER_NAME
15014.  operand on the FI macro (if any) and that it is an IF block.
15015.  Various errors receive messages and either intermediate blocks are
15016.  **BLENDED** as a fixup or ERROR_OCCURRED is set.
15017.  (Lemma: If CURRENT_NEST_LEVEL > 0 and
15018.  [ USER_NAME <> '' or = BLOCK_NAME(CURRENT_NEST_LEVEL) ] and
15019.  BLOCK_TYPE(CURRENT_NEST_LEVEL) = 'IF', then
15020.  ERROR_OCCURRED will be set false and CURRENT_NEST_LEVEL will be
15021.  unmodified.) */
15022.  **if** ERROR_OCCURRED
15023.  **then**
15024.  **mexit**
15025.  **fi**
15026.  **if** OPERAND1(CURRENT_NEST_LEVEL) ''
15027.  **then**
15028.  **mnote** (8, 'ELSE BLOCK "' OPERAND1(CURRENT_NEST_LEVEL)
15029.  " NOT FOUND')
15030.  **fi**
15031.  **call** POP_OLD_BLOCK ( OPERAND3(CURRENT_NEST_LEVEL))
15032.  /* Delete current block, generating END and XII labels as required,
15033.  and popping stack. (Lemma: Execution of POP_OLD_BLOCK always
15034.  results in decrementing of CURRENT_NEST_LEVEL by exactly 1.) */
15035.  **end**
15036.  /* (Lemma: IF CURRENT_NEST_LEVEL > 0 and
15037.  [ USER_NAME <> '' or = BLOCK_NAME(CURRENT_NEST_LEVEL) ] and
15038.  BLOCK_TYPE(CURRENT_NEST_LEVEL) = 'IF' at entry to FI, then
15039.  CURRENT_NEST_LEVEL will be decremented by exactly 1.) */
DO Macro -- 21 June 1973

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 $nnnLB.
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. */
21075. call TRACE_PRINTER (: "DO")
21076. /* Prints macro name "DO" in mnote if tracing on. */
21077. call PUSH_NEW_BLOCK(USER_NAME; BLOCK_TYPE_VALUE='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. exit
21084. fi
21085. if REL '=' OR MASK = ' '
21086. then
21087. mnote (8, "STRC2113 REL= OR MASK= NOT IN PARENTHESES—IGNORED")
21088. fi
21089. LABEL := USER_NAME
21090. call DO_SCAN_OPERANDS
21091. /* Collect scanning information and looping branch (BCT, BXH, and
21092. BXLE) information from operands. Set OPERAND_FORMAT based on
21093. these values. */
21094. if OPERAND_FORMAT # 0 and F 10
21095. # 12 and # 19
21096. then
21097. END_LABEL_REQD(CURRENT_NEST_LEVEL) := true
21098. fi
21099. if DEBUG_BLOCKCOUNTS_REQD
21100. then /* Generate reset of current loop count. */
21101. generate (LABEL II ' SR 1,1)
21102. LABEL := ' '
21103. generate (" STM 1,' | BLOCK_LABEL_PREFIX | 'DOL')
21104. fi
21105. if OPERAND_FORMAT # 0
21106. then /* Not infinite loop. */
21107. call DO_BRANCH_FOR_LOOP_ENTRY
21108. /* Generate flow point 0 branch, if required, to proper label to insure
21109. UNTIL tests are not made before first loop. */
21110. call DO_GENERATE_ALL_CONDITIONAL_TESTS
21111. /* Cause WHILE and UNTIL tests to be generated (flow points 1 through
21112. 6) with proper labels. */
21113. fi
21114. call DO_LABEL_BLOCK
21115. /* Store begin label (flow point 7) into 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 II ' DS 0H')
21124. fi
21125. mend
PROC DO_SCAN_OPERANDS

/* Collect WHILE_INDEX, UNTIL_INDEX, UNTIL_END_INDEX, the corresponding keywords include a conditional test to be generated; set looping branch information (LOOPING_BRANCH_TYPE, UNTIL_INDEX, UNTIL_END_INDEX, 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_ENDINDEXES AND MAIN_OP

/* For each type xxxx (WHILE and UNTIL) which is present, put index of the last operand of looping group which belongs to 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_INDEX 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 IB_LOGIC_OP to make decision. */

END
DO_FIND_KEYNORDS AND PRESENCE

21160. ENSU TO FIND OPERANDS OF 'WHILE' AND 'UNTIL' KEYS; SET TO ZERO
21161. /* Find operand index of 'WHILE' and 'UNTIL' keys; set to zero
21162. if omitted. Put index of last valid operand in LASTOP.
21163. Set WHILE_PRESENT and UNTIL_PRESENT TRUE if corresponding looping
21164. group is present, else set to FALSE. */

21165. WHILE_INDEX, UNTIL_INDEX := 0
21166. if SYSLIST(1) = 'WHILE' and 'UNTIL'
21167. then /* First operand must be "WHILE", "UNTIL", "FOREVER", OR OMITTED */
21168. if WHILE_INDEX > 0 then /* No WHILE found before */
21169. WHILE_INDEX := I
21170. else / Operand was not "WHILE". */
21171. UNTIL_INDEX := I - 1
21172. fi
21173. if UNTIL_INDEX > 0 then /* No UNTIL found before */
21174. UNTIL_INDEX := I
21175. else /* Operands after second "UNTIL" ignored */
21176. if SYSLIST(1) = 'UNTIL'
21177. then /* Unknown operand. */
21178. if WHILE_INDEX = 0 then /* No WHILE found before */
21179. WHILE_INDEX := I
21180. else /* Operands after second "UNTIL" ignored */
21181. UNTIL_INDEX := I - 1
21182. fi
21183. if UNTIL_INDEX = 0 then /* No UNTIL found before */
21184. UNTIL_INDEX := I
21185. else /* Operand was not "UNTIL". */
21186. if SYSLIST(1) = 'WHILE'
21187. then /* First operand must be "WHILE", "UNTIL", "FOREVER", OR OMITTED */
21188. if WHILE_INDEX > 1 then /* No WHILE found before */
21189. WHILE_INDEX := I
21190. else /* Operands after second "UNTIL" ignored */
21191. UNTIL_INDEX := I - 1
21192. fi
21193. if UNTIL_INDEX > 1 then /* No UNTIL found before */
21194. UNTIL_INDEX := I
21195. else /* Operands after second "UNTIL" ignored */
21196. fi
21197. fi
21198. if WHILE_INDEX > 1 then /* No WHILE found before */
21199. WHILE_INDEX := I
21200. else /* Operands after second "UNTIL" ignored */
21201. fi
21202. if UNTIL_INDEX > 1 then /* No UNTIL found before */
21203. UNTIL_INDEX := I
21204. else /* Operands after second "UNTIL" ignored */
21205. fi
21206. /* Decide whether WHILE and UNTIL looping groups are present. The
21207. possible operand formats are:
21208. DO UNTIL,looping-group
21209. DO WHILE,looping-group
21210. DO WHILE,looping-group,<and/or>,UNTIL,looping-group
21211. DO WHILE,looping-group,<and/or>,WHILE,looping-group
21212. DO [No operand or single operand "FOREVER"]
21213. means infinite loop. */
21214. UNTIL_PRESENT := (UNTIL_INDEX > 0)
21215. WHILE_PRESENT := (WHILE_INDEX > 0) OR
21216. (UNTIL_INDEX > 1 OR
21217. (~ UNTIL_PRESENT) AND LASTOP > 0)
21218. /* Last two alternatives are only to fix up when WHILE was
21219. omitted. */
21220. END
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 = 1 and SYSLIST(I) = 'FOREVER'
21233.  then WHILE_PRESENT, UNTIL_PRESENT := false
21234.  else if WHILE_PRESENT
21235.  then
21236.    AT UNTIL_PRESENT
21237.  then
21238.    I := WHILE_INDEX - 1
21239.  fi
21240.  if UNTIL_INDEX < WHILE_INDEX
21241.  then /* UNTIL is first: "DO UNTIL,"<test>,<and/or>,WHILE,"<test>" */
21242.    WHILE_END_INDEX := I - 1 /* Point at end of WHILE. */
21243.    if SYSLIST(I) # "AND" and # "OR"
21244.      THEN END_INDEX := I /* Error message will be printed later. */
21245.    fi
21246.  else /* WHILE is first: "DO WHILE,"<test>,<and/or>,UNTIL,"<test>" */
21247.    I := UNTIL_INDEX - 1 /* Point at <and/or>, */
21248.    WHILE_END_INDEX := I
21249.    if SYSLIST(I) # "AND" and # "OR"
21250.      THEN END_INDEX := I /* Error message will be printed later. */
21251.    fi
21252.  fi
21253.  if WHILE_INDEX = WHILE_END_INDEX
21254.  then /* One of the following was entered: */
21255.    "DO WHILE,"<test>,"<and/or>,WHILE,"<test>" */
21256.    if SYSLIST(I) = 'CR'
21257.      THEN MAIN_OP := 'OR'
21258.    fi
21259.    MAIN_OP := 'AND' /* Assumed. */
21260.    if SYSLIST(I) = 'OR'
21261.      THEN MAIN_OP := 'OR'
21262.    fi
21263.  else
21264.    if WHILE_PRESENT and UNTIL_PRESENT
21265.      THEN /* "DO UNTIL,"<test>,"<and/or>,UNTIL,"<test>" */
21266.        MAIN_OP := 'AND' /* Assumed. */
21267.      else if SYSLIST(I) = 'OR'
21268.        THEN MAIN_OP := 'OR'
21269.      fi
21270.      if WHILE_PRESENT and UNTIL_PRESENT
21271.        THEN /* "DO UNTIL,"<test>,"<and/or>,UNTIL,"<test>" */
21272.          MAIN_OP := 'AND' /* Assumed. */
21273.      else if SYSLIST(I) = 'OR'
21274.        THEN MAIN_OP := 'OR'
21275.      fi
21276.    fi
21277.    if WHILE_INDEX = WHILE_END_INDEX /* Which is equal to LASTOP. */
21278.      THEN /* "DO WHILE" with no other operands. */
21279.    fi
21280.    if UNTIL_INDEX = UNTIL_END_INDEX /* One of the following was entered: */
21281.      "DO UNTIL,"<test>,"<and/or>,UNTIL,"<test>" */
21282.      if SYSLIST(I) = 'CR'
21283.        THEN MAIN_OP := 'OR'
21284.      fi
21285.      MAIN_OP := 'AND' /* Assumed. */
21286.      if SYSLIST(I) = 'OR'
21287.        THEN MAIN_OP := 'OR'
21288.      fi
21289.    fi
21290.    if WHILE_INDEX = WHILE_END_INDEX
21291.      THEN /* "DO WHILE,"<test>,"<and/or>,UNTIL,"<test>" */
21292.        MAIN_OP := 'AND' /* Assumed. */
21293.      else if SYSLIST(I) = 'OR'
21294.        THEN MAIN_OP := 'OR'
21295.      fi
21296.    fi
21297.    if UNTIL_INDEX = UNTIL_END_INDEX
21298.      THEN /* "DO UNTIL,"<test>,"<and/or>,UNTIL,"<test>" */
21299.        MAIN_OP := 'AND' /* Assumed. */
21300.    fi

C-16
LB := 0
LOOPING_BRANCHTYPE := 'NCNE'
if UNTILPRESENT then
  I, UNTIL_INDEX := UNTIL_INDEX + 1
  /* Move UNTIL_INDEX from pointing at "UNTIL" to pointing at first
     UNTIL operand. */
  if SYSLIST(I,1) = 'BCT' or = 'BXLE' or = 'BXH' then
    LOOPING_BRANCHTYPE := 'UNTIL'
    UNTIL_CONDTEST := (UNTIL_END_INDEX > I)
    LB := I
    /* UNTIL_INDEX is still pointing at the looping branch; we aren't
     sure how far to advance it yet. */
    else
      UNTIL_COND_TEST := false
      /* Turn off all UNTIL stuff. */
    fi
  fi
  if WHILE_PRESENT then
    I, WHILE_INDEX := WHILE_INDEX + 1
    /* Move WHILE_INDEX from pointing at "WHILE" to point at first
       WHILE operand. */
    if SYSLIST(I,1) = 'BCT' or = 'BXLE' or = 'BXH' then
      LOOPING_BRANCHTYPE := 'WHILE'
      WHILE_CONDTEST := (WHILE_END_INDEX > I)
      while CONDITION is still pointing at looping branch; we aren't sure
      how far to advance it yet. */
      if SYSLIST(I,1) = 'BCT'
        then
          mnote (4, "STC2103 WARNING--"WHILE,(BCT,..."
          ELSE\"LESS TIME THAN VALUE IN REGISTER"")
          fi
          else /* There is also an UNTIL looping branch. */
            mnote (4, "STC2105 TWO LOOPING BRANCHES INVALID IN "DO"
            ELSE\"WHILE IGNORED")
            WHILE_PRESENT, WHILE_COND_TEST := false
            WHILE_INDEX, WHILE_ENDINDEX := 0
      fi
    else
      WHILE_COND_TEST := true
    fi
  else
    WHILE_INDEX, WHILE_ENDINDEX := 0
    WHILE_COND_TEST := false
  fi
endif

DO Macro -- 21 June 1973

21374. PROC DO_LOOPING_BRANCH_PROCESS
21375. /* Collect looping branch information:
21376.    LB_OPCODE_ID  'BCT', 'GILZ' or 'BRK'
21377.    LB_OPERAND1 First operand, null (or garbage) for BCT
21378.    LB_OPERAND2 Second operand, null (or garbage) for BCT
21379.    LB_LOGIC_OP Logic operand connecting looping branch to
21380.        rest of WHILE or UNTIL.
21381.    LB_LABEL_REQ Indicates whether looping branch will need
21382.        a label.
21383.    Also step WHILE or UNTIL_INDEX over looping branch. If any looping
21384.        branch is present, LB contains it's index; else, LB = 0. */
21385.    INT OP_COUNT /* Number of operands looping branch needs. */
21386.    LB_OPCODE_ID, LB_OPERAND1, LB_OPERAND2, LB_LOGIC_OP := '1'
21387. /* Assume no looping branch is present. */
21388.    IF LB # 0 THEN
21389. 21390.    LB_OPCODE_ID := SYSLIST(LB,1)
21391.    IF LB_OPCODE_ID = 'BCT'
21392.        LB_OPCODE_ID := SYSLIST(LB,2)
21393.    ELSE
21394.        OP_COUNT := 2
21395.    ENDIF
21396.    LB_OPCODE_ID := SYSLIST(LB,3)
21397.    IF SysList(LB) # 1
21398.        IF SysList(LB) = 1 THEN
21399.            LOOPING_BRANCH_TYPE := WHILE
21400.        ENDIF
21401.    ELSE
21402.        LOOPING_BRANCH_TYPE := UNTIL
21403.    ENDIF
21404.    IF LB_OPCODE_ID = 'BCT'
21405.        LB_OPCODE_ID := SYSLIST(LB,1)
21406.        IF SysList(LB) = 1 THEN
21407.            LOOPING_BRANCH_TYPE := WHILE
21408.        ENDIF
21409.    ELSE
21410.        LOOPING_BRANCH_TYPE := UNTIL
21411.    ENDIF
21412.    IF LOOPING_BRANCH_TYPE = 'WHILE' or LOOPING_BRANCH_TYPE = 'UNTIL'
21413.        LB_OPCODE_ID := SYSLIST(LB,2)
21414.        OP_COUNT := 2
21415.    ENDIF
21416.    IF LB_OPCODE_ID = 'GILZ'
21417.        LB_OPCODE_ID := SYSLIST(LB,1)
21418.        IF SysList(LB) = 1 THEN
21419.            LOOPING_BRANCH_TYPE := 'WHILE'
21420.        ENDIF
21421.    ELSE
21422.        LOOPING_BRANCH_TYPE := 'UNTIL'
21423.    ENDIF
21424.    IF LOOPING_BRANCH_TYPE = 'WHILE'
21425.        WHILE_INDEX := LB
21426.    ELSE
21427.        UNTIL_INDEX := LB
21428.    ENDIF
21429.    IF LOOPING_BRANCH_TYPE = 'WHILE'
21430.        WHILE_INDEX := LB
21431.    ELSE
21432.        UNTIL_INDEX := LB
21433.    ENDIF
21434.    LB_LABEL_REQ := (LOOPING_BRANCH_TYPE = 'WHILE')
21435.    COMP

C-18
DO Macro -- 21 June 1973

```plaintext
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.                           ELSE
21455.                           OPERAND_FORMAT := 13
21456.                         ELSE
21457.                         OPERAND_FORMAT := 14
21458.                       ELSE
21459.                       OPERAND_FORMAT := 11
21460.                     ELSE
21461.                     OPERAND_FORMAT := 10
21462.                   ELSE
21463.                   OPERAND_FORMAT := 12
21464.                 ELSE
21465.                 OPERAND_FORMAT := 13
21466.               ELSE
21467.               OPERAND_FORMAT := 18
21468.             ELSE
21469.             OPERAND_FORMAT := 17
21470.         ELSE
21471.         OPERAND_FORMAT := 13
21472.       ELSE
21473.       OPERAND_FORMAT := 10
21474.   ELSE
21475.   OPERAND_FORMAT := 12
21476. fi
21477. fi
```

C-19
/* if WHILE_PRESENT then */
if LOOPING_BRANCH_TYPE = 'WHILE' then *** */
else if LOOPING_BRANCH_TYPE = 'UNTIL'
then
  if UNTIL_COND_TEST then
    if LD_LOGIC_OP = 'AND'
    then
      if MAIN_OP = 'AND'
      then
        OPERAND_FORMAT := 7
      else
        OPERAND_FORMAT := 9
      fi
    else
      if MAIN_OP = 'AND'
      then
        OPERAND_FORMAT := 6
      else
        OPERAND_FORMAT := 8
      fi
    fi
  else
    fi
fi
else
  if MAIN_OP = 'AND'
  then
    OPERAND_FORMAT := 2
  else
    OPERAND_FORMAT := 3
  fi
else
  OPERAND_FORMAT := 1
fi
fi
fi
fi
fi
/* if WHILE_PRESENT then */
else
   if UNTIL_PRESENT
      then
         if LOOPING_BRANCH_TYPE = 'UNTIL'
            then
               if UNTIL_COND_TEST
                  then
                     if LB_LGHIC_OP = 'AND'
                        then
                           OPERAND_FORMAT := 22
                        else
                           OPERAND_FORMAT := 21
                        fi
                    else
                        OPERAND_FORMAT := 19
                    fi
                else
                    OPERAND_FORMAT := 20
                fi
            else
                OPERAND_FORMAT := 0
            fi
         else
            Operand_FORMAT := 0
         fi
      else
        Corp
     fi
"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. desc OPERAND_FORMAT ifany
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. esac
21571. case (10-18)
21572. */ Branch to WHILE looping branch first. */
21573. generate (LABEL || 'B' || BLOCK_LABEL_PREFIX || 'LBP')
21574. LABEL :=
21575. esac
21576. case (20,21,22)
21577. */ Branch around UNTIL conditional test to DO internal code. */
21578. generate (LABEL || 'B' || BLOCK_LABEL_PREFIX || 'REG')
21579. LABEL :=
21580. esac
21581. esac
21582. code
21584. PROC DO_GENERATE_ALL_CONDITIONAL_TESTS
21585. /* Cause WHILE and UNTIL conditional tests to be generated with proper
21586. labels. */
21587. int PASS /* Looping index. */
21590. if PASS = 1
21591. then
21592. call DO UNTIL_PREPROCESS
21593. else
21594. call DO WHILE_PREPROCESS
21595. fi
21596. /* The DO_xxxx_PREPROCESS proc must set:
21597. THISCONDITIONAL_REQD
21598. FIRST_INDEX from xxxx_INDEX
21599. LAST_INDEX from xxxx_END_INDEX
21600. UNIQUE_LABEL_ID with the first letter of xxxx
21601. ULTIMATE_BRANCH_LABEL with the branch target
21602. ULTIMATE_FALLTHRU_LABEL with the fallthru target
21603. ULTIMATE_FALLTHRU_CONDITION with the proper value
21604. FIRST_ID with the first label
21605. to insure proper test generation. */
21606. if THIS_CONDITIONAL_REQD
21607. then
21608. call DO_GENERATE_CONDITIONAL_SET
21609. /* Generate code to pass control to the ULTIMATE_FALLTHRU_LABEL (or
21610. to fall through to it) if the conditional test specified by
21611. SYSLIST(FIRST_INDEX) through SYSLIST(LAST_INDEX) has the logical
21612. value stored in ULTIMATE_FALLTHRU_CONDITION; else pass control to
21613. the ULTIMATE_BRANCH_LABEL. If a branch is generated to the
21614. ULTIMATE_FALLTHRU_LABEL, set FALLTHRU_LABEL_USED to true;
21615. else set it false. Include definition of any LABEL outstanding
21616. before generating code. */
21617. if PASS = 1
21618. then
21619. call DO UNTIL_PCSTPROCESS
21620. /* For those cases where the ULTIMATE_FALLTHRU_LABEL was not to
21621. follow the conditional test as the next sequential instruction,
21622. generate an unconditional branch to the ULTIMATE_FALLTHRU_LABEL
21623. and clear FALLTHRU_LABEL_USED. */
21624. if FALLTHRU_LABEL_USED
21625. then
21626. LABEL := ULTIMATE_FALLTHRU_LABEL
21627. /* Generate label at next opportunity. */
21628. fi
21629. fi
21630. else
21631. fi
21632. PASS := PASS + 1
21633. od /* (Termination: PASS incremented only (not modifiable by called
21634. proc), must eventually exceed 2.) */
DO Macro -- 21 June 1973

21632. DO DO UNTIL PREPROCESS
21633.    /* Must set up THIS_CONDITIONAL_REQD, FIRST_INDEX, LAST_INDEX,
21634.    UNIQUE_LABEL_ID, ULTIMATE_BRANCH_LABEL, ULTIMATE_FALLTHRU_LABEL,
21635.    ULTIMATE_FALLTHRU_CONDITION, and FIRST_ID. */
21636.    THIS_CONDITIONAL_REQD := UNTIL_COND_TEST
21637.    if UNTIL_COND_TEST
21638.    then
21639.    FIRST_INDEX := UNTIL_INDEX
21640.    LAST_INDEX := UNTIL_END_INDEX
21641.    ULTIMATE_BRANCH_LABEL := BLOCK_LABEL_PREFIX || 'BEG'
21642.    /* Flow point 2 normally connects to flow point 12. */
21643.    ULTIMATE_FALLTHRU_LABEL := BLOCK_LABEL_PREFIX || 'W1'
21644.    /* Flow point 3 usually falls through to flow point 4. */
21645.    ULTIMATE_FALLTHRU_CONDITION := false
21646.    UNIQUE_LABEL_ID := 'U'
21647.    FIRST_ID := BLOCK_LABEL_PREFIX || 'U1'
21648.    do case OPERAND_FORMAT |type
21649.     of
21650.     CASE (3, 6, 9) /* UNTIL test ORed with WHILE test. */
21651.           ULTIMATE_FALLTHRU_CONDITION := false
21652.           ULTIMATE_BRANCH_LABEL := BLOCK_LABEL_PREFIX || 'BEG'
21653.     CASE (11, 15, 17) /* UNTIL test ANDed with WHILE looping branch. */
21654.           ULTIMATE_BRANCH_LABEL := BLOCK_LABEL_PREFIX || 'LB'
21655.     CASE (12, 16, 18) /* UNTIL test ORed with WHILE looping branch. */
21656.           ULTIMATE_BRANCH_LABEL := BLOCK_LABEL_PREFIX || 'LB'
21657.     CASE (20, 21, 22) /* UNTIL conditional test only. */
21658.           ULTIMATE_FALLTHRU_LABEL := BLOCK_LABEL_PREFIX || 'BEG'
21659.     CASE (23) |type
21660.     END
DO_WHILE_PROCESS

/* Must set up THIS_CONDITIONAL_REQD, FIRST_INDEX, LAST_INDEX,
   UNIQUE_LABEL_ID, ULTIMATE_BRANCH_LABEL, ULTIMATE_FALLTHRU_LABEL,
   ULTIMATE_FALLTHRU_CONDITION, and FIRST_ID. */

THIS_CONDITIONAL_REQD := WHILE_COND_TEST
if WHILE_COND_TEST
   then
      FIRST_INDEX := WHILE_INDEX
      LAST_INDEX := WHILE_END_INDEX
      ULTIMATE_BRANCH_LABEL := BLOCK_LABEL_PREFIX || 'END'
   /* Flow point 5 always branches to flow point 12. */
      ULTIMATE_FALLTHRU_LABEL := BLOCK_LABEL_PREFIX || 'BEG'
   /* Flow point 6 always falls through to flow point 7. */
      ULTIMATE_FALLTHRU_CONDITION := true
   UNIQUE_LABEL_ID := 'W'
   FIRST_ID := BLOCK_LABEL_PREFIX || 'W1'
fi
end
21695. "DC" Macro -- 21 June 1973

21696. PROC DO_GENERATE_CONDITIONAL_SET
21697. /* Generate code to pass control to the ULTIMATE_FALLTHRU_LABEL (or
21698. to fall through to it) if the conditional test specified by
21699. SYSLIST(FIRST_INDEX) through SYSLIST(LAST_INDEX) has the
21700. logical value which is stored in ULTIMATE_FALLTHRU_CONDITION;
21701. else to pass control to the ULTIMATE_BRANCH_LABEL. Also
21702. see that FALLTHRU_LABEL_USED is set to true if branch to
21703. 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. LABEL := DS
21714. generate (LABEL | | DS OH)
21715. /* Get the LABEL label out of the way. */
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. */

C-25
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
21750. esacod
21751. esacod
21752. core
"DO" Macro -- 21 June 1973

21754. PROC DO_LABEL BLOCK
21755. /* If a begin label is required, generate it. */

21757. DOCASE OPERAND FORMAT LABEL
21758. OF
21759. CASE (0,3,5,8-12,14,16-22)
21760. IF LABEL # BLOCK_LABEL_PREFIX || 'BEG'
21761. THEN
21762. /* Begin label must be generated. */
21763. IF LABEL # ''
21764. THEN
21765. LABEL GENERAL
21766. FI
21767. LABEL := BLOCK_LABEL_PREFIX || 'BEG'
21768. FI
21769. ENDCASE
21770. ENDS
21771. CODE
21772. PROC DC_INFO_SAVE
21773. /* Insert into stack all information required to close loop at
21774.   TERMINATE_DO_LOOP. */
21775. case B0, B10, B11
21776.   /* One character codes indicating flow point to follow points 8,
21777.   10, and 11. */
21778. B8 := 'W' /* Assume branch at point 8 is to WHILE group (point 4). */
21779. B10, B11 := '0'
21780. /* Assume no looping branch (thus no branches at 10 and 11). */
21781. /* Set B8. */
21782. docase OPERAND_FORMAT only
21783.   of
21784.   case (2,3,11,16-18,20)
21785.     B8 := 'G' /* To flow point 1. */
21786.   esac
21787.   case (4-10,13,14,19,21,22)
21788.     B8 := 'T' /* To flow point 9. */
21789.   esac
21790. case (0) /* Infinite loop. */
21791.     B8 := '0' /* To flow point 7. */
21792. esac
21793. esac
21794. if LOO ging_BRANCH_TYPE == 'NOW'
21795. then
21796. /* Set B10. */
21797. docase OPERAND_FORMAT only
21798. of
21799. case (4,7,13,15,16)
21800.     B10 := 'W' /* To flow point 4. */
21801. esac
21802. case (5-9,12,14,17,19,22)
21803.     B10 := 'B' /* To flow point 7. */
21804. esac
21805. case (6,8,21)
21806.     B10 := 'U' /* To flow point 1. */
21807. esac
21808. esac
21809. /* Set B11. */
21810. docase OPERAND_FORMAT only
21811. of
21812. case (4,6,10,13,15,16,19,21)
21813.     B11 := 'W' /* Fall through to flow point 12 (end of DO block). */
21814. esac
21815. case (5,8,14,17,18)
21816.     B11 := 'B' /* To flow point 8. */
21817. esac
21818. case (7,9,22)
21819.     B11 := 'U' /* To flow point 1. */
21820. esac
21821. esac
21822. /* INFORMATION(CURRENT_N EST_LEVEL) := B8 || B10 || B11 || LD_LABEL_REQ ||
21823.   false false false false false false
21824.   /* Byte 5 is set true when the loop is terminated (by ATEND, ONEXIT,
21825.   or 00). */
21826. Byte 6 is set true when an ATEND occurs for this DC. */
21827. Byte 7 is set true when an ONEXIT occurs for this DO.
21828. Byte 8 is set true if a FIN label is required in the DO code. */
21829. OPERAND2(CURRENT_N EST_LEVEL) := LD_OPERAND2
21830. OPERAND3(CURRENT_N EST_LEVEL) := LD_OPCODE_ID
21831. OPERAND3(CURRENT_N EST_LEVEL) := LD_OPCODE_ID
21832. OPERAND3(CURRENT_N EST_LEVEL) := LD_OPCODE_ID
"DOM" Macro -- 21 June 1973

1840. PROC DO_TRACE_COUNTERS
1841.     /* If debugging, generate block name and/or counters for block and
1842.     loop execution. */
1843.
1844.     if DEBUG_BLOCKCOUNTS_REQD or DEBUG_BLOCKNAMES_REQD
1845.     then
1846.         if DEBUG_BLOCKCOUNTS_REQD
1847.             then
1848.                 generate (LABEL || ' LB ' || BLOCK_LABEL_PREFIX || 'DOL')
1849.                 LABEL := 'LB
1850.                 generate ('' LA 1,1(1))
1851.                 generate ('' 2H 1,' || BLOCK_LABEL_PREFIX || 'DOL')
1852.                 generate ('' LA 1,1(1))
1853.                 generate ('' 3H 1,' || BLOCK_LABEL_PREFIX || 'DTM')
1854.                 generate ('' LA 1,1(1))
1855.             fi
1856.     /* Generate branch around block name and/or block counts. */
1857.     generate (LABEL || ' B ' || BLOCK_LABEL_PREFIX || 'GO')
1858.     LABEL := BLOCK_LABEL_PREFIX || 'GO'
1859.
1860.     if DEBUG_BLOCKNAMES_REQD
1861.         then
1862.             generate ('"", 0H')
1863.         fi
1864.     if DEBUG_BLOCKCOUNTS_REQD
1865.         then
1866.             generate (BLOCK_LABEL_PREFIX || "DOL DC H'0' CURRENT LOOP COUNT")
1867.             generate (BLOCK_LABEL_PREFIX || "DTM DC H'0' OVERALL LOOP COUNT")
1868.         fi
1869.     fi
1870.   corp

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

C-30
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 upon normal loop termination. If the ATEND has been preceded by an ONEXIT macro, the branch is generate to the OD for the ONEXIT block. 

\[\text{\texttt{ATEND\_GENNED}}\] /* Indicates whether ATEND has been generated previously for this block. */

\[\text{\texttt{TDL\_GENNED}}\] /* Indicates whether the TERMINATE\_DO\_LOOP macro has been evoked for this DO by a previous macro (properly, only by an ONEXIT). */

\[\text{\texttt{FIN\_LABEL\_REQD}}\] /* Indicates a branch to the label "FIN\_PIN" has been generated and must be defined at OD time. */

\[\text{\texttt{INFO}}\] /* Holds copy of INFORMATION(CURRENT\_NEST\_LEVEL). */

\[\text{\texttt{VERIFY\_END}}\] ( ; '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. */

\[\text{\texttt{VERIFY\_END}}\] ( ; '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. */

\[\text{\texttt{TERMINATE\_DO\_LOOP}}\] ( ) /* Terminate the loop by generating any necessary back branches. */

\[\text{\texttt{TERMINATE\_DO\_LOOP}}\] ( ) /* Terminate the loop by generating any necessary back branches. */
```
"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 abnoraml loop termination (EXIT macros) is then defined
25005. to allow the code which follows to be executed upon abnoraml 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_GENERED, /* Indicates whether ONEXIT has been generated
25010. previously for this block. */
25011. bit TDL_GENERED, /* 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. bit FIN_LABEL_REQD /* Indicates a branch to the label "$snapFIN" has
25015. been generated and must be defined at OD time. */
25016. char INFO /* Holds copy of INFORMATION(CURRENT_NEST_LEVEL). */
25017.
25018. call TRACE_PRINTER ( ; 'ONEXIT') /* Prints macro name "ONEXIT" in mnote if tracing on. */
25019.
25020. if CURRENT_NEST_LEVEL > NESTING_LIMIT then
25021. mexit
25022. fi
25023.
25024. call VERIFY_END ( ; 'DO' USER_NAME) /* Verifies current block has the name specified
25025. by the USER_NAME operand of the ONEXIT macro (if any) and that it is a DO block.
25026. Various errors receive messages and either intermediate blocks are
25027. BLENDed as a fixup or ERROR_OCCURRED is set. */
25028.
25029. if ERROR_OCCURRED then
25030. mexit
25031. fi
25032.
25033. INFO := INFORMATION(CURRENT_NEST_LEVEL)
25034.
25035. ONEXIT_GENERED := INFO[7,1] /* See if we've already generated an ONEXIT. */
25036.
25037. if ONEXIT_GENERED then
25038. note (8, 'SRC2501 MORE THAN ONE "ONEXIT" IN BLOCK')
25039. mexit
25040. fi
25041.
25042. if -EXIT_LABEL_REQD(CURRENT_NEST_LEVEL) then
25043. note (8, 'SRC2502 NO EXIT FOR THIS "DO"')
25044. mexit
25045. fi
25046.
25047. FIN_LABEL_REQD := INFO[8,1] /* Note whether a FIN label has already been referenced. */
25048.
25049. if TDL_GENERED then
25050. note (8, 'SRC2503 FIN LABEL NEEDED')
25051. mexit
25052. fi
25053.
25054. call TERMINATE_DO_LOOP ( ;
25055. if TDL_FALLTHRU_OCCURS then /* Looping branch expects to fall through to END label. */
25056. generate ('B' | BLOCK_LABEL_PREFIX | 'END')
25057. fi
25058. else /* TERMINATE_DO_LOOP must have been done by previous ATEND. */
25059. generate ('B' | BLOCK_LABEL_PREFIX | 'FIN')
25060. FIN_LABEL_REQD := true
25061. fi
25062. /* Provide branch to FIN for ATEND block. */
25063.
25064. generate (BLOCK_LABEL_PREFIX | 'EXIT DS OD')
25065.
25066. FIN_LABEL_REQD(CURRENT_NEST_LEVEL) := false
25067. /* Provide target for EXIT branch and note that it is no longer
25068. needed. */
25069.
25070. INFORMATION(CURRENT_NEST_LEVEL) := INFO[1,43] /* Forward FIN_LABEL_REQD to OD. */
25071.
25072. endif
25073.
```
```
27001. macro OD ( ; USER_NAME)
27002. /* Terminate DO loop if ATEND or ONEXIT have not done so and do
27003. standard block closing. */
27004. ret
27005. /* Indicates whether the looping code has been generated
27006. yet. */
27007. TDL_GENNED
27008. /* Indicates whether a "$nnnFIN" label is
27009. required. (It is used at the end of the ATEND or ONEXIT code.) */
27010. if
27011. INFO /* Holds a copy of INFORMATION(CURRENT_NEST_LEVEL). */
27012. call TRACE_PRINTER ( ; "OD") /* Prints macro name "OD" on note if tracing on. */
27013. if CURRENT_NEST_LEVEL < NESTING_LIMIT
27014. then
27015.
27016. call VERIFY_END ( ; 'DO', USER_NAME) /* Verifies current block has the name specified by the USER_NAME
27017. operand of the OD macro (if any) and that it is a DO block.
27018. Various errors receive messages and either intermediate blocks are
27019. BLENDED as a fixup or ERROR_OCCURRED is set.
27020. (Lemma: If CURRENT_NEST_LEVEL > 0 and
27021. [USER_NAME = '*' or = BLOCK_NAME(CURRENT_NEST_LEVEL)] and
27022. BLOCK_TYPE(CURRENT_NEST_LEVEL) = 'DO', then
27023. ERROR_OCCURRED will be set false and CURRENT_NEST_LEVEL will
27024. not be modified.) */
27025. if ERROR_OCCURRED
27026. then
27027. mexit
27028. fi
27029.
27030. INFO := INFORMATION(CURRENT_NEST_LEVEL)
27031. TDL_GENNED := INFO[5,1]
27032. FIN_LABEL_REQD := INFO[8,1]
27033. if ~ TDL_GENNED
27034. then
27035. call TERMINATE_DO_LOOP ( ; ) /* Call separate macro to generate loop-terminating branches.
27036. (Lemma: TERMINATE_DO_LOOP does not modify CURRENT_NEST_LEVEL.) */
27037. else /* ATEND or ONEXIT occurred; we may need FIN label. */
27038. if FIN_LABEL_REQD
27039. then
27040. Generate ("$" | | BLOCK_NUMBER(CURRENT_NEST_LEVEL) | | 'FIN DS OH')
27041. fi
27042. fi
27043. fi
27044.
27045. call POP_OLD_BLOCK ( ;)
27046. /* Delete current block from the stack.
27047. (Lemma: POP_OLD_BLOCK decrements CURRENT_NEST_LEVEL by exactly
27048. one.) */
27049. send
27050. /* (Lemma: If CURRENT_NEST_LEVEL > 0 and
27051. [USER_NAME = '*' or = BLOCK_NAME(CURRENT_NEST_LEVEL)] and
27052. BLOCK_NAME(CURRENT_NEST_LEVEL) = 'DO' at entry to OD, then
27053. CURRENT_NEST_LEVEL will be decremented by exactly one.) */
```
The DOCASE macro is used to select one of its immediate subblocks defined by CASE macros for execution. The operands are scanned to determine the type of case specification provided. Depending on the format indicated, some instructions may be generated at this time and various data are stored in the stack to direct code generation at the CASE and ESACCD macros.

```
macro DOCASE (USER_NAME; INDEX, OPTION, RANGE)
/* The DOCASE macro is used to select one of its immediate subblocks defined by CASE macros for execution. The operands are scanned to determine the type of case specification provided. Depending on the format indicated, some instructions may be generated at this time and various data are stored in the stack to direct code generation at the CASE and ESACCD macros. */

BIT
BRANCH_TO_CASE1, /* Initially false: to be set true at any time a branch is generated which would require the first CASE to be labeled (as opposed to falling through to the first CASE). */
INDEX_RANGE_ASSURED /* Set to true if "ONLY" option is specified to indicate index will take on only values represented by the following CASE blocks. */

CHAR
INDEX_REG, /* Name of register containing DOCASE index, if in a register. */
INDEX_LENGTH, /* Length (or symbol indicating length) of index for CHARCOMP operand. */
INDEX_TYPE, /* Type of DOCASE index: "R" register, "W" word (or no index—CONDTEST type DOCASE), "H" halfword, or "B" byte (or character string—CHARCOMP type DOCASE). */
CASE_FORMAT, /* Format of CASE macros to follow: "GENERAL" (branch vector and/or symbolic compares with index), "SPARSE" (symbolic compares only), "CHARCOMP" (character string compares), "SIMPLE" (short sequence of integers in order 1, 2, 3, *--), or "CONDTEST" (no index on DOCASE, conditional test on each CASE macro). */
LABEL, /* Any outstanding label, to be generated on next executable instruction. */
INDEX_ADDR /* Symbolic address of byte-type or CHARCOMP index, if any; else null. */

/* Ground rules: LABEL is to be generated on the first of any executable instruction sequence and then cleared to null; any label which needs to be so generated may replace a null LABEL. BRANCH_TO
CASE must be set by any branch directly or indirectly to the first CASE (i.e., by allbut falling through to the first CASE). */
```
"DOCASE" Macro -- 26 June 1973

31040. call TRACE_PRINTER ( ; 'DOCASE')
31041. /* Print macro name "DOCASE" in mnote if tracing on. */
31042. call PUSH_NEW_BLOCK (USER_NAME; bluff)
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 exit
31050. fi
31051. 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_REG, 
31055. INDEX_TYPE, INDEX_RANGE_ASSURED, INDEX_LENGTH and CASE_FORMAT. */
31056. if CASE_FORMAT # 'CONDTEST'
31057. then
31058. call DOCASE_INDEX_TO_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. call DOCASE_DEBUGSTUFF
31070. /* Generates last-case variable and block same constant if required. */
31071. call DOCASE_INFO_SAVE
31072. /* Store in stack all data needed by CASE and ESACOD to complete 
31073. case processing. */
31074. if LABEL €
31075. then generate (LABEL | | DS QU)
31076. fi
31077. mexit
31078. end
31082.  BLOC DOCASE_EXTRACT_OPERANDS
31083.      /* Validate operands and issue any error messages; set INDEX_REG,
31084.       INDEX_TYPE, INDEX_RANGE_ASSURED, INDEX_LENGTH, and CASE_FORMAT. */
31085.      
31086.      IF OPTION = 'SIMPLE' OR = 'SPARSE'
31087.            THEN
31088.            CASE_FORMAT := OPTION
31089.            ELSE
31090.            IF OPTION = 'ONLY'
31091.                THEN /* Allow range specification as second operand of macro, also. */
31092.                INDEX_RANGE_ASSURED := TRUE
31093.                ELSE
31094.                IF OPTION = 'ADD' OR = 'IFNAME'
31095.                    THEN
31096.                    notes (6, 'STEC3102: ' || OPTION ||
31097.                        ' INVALID SECOND OPERAND—IGNORED')
31098.                    fi
31099.                    fi
31100.                    IF INDEX = ''
31101.                        THEN
31102.                        CASE_FORMAT := 'COMPLEMENT'
31103.                        ELSE
31104.                        CASE_FORMAT := 'GENERAL'
31105.                        IF INDEX(1) = 'IFNAME' OR = 'ONLY'
31106.                            THEN
31107.                            notes (6, 'STEC3101 WARNING—'' || INDEX(1) ||
31108.                                ' NOT ASSUMED AS INDEX; USE "DOCASE .'' || INDEX(1) ||
31109.                                ' FOR RANGE SPEC')
31110.                            fi
31111.                            fi
31112.                            IF RANGE = 'ONLY'
31113.                                THEN
31114.                                INDEX_RANGE_ASSURED := TRUE
31115.                                ELSE
31116.                                IF RANGE = 'ADD' OR = 'IFNAME'
31117.                                    THEN
31118.                                    notes (6, 'STEC3101: '' || RANGE ||
31119.                                        ' INVALID THIRD OPERAND—IGNORED')
31120.                                    fi
31121.                                    fi
31122.                                    IF INDEX_LENGTH = '0' /* Assume got CHARCOMP */
31123.                                        IF INDEX = 1 AND INDEX[1,1] = '(*'
31124.                                            THEN /* A one-element sublist was specified; we take it to be a
31125.                                                register. */
31126.                                                INDEX_REG := INDEX(1)
31127.                                                INDEX_TYPE := 'S' /* Index is specified as a register. */
31128.                                        ELSE
31129.                                                INDEX_REG := '1'
31130.                                        IF N'INDEX > 1
31131.                                            THEN
31132.                                            INDEX_REG := INDEX(2)
31133.                                            IF INDEX_TYPE = 'W' AND N'INDEX AND N'B' AND N'B'
31134.                                                THEN
31135.                                                INDEX_LENGTH := INDEX_TYPE /* Operand two is length specification. */
31136.                                                CASE_FORMAT := 'CHARCOMP' /* Change format to CHARCOMP. */
31137.                                            fi
31138.                                            fi
31139.                                            fi
31140.                                            fi
31141.                                            fi
31142.                                            fi
31143.                                            fi
31144.                                            fi
31145.                                            fi
31146.                                            fi
31147.                                            fi
C-36
31149. PROC DOCASE_INDEX_TO_REG
31150. /* If case format is GENERAL, SPARSE, or CHARCCRT and the index is a
31151. byte, save symbolic address of the index in INDEX_ADDR, otherwise
31152. set INDEX_ADDR to null and generate code to put index into GPR1.
31153. Given: This proc is not called for CCNDTEST format. */
31155. INDEX_ADDR := "" /* Assume index will be stored in GPR1. */
31156. DOCASE INDEX_TYPE ONLY
31157. OF
31158. CASE ('E') /* Register index. */
31159. IF INDEX_REG = '1'
31160. THEN generate (LABEL II ' LR 1,' || INDEX_REG)
31161. LABEL := ""
31162. ELSE
31163. END;
31164. CASE ('W') /* Word index. */
31165. generate (LABEL II ' L 1,' || INDEX(1))
31166. LABEL := ""
31167. CASE ('H') /* Halfword index. */
31168. generate (LABEL II ' LH 1,' || INDEX(1))
31169. LABEL := ""
31170. CASE ('B') /* Byte index. */
31171. IF CASE_FORMAT = 'SIMPLE'
31172. THEN generate (LABEL II ' SR 1,1')
31173. LABEL := ""
31174. ELSE
31175. generate (LABEL II ' IC 1,' || INDEX(1))
31176. END;
31177. ELSE
31178. generate (INDEX_ADDR := INDEX(1))
31179. END;
31180. END;
31181. /* Postpone loading cf index into register; we may want to do CLI's. */
31182. END;
31183. END;
31184. END;
31185. END.
"OCASE" macro -- 26 June 1973

31187. PROC DOCASE_GENERAL_SETUP
31188. /* Generate branch to beginning of general handler for general format
31189. DOCASE. In any case, advance GCASE_NEST_LEVEL for the GCASE stack
31190. and initialize the GCASE globals. It is assumed that this proc is
called only for GENERAL, SPARSE, and CHARCOMP case formats. */
31191. label 1, J /* Temporaries. */
31192. if CASE_FORMAT = 'GENERAL'
31193. then
31194. generate (LABEL | '' B || BLOCK_LABEL_PREFIX || 'BEG')
31195. /* Generate branch to general handler which is defined at ESACOD. */
31196. LABEL := ''
31197. BRANCH_TO_CASE1 := true /* Albeit indirectly. */
31198. fi
31199. GCASE_NEST_LEVEL := GCASE_NEST_LEVEL + 1 /* Advance GCASE stack. */
31200. if GCASE_NEST_LEVEL <= GCASE_NEST_LIMIT
31201. then /* Clear GCASE globals. */
31202. MAX_CASE_VALUE(GCASE_NEST_LEVEL) := -1
31203. /* Maximum branch vector value found. */
31204. NEXT_COMP_LABEL_NO(GCASE_NEST_LEVEL) := 1
31205. /* Case number for next comparison case label to be generated. */
31206. J := GCASE_NEST_LEVEL * 256
31207. I := J - 255
31208. if CASE_FORMAT = 'GENERAL'
31209. then /* Clear CASE_OCCURS bits. */
31210. CASE_OCCURS(I) := false
31211. else /* GCASE stack overflow. */
31212. M note (12, 'STRC3104 GENERAL/SPARSE/CHARCOMP GCASE NESTING LEVEL ' II
31213. GCASE_NEST_LEVEL || ' EXCEEDS MAXIMUM OF ' ||
31214. GCASE_NEST_LIMIT || '--MACROS MUST BE MODIFIED')
31215. fi
31216. fi
31217. corp
"DOCASE" Macro -- 26 June 1973

31226. **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. endif
31232. then
31233. endif
31234. endif
31235. endif
31236. endif
31237. endif
31238. endif
31239. endif
31240. endif
31241. endif
31242. endif
31243. endif
31244. endif
31245. endif
31246. endif
31247. endif
31248. endif
31249. endif
31250. endif
31251. endif
31252. endif
31253. endif
31254. endif
31255. endif
31256. endif
31257. endif
31258. endif
31259. endif
"DCASE" Macro -- 26 June 1973

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 || CASE1 || TRUE || FALSE || 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. */
CASE Macro

The CASE macro is used to specify a block of code which is one of the alternatives for the immediately surrounding DOCASE macro. If the CASE macro is not the immediate daughter of a DOCASE and no fixup is possible, a BLOCK macro is substituted. Otherwise, the information stored by the DOCASE is extracted and the operands of the CASE are processed to produce the necessary code for the selecting of this block in the indicated case. Finally, any debugging code required is generated.

```c
int CASE_COUNTER, /* Case number for this CASE maintained in mother info. */
COMP_LABEL_No, /* Number to be used in next compare label to be defined. */
I, /* SYSLIST index. */
```

```c
bit CASE_LABEL_REQD, /* true unless DOCASE is falling through into first CASE. */
INDEX_RANGE_ASSURED, /* true if we have been assured (by "DOCASE ---,ONLY") that no values other than those specified by CASE operands will occur. */
EQUAL_TEST_OUTSTANDING,
/* Indicates that a compare for the current operand has been generated but the "BE" to the beginning of the block (or "BNE" around the block) has not been generated yet. */
RANGE_TEST_OUTSTANDING,
/* Indicates that a compare for the current range operand has been generated as well as the branch if below the range; the branch if within the range to the beginning of the block (or "BH" around the block) has not been generated yet. */
MISC_FOUND, /* Indicates whether MISC has been found yet. */
MULTIPLESOF4, /* Indicates whether all the self-defining operands of the CASE macros processed so far are multiples of 4. */
UNEXPECTED_OPERANDS_FOUND /* Indicates whether any operands have been found so far in the CASE macros' operands which either were symbolic or were self-definers not equal to their own case number. */
```

```c
char CASE_FORMAT, /* Type of CASE operands expected: GENERAL, SPARSE, SIMPLE, CHARCOMP, or CONDTEST. */
INDEX_ADDR, /* Symbolic address of byte or CHARCOMP operand. */
LABEL, /* Outstanding label waiting to be generated. */
NEXT_CASE, /* Label to be generated on next SIMPLE or CONDTEST CASE macro. */
```

INDEX_LENGTH /* Length of CHARCOMP index. */

C-41
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, "STBC3290 CASE TREATED AS "BLOCK" MACRO")
33055. call BLOCK (USER_NAME;)
33056. print
33057. fi
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 then /* during PUSH_NEW_BLOCK (viz., stack overflow) */
33061. mexit
33066. fi
33067. if REL "ID" or MASK "ID"
33068. then
33069. note (8, "STBC3110 REL= OR MASK= NOT IN PARENTHESES—IGNORED")
33071. call CASE_GET_DOCASE_INFO
33072. /* Extract CASE_FORMAT, CASE_LABEL_REQD, CASE_COUNTER, MISC_FOUND, MAMA_BLOCK_PREFIX, INDEX_RANGE_ASSUMED, INDEX_ADDR, and INDEX_LENGTH from other DOCASE block. */
33073. if USER_NAME "ID"
33074. then
33075. if SYSLIST(1) = 'MISC'
33076. then
33077. call CASE_MISC_PROCESS
33078. /* Completely process miscellaneous CASE block. */
33079. else
33080. if CASE_FORMAT = 'GENERAL' or = 'SPARSE' or = 'CHARCOMP'
33081. then
33082. if GCASE_NEST_LEVEL < GCASE NEST LIMIT
33083. then
33084. call CASE_PROCESSCOMPAREOPERANDS
33085. /* 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. */
33086. if CASE_FORMAT = 'GENERAL'
33087. then
33088. call CASE_PROCESSVECTOROPERANDS
33089. /* Generate labels and save information about any operands which are to be handled via branch vector. */
33090. fi
33091. else
33092. if CASE_FORMAT = 'SIMPLE'
33093. then
33094. call CASE_BCT_GEN
33095. /* Generate BCT instruction for this case. */
33096. else
33097. if CASE_FORMAT = 'CONDTEST'
33098. then
33099. call CASE_CONDTEST_GEN
33100. /* Generate conditional test specified on CASE macro. */
33101. fi
33102. fi
33103. call CASE_TRACE_COUNTER
33104. /* Generate code to count this block, note last case number, and/or display block name if appropriate debugging requested. */
33105. call CASE_UPDATE_INFO
33106. /* Update the information stored in mother DOCASE block. */
33107. if LABEL "ID"
33108. then
33109. generate (LABEL "ID")
33110. fi
33111. fi
PREG CASE_POSITION_CHECK
    /* 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
            mnote (8, 'STRC3301 "CASE" NOT IMMEDIATE LAUGHTER OF "DOCASE"')
            ERROR_OCCURRED := false /* Assume no fixup possible. */
        if CURRENT_NEST_LEVEL > 1 and
            BLOCK_TYPE(CURRENT_NEST_LEVEL-1) = 'DOCASE'
            then
                mnote (8, 'STRC3303 ASSUMING TWO "BLENS" OMITTED--INSERTED')
                call BLEND ( ; )
                ERROR_OCCURRED := false /* Note patch up. */
            else
                if CURRENT_NEST_LEVEL > 2 and
                   BLOCK_TYPE(CURRENT_NEST_LEVEL-2) = 'DOCASE'
                    then
                        mnote (8, 'STRC3303 ASSUMING TWO "BLENS" OMITTED--INSERTED')
                        call BLEND ( ; )
                        call BLEND ( ; )
                        ERROR_OCCURRED := false
                    fi
                fi
            fi
        fi
    fi
end
/* CASE Macro -- 27 June 1973 */

PROC CASE_GET_DOCASE_INFO
  /* Extract DOCASE information being maintained in mother's stack position. */
  char * 
  ans; /* Temporary. */
  if (CURRENT_NEST_LEVEL - 1)
    CASE_COUNTER := OPERAND1(MOB)
  if (SPLIT(F1) == 'MISC')
    X := CASE_COUNTER := CASE_COUNTER + 1
  if
  INDEX_ADDRES := OPERAND2(MOB)
  CASE_FORMAT := OPERAND3(MOB)
  INDEX_LENGTH := OPERAND4(MOB)
  Y := INFORMATION(MOB)
  CASE_LABEL_REQD := X[1, 1]
  MISC_POUND := X[2, 1]
  MULTIPLEOPS := X[3, 1]
  UNEXPECTED_OPERANDS_REQD := X[4, 1]
  INDEX_RANGE_ASSUMED := X[5, 1]
  PARAM_BLOCK_PIECE := '*' || BLOCK_NUMBER(MOB)

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
CASE Macro

33192. 
33193. PROC CASE PROCESS COMPARE OPERANDS
33194. 
33195. /* Generate compare-and-branch sequences for all "symbolic" operands
33196. (i.e., those which cannot be handled by the branch vector: all non-
33197. self-defining terms, all self-defining operands which are not in
33198. the range 0-255 inclusive, and all "range" operands (m,R)
33199. where either m or R is either non-self-defining or outside the
33200. range 0-255) or for all operands if LOCASE was flagged as SPARSE or
33201. CHARCOMP. */
33202. 
33203. I := 1 /* Start search with first operand. */
33204. COMP_LABEL_NO := NEXT_COMP_LABEL_NO(GCASE_NEST_LEVEL)
33205. /* Note the next compare label number. */
33206. EQUAL_TEST_OUTSTANDING, RANGE_TEST_OUTSTANDING := false
33207. 
33208. while I < N'SYSLIST
33209. do /* (CASE_FORMAT = 'SPARSE' OR 'CHARCOMP' OR
33210. [N'SYSLIST(I) < 1 OR (T'SYSLIST(I) = 'N') OR
33211. OR (T'SYSLIST(I) > 1 AND (T'SYSLIST(I,1) = 'N') OR
33212. SYSLIST(I,1) > 0 AND T'SYSLIST(I,2) = 'N' OR
33213. STYSLIST(I,2) < 0 OR T'SYSLIST(I,2) > 255))
33214. then
33215. if EQUAL_TEST_OUTSTANDING
33216. then
33217. LABEL := BLOCK_LABEL_PREFIX || 'BEG'
33218. generate (' BEG ' || LABEL)
33219. /* After leaving this proc, someone will generate the BEG label
33220. at the beginning of the block. */
33221. EQUAL_TEST_OUTSTANDING := false
33222. else
33223. if RANGE_TEST_OUTSTANDING
33224. then
33225. LABEL := BLOCK_LABEL_PREFIX || 'BEG'
33226. generate (' ' BN ' || LABEL)
33227. /* Again, by leaving this proc, someone will generate the BN label
33228. when we branched on lower end of range. */
33229. RANGE_TEST_OUTSTANDING := false
33230. fi
33231. if CASE_LABEL_REQD
33232. then
33233. COMP_LABEL := MAMA_BLOCK_PREFIX || 'C' || COMP_LABEL_NO
33234. /* Generate label name to be attached to first instruction. */
33235. UNEXPECTED_OPERANDS_FOUND := TRUE
33236. /* All compare operands have now been processed. */
33237. if EQUAL_TEST_OUTSTANDING
33238. then
33239. generate (' ' BNE ' || MAMA_BLOCK_PREFIX || 'C' ||
33240. COMP_LABEL_NO)
33241. fi
33242. fi
33243. fi
33244. od /* (Termination: I is incremented above and not modified by
33245. called proc; N'SYSLIST is fixed; I must eventually exceed
33246. N'SYSLIST). */
33247. 
33248. if EQUAL_TEST_OUTSTANDING
33249. then
33250. /* Generate branch to next symbolic case. */
33251. generate (' ' BNE ' || MAMA_BLOCK_PREFIX || 'C' ||
33252. COMP_LABEL_NO)
33253. else
33254. if RANGE_TEST_OUTSTANDING
33255. then
33256. generate (' ' BN ' || COMP_LABEL)
33257. /* Generate branch to next compare case. Label was left in COMP_LABEL
33258. when we branched on lower end of range. */
33259. fi
33260. fi
33261. NEXT_COMP_LABEL_NO(GCASE_NEST_LEVEL) := COMP_LABEL_NO
33262. /* Store case number of next symbolic case to be defined. */
33263. fi
33264. /* Advance to next operand of CASE. */
33265. od /* (Termination: I is incremented above and not modified by
33266. called proc; N'SYSLIST is fixed; I must eventually exceed
33267. N'SYSLIST). */
33268. 
33269. if EQUAL_TEST_OUTSTANDING
33270. then
33271. /* Generate branch to next symbolic case. */
33272. generate (' ' BNE ' || MAMA_BLOCK_PREFIX || 'C' ||
33273. COMP_LABEL_NO)
33274. else
33275. if RANGE_TEST_OUTSTANDING
33276. then
33277. generate (' ' BN ' || COMP_LABEL)
33278. /* Generate branch to next compare case. Label was left in COMP_LABEL
33279. when we branched on lower end of range. */
33280. fi
33281. fi
33282. NEXT_COMP_LABEL_NO(GCASE_NEST_LEVEL) := COMP_LABEL_NO
33283. /* Store case number of next symbolic case to be defined. */
CASE Macro -- 27 June 1973

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

C-46
CASE Macro

```
if INDEX_ADDR = ''
  then /* Index is in GPR1. */
    generate (COMP_LABEL || ', ' || INDEX_ADDR || ', ' || INDEX_LENGTH || ')
  else /* Index is at INDEX_ADDR. */
    if CASE_FORMAT = 'CHARCCMP'
      then
        INSERT := *
        if SYSLIST(1)(1,1) != '
          then /* Character string. */
            INSERT := "=
          else
            generate (COMP_LABEL || ', ' || INDEX_ADDR || ', ' || INDEX_LENGTH || ')
        if SYSLIST(1,1)[1,1] = ",
          then /* Literal without the 
          (operand ends with "). */
            INSERT := "=
          else
            generate (COMP_LABEL || ', ' || INDEX_ADDR || ', ' || INDEX_LENGTH || ')
      else generate (COMP_LABEL || ', ' || INDEX_ADDR || ', ' || INDEX_LENGTH || ')
    if COMP_LABEL != '
      then
        generating (COMP_LABEL || ', ' || INDEX_ADDR || ', ' || INDEX_LENGTH || ')
      else RANGEPARAMS GENERATED (G, 'STRC3312 ', II SYSLIST(1,1)) || ' ' || (INDEX_ADDR || ')
    if 'SYSLIST(1) < 1
      then /* Operand is not a range. */
        EQUAL_TEST_OUTSTANDING := true
      else /* A range has been specified: (g,)
        RANGEPARAMS GENERATED (G, 'STRC3312 ', II SYSLIST(1,1)) || ' ' || (INDEX_ADDR || ')
      if 'SYSLIST(1) > 2
        then
          RANGEPARAMS GENERATED (G, 'STRC3312 ', II SYSLIST(1,1)) || ' ' || (INDEX_ADDR || ')
      else
        RANGEPARAMS GENERATED (G, 'STRC3312 ', II SYSLIST(1,1)) || ' ' || (INDEX_ADDR || ')
      case CASE_FORMAT = 'CHARCCMP'
      then
        INSERT := *
        /* Go through the same business figuring out the insert for a as
        we did for g. */
        if SYSLIST(1,2)(1,1) != 
          then /* Character string. */
            INSERT := "=
          else
            generate (COMP_LABEL || ', ' || INDEX_ADDR || ', ' || INDEX_LENGTH || ')
        if SYSLIST(1,2)[1,1] = ",
          then /* Literal without the 
          (operand ends with "). */
            INSERT := "=
          else
            generate (COMP_LABEL || ', ' || INDEX_ADDR || ', ' || INDEX_LENGTH || ')
      else generate (COMP_LABEL || ', ' || INDEX_ADDR || ', ' || INDEX_LENGTH || ')
    fi
fi
if COMP_LABEL != '
  then
    generating (COMP_LABEL || ', ' || INDEX_ADDR || ', ' || INDEX_LENGTH || ')
  else RANGEPARAMS GENERATED (G, 'STRC3312 ', II SYSLIST(1,1)) || ' ' || (INDEX_ADDR || ')
```

C-47
CASE PROCESS_VECTOR_OPERANDS

/* Generate labels and note that CASE OCCURS for any operands which
   can be handled via branch vector: viz., any of the form m or
   (m,n) where m and n are self-defining terms in the range
   0-255 inclusive. This procedure assumes the CASE_FORMAT is general
   (not sparse). */

int BASE, /* Array position in CASE_OCCURS of the case for zero. */
OP, /* Case value currently being considered. */
LIMIT /* High limit in range operands. */

BASE := ((GCASE_NEST_LEVEL - 1) * 256) + 1 /* Calculate offset in CASE_OCCURS array for this DOCASE. */

if N'SYSLIST > 0

then /* One or more operands were specified. */

I := 1 /* Start with first operand. */

while I ≤ N'SYSLIST

do T'SYSLIST(I,1) = 'm'
    then /* m is a self-defining term. */
        OP := T'SYSLIST(I,1)
        if OP \in [0,255] and OP \geq 0
            then /* It's in the range. */
                LIMIT := T'SYSLIST(I,2)
                if LIMIT \in [0,255] and LIMIT \leq OP
                    then /* n is self-defining or not present. */
                        if N'SYSLIST(I) > 2
                            then mnote (8, 'STRC3312 ' || SYSLIST(I) || ' INVALID--ONLY FIRST TWO SUBOPERANDS PROCESSED')
                        fi
                fi
            if T'SYSLIST(I,2) = 'n'
                then LIMIT := T'SYSLIST(I,2)
                    fi
            fi
        fi
    then LIMIT := OP
        fi
    fi

if LIMIT > MAX_CASE_VALUE(GCASE_NEST_LEVEL)

then /* We have found a new maximum case number. */

MAX_CASE_VALUE(GCASE_NEST_LEVEL) := LIMIT

while OP \leq LIMIT

do CASE_OCCURS(BASE+OP) := true

SUBCASE (ABA_BLCCR_PREFIX 'G' || OP 'DS 0E')

if OP ≠ CASE_COUNTER

then UNEXPECTED_OPERANDS_FOUND := true

fi

if OP/4*4 = 0

then MULTIPLESOP4 := false

fi

OP := OP + 1

fi /* (Termination: OP is incremented, LIMIT is fixed during loop; OP must eventually exceed LIMIT.) */

fi

if I = I + 1 /* do do next operand. */

then /* (Termination: I is incremented, N'SYSLIST is fixed during loop; I must eventually exceed N'SYSLIST.) */

else /* No operands present. */

call CASE_ASSUMED_VECTORCASE /* If all the CASE macros so far have had no operands or only

"expected" ones (integers which match the case counter), assume

this one matches too and generate the single operand. */

fi

fi
```
33434. ENDG CASE_ASSUMED_VECTOR_CASE
33435. /* Generate label for branch vector cases with no operands. Value used
33436. is the next higher value than the maximum used so far. CASE_OCCURRS
33437. is noted, and the SELFDEF_COUNT and MAX_CASE_VALUE are updated. If
33438. any previous operands have occurred which were not expected, a
33439. message is printed. */
33440. initialize GUESS /* Assumed operand. */
33441. GUESS := MAX_CASE_VALUE(GCASE_NEST_LEVEL) + 1
33442. /* Guess at what omitted operand was intended. */
33443. if GUESS <= 0
33444. then /* First guess. */
33445. GUESS := 1
33446. fi
33447. MAX_CASE_VALUE(GCASE_NEST_LEVEL) := GUESS
33448. CASE_OCCURRS(BASE+GUESS) := true
33449. generate (MAMA_BLOCK_PREFIX || "G" || GUESS || "ES OH")
33450. if UNEXPECTED_OPERANDS_FOUND
33451. then
33452. note ("SC3306 EARLIER UNEXPECTED OPERAND IMPLIES THIS TO BE CASE – ",
33453. GUESS)
33454. fi
33455. fi
```

"CASE" Macro -- 27 June 1973
"CASE" MACRO  --  27 June 1973

33459.  PROC CASE_SET_NAMES
33460.  /* Set LABEL if one will be required on this SIMPLE or CONDTEST case
33461.     code (usually is; only exception involves when DOCASE falls through
33462.     to first case).  Also set NEXT_CASE with label of next case to be
33463.     generated.  LABEL is always null at entry. */
33464.     if CASE_LABEL_REQ
33465.     then
33466.         LABEL := MAMA_BLOCK_PREFIX || 'C' || CASE_COUNTER
33467.     fi
33468.     I := CASE_COUNTER + 1
33469.     NEXT_CASE := MAMA_BLOCK_PREFIX || 'C' || I
33470.     Corp
33471.     Corp
/* Case Macro -- 27 June 1973 */

33473. DEFINE CASE_BCT_GEN
          /* Generate BCT for this single case. Verify operand, if any. */
33476.         GENERATE (LABEL || ' BCT ' || NEXT_CASE)
33477.         LABEL := ";" || SYSLIST(1) || '"'
33478.         if SYSLIST(1) = '0'
33479.     then /* An operand was specified. */
33480.         if SYSLIST(1) = 'W'
33481.     then /* Operand is a self-defining term. */
33482.         if SYSLIST(1) = CASE_COUNTER
33483.     then
33484.         MESSAGE (0, 'SYNC:3307 OPERAND INVALID VALUE OF SIMPLE CASE ' ||
33485.             CASE_COUNTER)
33486.     fi
33487.     else /* Operand is not self-defining term. */
33488.         MESSAGE (0, 'SYNC:3307 OPERAND MUST BE SELF-DEFINING TERM OR OMITTED ' ||
33489.             ON SIMPLE CASE ' || CASE_COUNTER)
33490.     fi
33491. fi
33492. fi

C-51
"CASE" Macro -- 27 June 1973

PROC CASE_CONDTEST GEN

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

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

/* Generate conditional expression processor (LABEL, SYSLIST)
(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

fi

C-52
"CASE" Macro -- 27 June 1973

33535. proc CASE_MISC_PROCESS
33536.     /* Generate label for miscellaneous block, including a branch around
33537.     this block in case we were falling through into it.  Note that a
33538.     miscellaneous block has been found and verify that no other
33539.     miscellaneous case has occurred for this DOCASE. */
33540.     if ~ CASE_LABEL_REQD
33541.         then /* We are falling through into this block. */
33542.             generate (* B * ||
33543.                 MAMA_BLOCK_PREFIX || 'C' || CASE_COUNTER)
33544.             /* Generate branch to next case number (probably C1). */
33545.         fi
33546.     if MISC_FOUND
33547.         then
33548.             note (8, 'STRC3311 MULTIPLE MISC CASES IN DOCASE--THIS BLOCK ' ||
33549.                 'IS DEAD CODE')
33550.         else
33551.             LABEL := MAMA_BLOCK_PREFIX || 'MSC'
33552.             /* Make MSC label outstanding (generate on next instruction).
33553.             It is assumed that no LABEL can be outstanding when CASE_MISC_PROCESS
33554.             is called. */
33555.             MISC_FOUND := true
33556.         fi
33557.     if INDEX_RANGE_ASSURED
33558.         then
33559.             note (8, 'STRC3300 "DOCASE ...ONLY" INVALID WITH MISC')
33560.         INDEX_RANGE_ASSURED := false
33561.     fi
33562.     corp C-53
CASE- Macro -- 27 June 1973

33565. PROC CASE_TRACE_COUNTER
33566. /* Generate any debugging counters and/or labels requested. */
33567. if DEBUG_BLOCKCOUNTS_REQD or DEBUG_BLOCKNAMES_REQD
33568. then
33569. if DEBUG_BLOCKCOUNTS_REQD
33570. then
33571. /* Generate code to advance this case's counter. */
33572. generate (LABEL || ' LH 1,' || BLOCK_LABEL_PREFIX || 'CTR')
33573. generate (' LA 1,1(1)')
33574. generate (' STB 1,' || BLOCK_LABEL_PREFIX || 'CTR')
33575. if SYSLIST(1) = "MISC" or CASE_COUNTER > 255
33576. then
33577. generate (' MVI ' || MAMA_BLOCK_PREFIX || "LSC,X'FF'"
33578. else
33579. HEX_IN := CASE_COUNTER
33580. call XHEX ( )
33581. generate (' STB,T,' || MAMA_BLOCK_PREFIX || "LSC,X'01'"
33582. HEX_OUT := "CASE NUMBER FOR TRACING"
33583. fi
33584. if DEBUG_BLOCKNAMES_REQD
33585. then
33586. generate ("DC C'CURRENT_NEST_LEVEL''"
33587. if DEBUG_BLOCKCOUNTS_REQD
33588. then
33589. generate ("DC H'0' CASE COUNT"
33600. fi
33601."
33602. C-54
"CASE" Macro -- 27 June 1973

33604.  BLOG CASE_UPDATE_INFO
33605.  /* Returns to another DOCASE level possibly updated information which
33606.  was extracted by CASE_GET_DOCASE_INFO. */
33607.  ANI MON /* Index level of DOCASE block. */
33608.
33609.  MON := CURRENT_NEST_LEVEL - 1
33610.  INFORMATION(MON) := 'LCHR || MISCFOUND || MULTIPLE5OF4 ||
33611.  UNEXPECTED_OPERANDS_FOUND || INDEX_RANGE_ASSURED
33612.  /* First byte indicates case label is required on next case. */
33613.  OPERAND1(MON) := CASE_COUNTER
33614.  /* No need to update OPERAND2 (INDEX_ADDR) or OPERAND3 (CASE_FORMAT)
33615.  or OPERAND4 (INDEX_LENGTH). None ever change. */
33616.  CEEP
"ESAC" Macro -- 3 July 1973

35001. macro ESAC ( ; USER_NAME)
35002.  /* Generate end to match CASE block. Do standard block closing, then
generate branch to end of ather ECASE block. */
35005.  call TRACE_PRINTER ( ; 'ESAC')
35006.  /* Print macro name "ESAC" in mnote if tracing on. */
35007.  if CURRENT_NEST_LEVEL <= NESTING_LIMIT
35008.    then
35009.      call VERIFY_END ( ; 'CASE', USER_NAME)
35010.      /* Verifies current block has the name specified by the USER_NAME
operand of the ESAC macro (if any) and that it is a CASE block.
Various errors receive messages and either intermediate blocks are
SLENDed 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(CURRENT_NEST_LEVEL) = 'CASE', then
ERROR_OCCURRED will be set false and CURRENT_NEST_LEVEL will not
be modified.) */
35018.      if ERROR_OCCURRED
35019.        mexit
35020.      fi
35023.  fi
35029.  if CURRENT_NEST_LEVEL <= NESTING_LIMIT
35030.    then
35031.      generate ('$' 11 BLOCK_NUMBER(CURRENT_NEST_LEVEL) II
35032.        'END')
35033.        /* Generate branch to end of DACASE. */
35036.      fi
35039.  end
35041.  /* (Lemma: if CURRENT_NEST_LEVEL > 0 and
USER_NAME = "" or = BLOCK_NAME(CURRENT_NEST_LEVEL) ] and
BLOCK_TYPE(CURRENT_NEST_LEVEL) = 'CASE' at entry to ESAC, then
CURRENT_NEST_LEVEL will be decremented by exactly one.) */

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
ESACOD

37001.  BEGIN
37002.  ESACOD ( : USER_NAME)
37003.  /* Generates final part of DOCASE processing: for SIMPLE, CONDTRED,
37004.         or SParse type DOCASE, the EQU for the MISC block. (or END of DOCASE)
37005.         to the last generated branch target is required; for GENERAL type
37006.         DOCASE, the branch vector and the transfer to any symbolic
37007.         compares or MISC block must be generated. Finally, the block is
37008.         popped. */
37009.  INC
37010.  CASE_COUNTER, /* Holds number of last case generated. */
37011.  T,     /* Temporary. */
37012.  COMP_LABEL_NO, /* Label number of outstanding compare case. */
37013.  MAX_SD_VALUE, /* Maximum self-defined operand. */
37014.  BASE /* Index within CASE_OCCURS array for CASE 0. */
37015.  BIT
37016.  MISC_FOUND, /* Indicates whether a MISC CASE was found. */
37017.  MULTIPLESOF4, /* Indicates whether all branch-vector operands were
37018.         multiples of 4. */
37019.  INDEX_RANGE_ASSURED, /* true if we have been assured (by
37020.         "DOCASE *** ONLY") that no values other than those specified
37021.         by CASE operands will occur. */
37022.  ANY_COMP_CASES, /* Indicates whether any "compare" cases were
37023.         generated (either CHARCOMP or symbolic general case operands). */
37024.  ANY_SELFDEF_CASES, /* Indicates whether any "self-defining" cases (to
37025.         be handled by branch vector) were generated. */
37026.  RANGE_TEST_REQD /* Indicates that both branch vector and compare
37027.         operands were present. */
37028.  CHAR
37029.  CASEFORMAT, /* Type of CASEs present: GENERAL, SPARSE,
37030.         CHARCOMP, SIMPLE, or CONDTRED. */
37031.  INDEX_ADDR, /* Address of DOCASE index. */
37032.  NOCASE, /* Label for branch vector processing used for unspecified
37033.         cases. */
37034.  LABEL /* Any outstanding label waiting to be generated. */
37035.  /* (Ground rules: No ESACOD proc modifies CURRENT_NEST_LEVEL.
37036.         This can be shown by referring to the cross-reference index.) */

"ESACOD" Macro -- 3 July 1973
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. */

   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_LABEL_PREF, INDEX_RANGE_ASSURED, and MULTIPLES_OF_4 from stack. */
   if CASE_FORMAT = 'GENERAL'
   then
      call ESACOD_GENERAL_CASE_CHOICE
   else
      if CASE_FORMAT = 'SPARSE' or 'CHARCCMP'
      then
         T := NEXT_COMP_LABEL_NO(CASE_NEST_LEVEL)
         /* We need to define last compare case target. */
         CASE_NEST_LEVEL := CASE_NEST_LEVEL - 1
         /* Pop GCASE stack. */
         else /* CONDTST or SIMPLE */
            T := CASE_COUNTER + 1
            /* We need to define last conditional test target. */
         fi
      fi
   if MISC_FOUND
   then
      generate (BLOCK_LABEL_PREFIX || 'C' || T || ' EQU ' ||
      BLOCK_LABEL_PREFIX || 'MSC')
   else
      generate (BLOCK_LABEL_PREFIX || 'C' || T || ' DS OH')
   fi
   call POP_OLD_BLOCK ()
/* (Lemma: POP_OLD_BLOCK decrements CURRENT_NEST_LEVEL by exactly one. */

   mendl
/* (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. */
ESACOD" Macro — 3 July 1973

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

37096. CASE_COUNTER := OPERAND1(CURRENT_NEST_LEVEL)
37097. INDEX_ADDR := OPERAND2(CURRENT_NEST_LEVEL)
37098. CASE_FORMAT := OPERAND3(CURRENT_NEST_LEVEL)
37099. MISC_FOUND := INFORMATION(CURRENT_NEST_LEVEL)[2,1]
37100. MULTIPLESOF4 := INFORMATION(CURRENT_NEST_LEVEL)[3,1]
37101. INDEX_RANGE_ASSURED := INFORMATION(CURRENT_NEST_LEVEL)[5,1]
37102. BLOCK_LABEL_PREFIX := '$' || BLOCK_NUMBER(CURRENT_NEST_LEVEL)
37103. GOTO
"ESACOD" Macro -- 3 July 1973

37105. ENSC ESACOD_GENERAL_CASE_CHOICE
37106. /* Generate all code to complete processing of general case.
37107. Includes the generation of a branch vector, if required. */
37108.
37109. call ESACOD_GENERAL_CASE_INFO
37110. /* Pops MAX_SD_VALUE, CCMN_BASE_NO, and BASE (of CASE_OCCURS array) out of CASE stack. */
37111.
37112. if ~ ERROR_OCCURRED
37113. then
37114. if ANY_SELFDEF_CASES
37115. then
37116. call ESACOD_SELFDEF_GEN
37117. /* Handles branch vector-type implementation for all cases which
37118. contain self-defining terms (of value < 256). Also generates
37119. linkage for any other terms and/or MISC case which were used
37120. with the self-definers. */
37121.
37122. else
37123. if ANY_COMP_CASES
37124. then
37125. call ESACOD_GENERAL_SYM_ONLY
37126. /* Generate linkage to process symbolic operands and MISC in the
37127. absence of self-definers. */
37128. else
37129. mnote (8, 'STRC3701 DOCASE CONTAINS NO VALID CASES')
37130. fi
37131. fi
37132. fi
37133. corp
37135.     DGEG ESACOD_GENERAL_SYMBOLS
37136.     /* Generate linkage to process symbolic operands and MISC in the
37137.     absence of self-definers (self-defined terms of value < 256). */
37138.     generate (BLOCK_LABEL_PREFIX || 'SIG' EQU ".")
37139.     if MISC_FOUND
37140.     generate (BLOCK_LABEL_PREFIX || 'C' || COMP_LABEL_NO || 'EQU ".")
37141.     else
37142.     generate (BLOCK_LABEL_PREFIX || 'C' || COMP_LABEL_NO || 'DS 0B')
37143.     fi
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. */
37156. LABEL := BLOCK_LABEL_PREFIX || 'BEG'
37157. /* Note that BEG label must be generated on first instruction. */
37158. docase ifany
37159. of
37160. case ANY_COMP_CASES
37161. NOCASE := BLOCK_LABEL_PREFIX || 'C1'
37162. esac
37163. case MISC_FOUND
37164. NOCASE := BLOCK_LABEL_PREFIX || 'MSC'
37165. esac
37166. case Misc
37167. NOCASE := BLOCK_LABEL_PREFIX || 'END'
37168. esac
37169. esacod
37170. RANGE_TEST_REQD := ((INDEX_RANGE_ASSURED) or ANY_COMP_CASES)
37171. if RANGE_TEST_REQD
37172. then
37173. call ESACOD_OUTOF_RANGE_CHECK
37174. /* Generate check for index out of the range 0 through
37175. MAX_SD_VALUE. */
37176. fi
37177. call ESACOD_BRVCT_GEN
37178. /* Generate branch vector and all final constants and equates
37179. required. */
37180. cor
ESACOD Macro — 3 July 1973

37182. 
37183. PROC ESACOD_GENERAL_CASE_INFO
37184. /* Pops following information out of GCASE stack. Indicates success
37185. or lack thereof) in ERROR_OCCURR. */
37186. int I
37187. I := GCASE_NEST_LEVEL
37188. if I > GCASE_NEST_LIMIT
37189. then
37190. ERROR_OCCURRED := true
37191. MAX_SD_VALUE := MAX_CASE_VALUE(I)
37192. ANY_SELFDEF_CASES := (MAX_SD_VALUE > 0)
37193. COMP_LABEL_NO := NEXT_COMP_LABEL_NO(I)
37194. ANY_COMP_CASES := (COMP_LABEL_NO > 1)
37195. BASE := ((I-1) * 256) + 1
37196. ERROR_OCCURRED := false
37197. else
37198. endif
37199. core
37200. */

C-63
37202.  ESSAC 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.
37207.  if INDEX_ADDR == ""
37208.  then /* Index is in GPR1 */
37209.  generate ('LABEL || ' LTF 1,'1')
37210.  if CASE_OCCURES(BASE)
37211.  then /* CASE 0 occurs */
37212.  generate (' BM ' || NOCASE)
37213.  else
37214.  generate (' BN ' || NOCASE)
37215.  fi
37216.  generate (' C 1,' || BLOCK_LABEL_PREFIX || 'SIZ')
37217.  generate (' BM ' || NOCASE)
37218.  if MULTIPLESOF4
37219.  then
37220.  generate (' LA 0,3')
37221.  generate (' WE 0,1')
37222.  generate (' BNZ ' || NOCASE)
37223.  fi
37224.
37225.  else
37226.  generate ('LABEL || ' CLI ' || INDEX_ADDR || ',' || MAX_SD_VALUE)
37227.  generate (' BM ' || NOCASE)
37228.  if MULTIPLESOF4
37229.  then
37230.  generate (' TM ' || INDEX_ADDR || "00000011")
37231.  generate (' BNZ ' || NOCASE)
37232.  fi
37233.  fi
37234.  LABEL := ""
37235.  core
37236.
37237.  core
37236. PMSG ESACOD_BR vec_GEN
37237. /* Generate branch vector proper. */
37238. int I, INCP
37240. 
37241. if INDEX_ADDR != ""
37242. then /* Generate code to put byte index into GPR1. */
37243. generate (LABEL || ' SR 1,1' )
37244. Label := ""
37245. generate (' IC 1,1 || INDEX_ADDR )
37246. fi
37247. if MULTIPLEOF4
37248. then
37249. INCP := 4
37250. else
37251. INCP := 1
37252. generate (LABEL || ' S LA 1,2' )
37253. Label := ""
37254. fi
37255. if CASE_OCCURS(BASE) OR INDEX_ADDR != ""
37256. then /* Zero case must be included in branch vector. */
37257. generate (LABEL || ' B #9(1)' )
37258. I := 0
37259. else
37260. generate (LABEL || ' B #9(1)' )
37261. I := INCP
37262. fi
37263. Label := ""
37264. while I ≤ MAX_SD_VALUE
37265. do
37266. if CASE_OCCURS(HASI+1)
37267. then
37268. generate ( ' B ' || BLOCK_LABEL_PREFIX || ' G' || I )
37269. else
37270. generate ( ' B ' || NOCASE)
37271. fi
37272. I := I + INCP
37273. od /* (termination: INCP > 0, so I is incremented in loop; 
37274. MAX_SD_VALUE is fixed, therefore I must eventually exceed 
37275. MAX_SD_VALUE.) */
37276. if RANGE_TEST_REQD and INDEX_ADDR != ""
37277. then
37278. generate (BLOCK_LABEL_PREFIX || "SZ DC P" || MAX_SD_VALUE || ""*
37279. fi
37280. fi
37281. if ANY_COMP_CASES
37282. then
37283. if MISC_FOUND
37284. then
37285. generate (BLOCK_LABEL_PREFIX || 'C' || COMP_LABEL_NO || ' EQU ' || BLOCK_LABEL_PREFIX || 'RSC')
37286. else
37287. generate (BLOCK_LABEL_PREFIX || 'C' || COMP_LABEL_NO || ' DS DH')
37288. fi
37289. fi
37290. fi
37291. 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.   `mexit`
41015. `then`
41016.   `fi`
41017. `exit`
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 `then`
41023.   `generate` (LABEL `||` 'DS OH')
41024. /* Define label if one required and not yet defined. */
41025. `else`
41026.   `fi`
41027. `mend`
"BLOCK" Macro -- 15 June 1973

41029. PROC BLOCK_TRACE_COUNTERS
41030. /* Generate debugging information required—block name constant
41031. and/or block counters. */
41032. IF DEBUG_BLOCKCOUNTS_REQD OR DEBUG_BLOCKNAMES_REQD
41033. THEN
41034. IF DEBUG_BLOCKCOUNTS_REQD
41035. THEN
41036. IF DEBUG_BLOCKNAMES_REQD
41037. THEN
41038. /* Generate block count incrementing instructions. */
41039. generate (LABEL || 'LB 1,' || BLOCK_LABEL_PREFIX || 'BLC')
41040. LABEL := 10 /* Clear LABEL to show it has been generated. */
41041. generate ("LA 1,1(1)")
41042. generate (LABEL || 'BLC')
41043. /* Generate branch around block name and/or count. */
41044. LABEL := BLOCK_LABEL_PREFIX || 'GO'
41045. IF DEBUG_BLOCKNAMES_REQD
41046. THEN
41047. generate ("DC C'" || BLOCK_LABEL_PREFIX || 'BLC')
41048. THEN
41049. generate ("DC C'" || BLOCK_NAME(CURRENT_NESTLEVEL))
41050. THEN
41051. IF DEBUG_BLOCKCOUNTS_REQD
41052. THEN
41053. generate (BLOCK_LABEL_PREFIX || 'BLC')
41054. THEN
41055. THEN
41056. PROC
**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. of the former, the proper macro is invoked depending on the
43004. block type being terminated. For BLOCK blocks, the block is
43005. simply terminated. */
43006. M I /* Temporary. */
43007.
43008. **CALL** TRACE_PRINTER ( ; 'BLEND')
43009. /* Prints macro name "BLEND" in mnote if tracing on. */
43010. if CURRENT_NEST_LEVEL > NESTING_LIMIT
43011. then
43012. **CALL** POP_OLD_BLOCK ( ; )
43013. else
43014. if CURRENT_NEST_LEVEL = 0
43015. then
43016. **NOTE** (6, 'STAN401 NO BLOCKS ACTIVE--"BLEND" IGNORED')
43017. else
43018. i := CURRENT_NEST_LEVEL
43019. if USER_NAME = ''
43020. then
43021. while I > 0 and BLOCK_NAME(I) # USER_NAME
43022. do /* (Termination: I is decremented—must eventually become
43023. < 0.) */
43024. I := I - 1
43025. od
43026. if I = 0
43027. then
43028. **NOTE** (6, 'STAN402 NO BLOCK ACTIVE NAMED ' USER_NAME '—"BLEND" IGNORED')
43029. **EXIT**
43030. **FIN**
43031. **DOCASE** BLOCK_TYPE(I) only
43032. of 'IF',
43033. **CALL** IF ( ; USER_NAME)
43034. **ESAC**
43035. case 'DO',
43036. **CALL** OD ( ; USER_NAME)
43037. **ESAC**
43038. case 'CASE',
43039. **CALL** ESAC ( ; USER_NAME)
43040. **ESAC**
43041. case 'DOCASE',
43042. **CALL** ESACOD ( ; USER_NAME)
43043. **ESAC**
43044. case 'PROC',
43045. **CALL** CORP ( ; USER_NAME, RETURN=RETURN, LINK=LINK, RESTORE=RESTORE, RC=RC)
43046. **ESAC**
43047. case 'BLOCK',
43048. **CALL** POP_OLD_BLOCK ( ; )
43049. **ESAC**
43050. **ESACOD**
43051. **END**
43052. endif
43053. endif
43054. endif
43055. endif
43056. endif
43057. /* (Lemma: If CURRENT_NEST_LEVEL > 0 and
43058. [USER_NAME = '' or = BLOCK_NAME(CURRENT_NEST_LEVEL)] at entry to
43059. BLEND, then CURRENT_NEST_LEVEL will be decremented by exactly
43060. one.) */
macro FINAL ( ; )
/* Insure all blocks are closed. Then if SAVETRACE_ON_FIRST_PROC,
   define label $LASTSAV to be PREV_SAVETRACE_AREA and equate
   PREV_SAVETRACE_PTR to 0. */
call TRACE_PRINTER ( ; 'FINAL')
/* Print macro name "FINAL" in mnote if tracing on. */
while CURRENT_NEST_LEVEL > 0
do if CURRENT_NEST_LEVEL > NESTING_LIMIT
   then
      mnote (8, 'STRC53C1 BLEND OF OUTSTANDING BLOCK ASSUMED')
      call BLEND ( ; )
   fi
/* (Lemma: If CURRENT_NEST_LEVEL > 0 and no BLEND
   operands are specified, CURRENT_NEST_LEVEL will be
   decremented by exactly
   one.) */
od
/* (Termination: CURRENT_NEST_LEVEL decreases monotonically
   and therefore must eventually become 0.) */
if SAVETRACE_ON_FIRST_PROC
   then
      if PREV_SAVETRACE_PTR = '$FIRSTSV'
         then /* No non-OS procs occurred: generate dummy area. */
            generate ('$LASTSAV DC OF'0',XF'FFFFFFF',A(0,0) DUMMY SAVEAREA')
            generate ('$LASTSAV EQU $FIRSTSV')
         else
            generate ('$LASTSAV EQU ' || PREV_SAVETRACE_AREA)
            generate (PREV_SAVETRACE_PTR || ' EQU 0')
      fi
   fi
end
"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. exit
55009. if
55010. call EXIT_FIND ( ; EXIT_TARGET)
55011. /* Set ULTIMATE_BRANCH_LABEL to point to end of block whose name
55012. is the argument and, if so needed, an exit label; if no such block,
55013. issue message and set ERROR_OCCURRED. */
55014. if - ERROR_OCCURRED
55015. then
55016. generate (USER_NAME || 'B ' || ULTIMATE_BRANCH_LABEL)
55017. fi
55018. fi
55019. end
"PROC" Macro -- 5 July 1973

8101. **MACRO** PROC (**USER_NAME**: **LINKAGE**, **ID**= **BASE**= **MCR**= **SAVE**= **DEBUG**, **XIT**=)
8102. /* Defines a procedure block. If **LINKAGE**=OS is specified, standard
8103. save area conventions are followed; otherwise a simple non-
8104. linked save area is provided. A base register is established
8105. (unless **BASE**=NONE is specified under **OS** **LINKAGE** or **BASE** is
8106. omitted on local PROCs). Register values upon entry are
8107. saved to allow restoring at CORP time. */
8108. **BIT**
8109. 8110. **BIT** FIRST_PROC, /* Indicates whether this is the first PROC macro coded
8111. in this assembly. */
8112. 8113. **BIT** FIRST_VALUE_KNOWN, /* Indicates whether the first **SAVE** operand was
8114. a self-defining term (or omitted) or if it was symbolic. */
8115. 8116. **BIT** OS_LINKAGE, /* Indicates whether
8117. **LINKAGE**=(OS,**-**) was
8118. entered. */
8119. 8120. **BIT** SPECIAL_PREFIX, /* Indicates whether the BLOCK_LABEL_PREFIX was
8121. changed to the special debugging form "$pe". */
8122. 8123. **BIT** USING13, /* Indicates whether the base register is SPR13. */
8124. 8125. **BIT** WORKREG_USED /* Indicates whether the value in WORKBEG was
8126. modified and its contents saved in register 0. */
8127. **CHAR** COMMA2, /* Contain "," and "H" respectively if a range of
8128. registers is to be saved, or the null string if a single register
8129. to be saved. Used to generate "STM" or "STM" instruction. */
8130. 8131. **INT** FIRST, LAST, /* First and last register in range to be saved. */
8132. 8133. **INT** LABEL, /* Any outstanding label waiting to be generated. */
8134. 8135. **INT** LOCAL_POINTER, **OS_POINTER**, /* Instruction segments to generate
8136. store instruction for proper save area. */
8137. 8138. **INT** PREVIOUS_DEBUG_VECTOR, /* Holds value of debug switches on entry to
8139. **PROC** macro for restoring on exit from CORP. */
8140. 8141. **INT** PROC_ID_BYTE, /* Value of hex proc number (PROC_COUNTER in hex)
8142. used in various debugging instructions. */
8143. 8144. **INT** SAVE_LENGTH, /* Length of save area (in words), except length of
8145. register part only for local PROCs. */
8146. 8147. **INT** SAVETYPE, /* Type of save area generated: FULL (savetrace), OSSAVE,
8148. NORML, NORMLHDR, TRUNC, TRUNCBDR, or NONE. */
8149. 8150. **INT** SAVEREG, /* Register (work or base) which is pointing at new
8151. save area before chaining. */
8152. 8153. **INT** WORKPROC, /* Register used for setting up linkage, etc. */
8154. **INT**
8155. 8156. **INT** OFFSET, /* Offset (in words) to either FIRST (if FIRST_VALUE_KNOWN),
8157. or to GPRO within save area. */
8158. 8159. **INT** OFFSET_TO_GPRO, /* Offset in words to GPRO within save area. */
8160. 8161. **INT** SAF, SAL /* Register number to go into first register word of
8162. save area; this, for example, could be 14 even though FIRST
8163. is a symbolic register of unknown value at macro expansion time.
8164. SAL is similar but for last register. */

C-71
PROC Macro -- 5 July 1973

81052. call TRACE_PRINTER ( ; "PROC")
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_BLOCK_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_GENERATE_AREA
81079. /* Generate proper save area depending on the variables SAVE_TYPE and
81080. SAVE_LENGTH set by PROC_BLOCK_SAVE and depending on the SAVE macro
81081. operands. */
81082. call PROC_MULTIBASE_GEN
81083. /* Generate definition of adwords 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. */
81103. end
**PROC** Macro --- 5 July 1973

```plaintext
@E1105. PROC PROC_SCAN_OPTIONS
@E1106. /* Validate LINKAGE and WORK= keywords; issue error messages and set
@E1107. OS_LINKAGE and WORKREG (the latter receiving either the register
@E1108. specified by the WORK= operand or some default). Process completely
@E1109. the DEBUG= and EXIT= keywords. Change BLOCK_LABEL_PREFIX if
@E1110. necessary from the normal "$nnn" to the special PROC prefix
@E1111. "$P2Y". Set FIRST_PROC to indicate whether this is the first
@E1112. PROC macro coded in this assembly. */

@E1113. OS_LINKAGE := (LINKAGE(1) ≠ 'OS' and ≠ '')
@E1114. if LINKAGE(1) ≠ 'OS' and ≠ ''
@E1115. then
@E1116.   annotate (5, 'STC6101 LINKAGE= ' || LINKAGE(1) ||
@E1117.     ' INVALID--"OS" ASSUMED')
@E1118. fi

@E1119. if LINKAGE(2) ≠ 'CSECT' and ≠ ''
@E120. then
@E121.   annotate (5, 'STC6102 SECOND LINKAGE OPERAND IGNORED')
@E122. fi

@E123. WORKREG := WORK,
@E124. if WORKREG ≠ 'NONE' or ≠ ''
@E125. then
@E126.   if OS_LINKAGE
@E127.     WORKREG := '2'
@E128. else
@E129.     WORKREG := '1'
@E130. fi
@E131. fi

@E132. EXIT_SEVERITY := EXIT
@E133. if EXIT_SEVERITY := EXIT
@E134. then
@E135.   annotate (5, 'EXIT')
@E136. else
@E137.   annotate (5, 'EXIT')
@E138. fi

@E139. FINISH_PROC := NOT_FIRST_PROC /* Brilliant, eh? */
@E140. NOT_FIRST_PROC := true
@E141. /* Note whether this is the first proc and set global NOT_FIRST_PROC
@E142. so we will be able to tell on later PROCs. Note that we are making
@E143. use here of the fact that global bit variables are initialized to
@E144. false. */

@E145. callPROC_DEBUG_SET
@E146. /* Set debugging switches according to DEBUG= operand, saving prior
@E147. values for restoring at CORP time in PREVIOUS_DEBUG_VECTOR. */

@E148. SPECIAL_PREFIX := (DEBUG_SAVETRACE_REQD or
@E149. DEBUG_PROCTRACE_REQD or
@E150. DEBUG_PROCCOUNTS_REQD and
@E151. PROC_COUNTER < 254)

@E152. if SPECIAL_PREFIX
@E153. then
@E154.   PROC_COUNTER := PROC_COUNTER + 1
@E155. /* Advance counter only on those procs which use it. */
@E156.   HEX_IN := PROC_COUNTER
@E157.   call HEX (1)
@E158. procID_BYTE := HEX_OUT
@E159. BLOCK_LABEL_PREFIX := "$P2Y" || PROC_ID_BYTE
@E160. /* Change labels on PROC to ease understanding of debug information. */
@E161. procID || "TRC8105 " || BLOCK_NAME(CURRENT_NEST_LEVEL) ||
@E162. DEBUG_ID= || PROC_ID_BYTE || "")
@E163. else
@E164. procID_BYTE := '00'
@E165. fi
```

---

C-73
PROC Macro -- 5 July 1973

81167. PROC_PROC_DEBUG_SET
81168. /* Save the previous DEBUG specifications so they can be restored at
81169. CORP time. Scan the DEBUG= suboperands setting the debug flags
81170. indicated. If GLOBAL is specified now set restore value to
81171. suppress it. */
81172. bit GLOBAL, /* Indicates whether "GLOBAL" has been found as an
81173. operand of DEBUG=. */
81174. SAVETRACE_VALUE, /* Set true if SAVETRACE is to be turned on; set
81175. false if SAVETRACE is to be turned off; else not set. */
81176. SAVETRACECHECK /* Set true if SAVETRACE to be set either on or
81177. off. */
81178. int I /* List suboperand index. */
81179. GLOBAL := false /* Has not yet been found. */
81180. PREVIOUS_DEBUGVECTOR :=
81181. DEBUG_BLOCKNAMES_REQD 1
81182. DEBUG_PROCNAMES_REQD 1
81183. DEBUG_LISTBLOCKS_REQD 1
81184. DEBUG_BLOCKCOUNTS_REQD 1
81185. DEBUG_PROCCOUNTS_REQD 1
81186. DEBUG_CORPVALUES_REQD 1
81187. DEBUG_SAVETRACE_REQD
81188. /* Save current value of debug switches. */
81189. I := 1
81190. while I <= 'DEBUG
81191. do /* Scan all operands. */
81192. when DEBUG(I)
81193. of
81194. case ('GLOBAL', 'GBL')
81195. GLOBAL := true
81196. esac
81197. case ('BLOCKNAMES', 'BN')
81198. DEBUG_BLOCKNAMES_REQD := true
81199. /* BLOCKNAMES causes the name of each block to be generated as an
81200. in-line character constant at the start of each block (of any type,
81201. not just BLOCK macros) for ease of locating code in dumps. */
81202. esac
81203. case ('NOBLOCKNAMES', 'NBN')
81204. DEBUG_BLOCKNAMES_REQD := false
81205. esac
81206. case ('PROCNAMES', 'PN')
81207. DEBUG_PROCNAMES_REQD := true
81208. /* PROCNAMES causes the name of each PROC to be generated as an
81209. in-line character constant at the start of the PROC for ease of
81210. locating code in dumps. */
81211. esac
81212. case ('NOPROCNAMES', 'NPN')
81213. DEBUG_PROCNAMES_REQD := false
81214. esac
81215. case ('LISTBLOCKS', 'LB')
81216. DEBUG_LISTBLOCKS_REQD := true
81217. /* LISTBLOCKS causes the name, number, and depth of each block to be
81218. generated in an mnote at the start and end of the block. */
81219. esac
81220. case ('NOLISTBLOCKS', 'NLB')
81221. DEBUG_LISTBLOCKS_REQD := false
81222. esac
81223. case ('BLOCKCOUNTS', 'BC')
81224. DEBUG_BLOCKCOUNTS_REQD := true
81225. /* BLOCKCOUNTS causes counters to be kept on the number of executions
81226. of all blocks. */
81227. esac
81228. case ('NOBLOCKCOUNTS', 'NBC')
81229. DEBUG_BLOCKCOUNTS_REQD := false
81230. esac
81231. case ('PROCCOUNTS', 'PC')
81232. DEBUG_PROCCOUNTS_REQD := true
81233. /* PROCCOUNTS causes counters to be kept on the number of executions
81234. of all PROC blocks. */
81235. esac
81236. case ('NODOCCOUNTS', 'NPC')
81237. DEBUG_PROCCOUNTS_REQD := false
81238. esac
81239. case ('CORPVALUES', 'CORP')
81240. DEBUG_CORPVALUES_REQD := false
81241. esac
81242. esac
81243. I := I + 1
81244. endwhile
81245. global DEBUG(t)
81246. if GLOBAL
81247. /* Set true if SAVETRACE is to be set either on or off. */
81248. SAVETRACE := true
81249. endif
81250. /* Set false if SAVETRACE is to be turned off; else not set. */
81251. SAVETRACECHECK := false
81252.
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. */

    esac

    case ('NOPROCTRACE', 'NPT')
    DEBUG_PROCTRACE_REQD := false

    esac

    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. */

    esac

    case ('NOCORPVALUES', 'NCV')
    DEBUG_CORPVALUES_REQD := false

    esac

    case ('SAVETRACE', 'ST')
    SAVETRACE_VALUE := true
    SAVETRACE_CHECK := true
    /* Note that savetrace has been specified. */

    esac

    case ('NOSAVETRACE', 'NST')
    SAVETRACE_VALUE := false
    SAVETRACE_CHECK := true

    esac

    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. */

    SAVETRACE_CHECK := true

    esac

    /* The following operands are provided for the debugging of the
     structure macros themselves. They are not automatically restored
     at CORP time. */

    case ('MACRONAMES', 'MN')
    DEBUG_MACRONAMES_REQD := true
    /* MACRONAMES causes the name of each structure macro (including inner
     macros) to be printed in an mnote whenever invoked. */

    esac

    case ('NOMACRONAMES', 'NMN')
    DEBUG_MACRONAMES_REQD := false

    esac

    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. */

    esac

    case ('NODEBUGMACROS', 'NDM')
    /* Note that debugmacros has been specified. */

    esac

    case ('STRC8104 DEBUG='
    I := I + 1 /* Go on to next suboperand. */
    /* Termination: I is incremented, N'DEBUG is fixed in loop;
    I must eventually exceed N'DEBUG. */

    od

    /* (Termination: I is incremented, N'DEBUG is fixed in loop;
    I must eventually exceed N'DEBUG. */

    esac
"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.                     note (4, 'STEC8103 WARNING--SAVETRACE REQUIRES "FINAL" MACRO')
81321.                     else
81322.                       note (8, 'STEC8106 SAVETRACE REQUIRES FIRST PROC TO BE LINKAGE=OS')
81323.                   fi
81324. else /* Not first PROC. */
81325.   if SAVETRACE_CN_FIRST_PROC
81326.     then /* SAVETRACE is being resumed. */
81327.       DEBUG_SAVETRACE_REQD := true
81328.     else
81329.       note (8, 'STEC8105 SAVETRACE MUST BE SPECIFIED ON FIRST PROC')
81330.     fi
81331. fi
81332. else
81333.   DEBUG_SAVETRACE_REQD := false
81334. fi
81335. if GLOBAL
81336.   then
81337.     POSITION_DEBUG_VECTOR := ''
81338.   fi /* Null value suppresses restore by GERP. */
81339. fi
81340. corp
"PROC" Macro -- 5 July 1973

8133.  DEFINE PROC_HEADER
8134.  /t* Generates a "CSECT" and "USING *,15" if LINKAGE=(**,CSECT)
8135.  specified. If LINKAGE=OS,*** specified, generate inline ID
8136.  similarly to JAS SAVE macro: for non-OS linkage, do the same if
8137.  OBJSPEC(*,PRIVATE) or ID= specified, but omit ID-length-count field.
8138.  Return any label generated as branch target in LABEL. */

8139.  if (LINKAGE=(**,CSECT) or "PRIVATE") or ID= specified. If LINKAGE=(**,CSECT)
8140.  specified, generate inline ID similar to LABEL.
8141.  For non-OS linkage, do the same if ID= specified, but omit ID-length-count field.
8142.  Return a label generated as branch target in LABEL. */

8143.  char SECT, /* Private name (or "$PRIVATE") for default TID constant.
8144.  */
8145.  QUOTE, /* null, for generating ID.
8146.  */
8147.  TARGET, /* Temporary. */
8148.  LENST if OS_LINKAGE or ID= specified, but omit ID-length-count field.
8149.  Return a label generated as branch target in LABEL. */

8150.  if LINKAGE(**,CSECT) then
8151.  TARGET := BLOCK_LABEL(PREFIX 'A',CSECT)
8152.  else
8153.  LABEL := 'USING *,15'
8154.  space
8155.  fi

8156.  if ((OS_LINKAGE or ID= '') and ID= 'NONE') then
8157.  TARGET := $PRIVATE
8158.  else
8159.  SECT := BLOCK_NAME(CURRENT_NEST_LEVEL)
8160.  LENST := 8
8161.  fi

8162.  if OS_LINKAGE then
8163.  SECTION := 'USING *
8164.  LABEL := BLOCK_NAME(CURRENT_NEST_LEVEL)
8165.  fi

8166.  if OS_LINKAGE then
8167.  SECT := 'PRIVATE' /* Default name if none specified. */
8168.  else
8169.  SECT := USER_NAME
8170. fi

8171.  if ID= then
8172.  QUOTE := "" /* ID already contains surrounding quotes. */
8173.  QUOTE := ' "" /* ID already contains surrounding quotes. */
8174.  else
8175.  QUOTE := QUOTE ++ ""
8176.  fi

8177.  if OS_LINKAGE then
8178.  fi

8179.  if OS_LINKAGE then
8180.  LENST := ((K'ID/2)*2) + 1 /* Round up to odd number. */
8181.  fi

8182.  if OS_LINKAGE then
8183.  len := ((LENGTH/2)*2) + 1 /* Round up to odd number. */
8184.  len := ((LENGTH/2)*2) + 1 /* Round up to odd number. */
8185.  len := ((LENGTH/2)*2) + 1 /* Round up to odd number. */
8186.  len := ((LENGTH/2)*2) + 1 /* Round up to odd number. */
8187.  if OS_LINKAGE then
8188.  len := ((K'ID/2)*2) + 1 /* Round up to odd number. */
8189.  len := ((K'ID/2)*2) + 1 /* Round up to odd number. */
8190.  len := ((K'ID/2)*2) + 1 /* Round up to odd number. */
8191.  if OS_LINKAGE then
8192.  len := ((K'ID/2)*2) + 1 /* Round up to odd number. */
8193.  len := ((K'ID/2)*2) + 1 /* Round up to odd number. */
8194.  len := ((K'ID/2)*2) + 1 /* Round up to odd number. */
8195.  if OS_LINKAGE then
8196.  len := ((K'ID/2)*2) + 1 /* Round up to odd number. */
8197.  len := ((K'ID/2)*2) + 1 /* Round up to odd number. */

C-17
/* If SAVE=NOE and doing SAVETRACE generate store of all registers. */

PROC_RES_SAV

else /* If SAVE=NOE and no SAVETRACE, do nothing. For other than */

SAVE=NOE, extract from SAVE=NOE register list to be saved.

SAVE=NOE. Generate instruction to store registers. */

FIRST, LAST :=

OFFSET, SAF, SAL, OFFSET_TO_GPRO := 0

/* Initialize save request information variables. */

if SAVE = 'NOE' then

if DEBUG

SAVETRACE_REQ =

then

SAVETYPE := 'FULL'

SAVE LENGTH := 15

gl9aQte (LABEL 'STM 14,12,' || BLOCK_LABEL_PREFIX || 'SV+12')

LABEL :=

else

call PROC_SAVESAVE_TYPE

/* Puts type of save area to be generated into SAVE_TYPE. Set */

SAVE LENGTH with length (in words) of save area (except only length

of register part for non-OS_LINKAGE areas). Offset in save area

in words) of register 0 is put into OFFSET_TO_GPRO. Offset of

either FIRST (if FIRST_VALUE_KNOWN) or else of register 0 is

generated into OFFSET. Also set LOCAL_POINTER to ' ' and OS_POINTER to

' (13) if OS_LINKAGE, else LOCAL_POINTER to

' ' || BLOCK_LABEL_PREFIX || 'SV†' and OS_POINTER to ' '. Thus

LOCAL_POINTER || OFFSET || ' ' || OS_POINTER refers to the given

offset in the proper area. */

if SAVE_TYPE = 'FULL'

then

savearea (LABEL 'STM 14,12,' || BLOCK_LABEL_PREFIX || 'SV†')

/* Save all registers on FULL (savetrace-required) save area. */

else

if FIRST_VALUE_KNOWN

then

i := OFFSET + a /* Calculate offset in bytes. */

function (LABEL 'STM || MULT || ' || FIRST || ', ||

LAST || COMMA2 || LOCAL_POINTER || ' || OS_POINTER)

else

function (LABEL || ' STM || MULT || ' || FIRST || ', ||

LAST || COMMA2 || LOCAL_POINTER || ' ||offset ||' ||

-(' || FIRST || '+2)/(16*16)))*g || OS_POINTER)

fi

fi

LABEL :=

endif
"PROC" Macro -- 5 July 1973

81477. PROC_SET_SAVE_INFO
81478. /* Collect following save request information. Put character string
81479. name of first register to be saved in FIRST, last in LAST, "I" into
81480. MULT, and "I" into COMMA2. If only a single register to be saved,
81481. LAST, MULT, and COMMA2 get null strings. Register number which could
81482. go into first word of register part of save area goes into SAF, last
81483. into SAL. (These, for example, could be 14 and 12 while FIRST and
81484. LAST are symbolic register designations of unknown value at macro
81485. expansion time.) Set FIRST_VALUE_KNOWN if FIRST is not symbolic. */
81486.
81487. FIRST_VALUE_KNOWN := true
81488. MULT := 'I'
81489. COMMA2 := '
81490. /* Assumed values. */
81491. if T'SAVE(1) = 'O' /* At least first suboperand is omitted. */
81492. then
81493. FIRST := '14' SAF := 14
81494. LAST := '12' SAL := 12
81495. /* Default is to save all registers 14 through 12. */
81496. else
81497. if T'SAVE(1) = 'N' /* Self-defining term. */
81498. then
81499. SAF := SAVE(1) /* Store it as a number. */
81500. FIRST := SAF /* Convert it back to a string (done for non-decimal
81501. self-defining terms). */
81502. else /* It must be symbolic. */
81503. FIRST := SAVE(1) /* Store it as a character string. */
81504. SAF := 14 /* Just say first of save area is register 14. */
81505. FIRST_VALUE_KNOWN := false
81506. fi
81507. el
81508. if T'SAVE(2) = 'O' /* Second suboperand is omitted. */
81509. then
81510. LAST, MULT, COMMA2 := '
81511. if FIRST_VALUE_KNOWN
81512. then
81513. SAL := SAF /* Last register is same as first. */
81514. else
81515. SAL := 12 /* Last register is 12. */
81516. fi
81517. else
81518. if T'SAVE(2) = 'N' /* Self-defined. */
81519. then
81520. SAL := SAVE(2) /* Store it as a number. */
81521. LAST := SAL /* Convert it back to a string. */
81522. else
81523. LAST := SAVE(2) /* Store it as a character string. */
81524. SAL := 12 /* Just say last of save area is register 12. */
81525. fi
81526. fi
81527. fi
81528. end
"PROC" Macro -- 5 July 1973

81530. PROC DECIDE_SAVE_TYPE
81531. /* Set SAVE_TYPE with type of save area to be generated: NONE, OSSAVE,
81532. TRUNC, TRUNCADR, 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
81541. if OS_LINKAGE
81542. then
81543. if I := 13
81544. then
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
81560. /* Not OS LINKAGE. */
81561. LOCAL_POINTER := BLOCK_LABEL_PREFIX || 'SV+'
81562. if DEBUG_SAVETRACE_REQD
81563. then
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. else
81569. I := SAL - ((SAL+2)/16*16) - SAF + ((SAF+2)/16*16) + 1
81570. /* Convert calculated length to character string. */
81571. if SAF = 14
81572. then
81573. /* SAVE(1) was omitted, specified as 14, or symbolic. */
81574. if DEBUG_PROCCOUNTS_REQD
81575. then
81576. SAVE_TYPE := 'NORMLHDR' /* First register word is 14. */
81577. OFFSET_TO_GPRO, OFFSET := 3
81578. else
81579. SAVE_TYPE := 'NORML'
81580. OFFSET_TO_GPRO, OFFSET := 2
81581. fi
81582. else
81583. /* Save area is to start after 14: a truncated area. */
81584. if DEBUG_PROCCOUNTS_REQD
81585. then
81586. /* Header included for count. */
81587. SAVE_TYPE := 'TRUNCNDR'
81588. OFFSET := 1 /* To SAVE(1). */
81589. else
81590. /* SAVE_TYPE := 'TRUNC'
81591. OFFSET := 0 /* To SAVE(1). */
81592. fi
81593. OFFSET_TO_GPRO := OFFSET - SAF
81594. if SAF > 13
81595. then
81596. OFFSET_TO_GPRO := OFFSET_TO_GPRO + 16
81597. fi
81598. else
81599. if FIRST_VALUE_KNOWN /* SAVE(1) was not symbolic */ and
81600. then
81601. /* Adjust OFFSET from giving offset to GPRO to give offset to SAF. */
81602. OFFSET := OFFSET_TO_GPRO + SAF
81603. if SAF > 13
81604. then
81605. OFFSET := OFFSET - 16
81606. fi
81607. fi
81608. fi
81609. fi

*/
```
PROC PROC_ESTABLISH_BASE

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

char
  BASEREG /* Base 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
81624. INLINESAVAREA := (SAVE = 'NONE' and SAVE(3) = 0)
81625. if BASEREG = '13' and (INLINESAVAREA and OS_LINKAGE)
81626. then
81627. emit (3, 'STCR109 REGISTER 13 INVALID--IGNORED')
81628. BASEREG :=
81629. fi
81630. if BASEREG = ''
81631. then /* In base register specified. */
81632. if INLINESAVAREA and OS_LINKAGE
81633. then
81634. BASEREG := WORKREG
81635. /* We will load the base value first into the work register, then copy the value to register 13 after we finish all linkage. */
81636. J := 1
81637. if J = 1
81638. BASEREG :=
81639. else
81640. BASEREG := '12'
81641. fi
81642. fi
81643. I := 0
81644. I := 2
81645. while I < N'BASE
81646. do
81647. I := 2
81648. I := 2
81649. od
81650. if BASE(1) = '13'
81651. then
81652. if BASE(1) = '13'
81653. then
81654. BASE := (3, 'STCR109 REGISTER 13 INVALID--IGNORED')
81655. emit (BASE(1) = '1', 1) BASE := (LABEL || I || BASE(1) || '1', 1)
81656. BLOCK_LABEL_PREFIX || 'MBR-' || I)
81657. LABEL :=
81658. fi
81659. I := I + 1
81660. J := J + 1
81661. od /* (Termination: I is incremented in loop, N'BASE is fixed; I must eventually exceed N'BASE. */
81662. if INLINESAVAREA
81663. then
81664. if OS_LINKAGE
81665. then
81666. if OS_LINKAGE
81667. then
81668. /* Advance to fullword boundary; outstanding label can wait for next instruction. */
81669. fi
81670. fi
81671. fi
81672. if I = J + 1
81673. then
81674. emit (I, 'USING *, 1, BASEREG)
81675. fi
81676. if N'BASE > 1
81677. emit (I, 'USING *, 1, BASEREG)
81678. fi
81679. if N'BASE > 1
81680. emit (I, 'USING *, 1, BASEREG)
81681. for J = 0 to J = 4096
81682. do
81683. emit (I, 'USING *, 1, BASEREG)
81684. fi /* (Termination: Same proof as above. */
81685. fi
81686. fi
PROC Macro -- 5 July 1971

81700. PROC GEN SAVEAIREA
81701. /* Generate appropriate save area according to SAVE_TYPE, SAVE_LENGTH, and the SAVE suboperands. */
81702. if SAVE_TYPE = 'OSSAVE'
81703. then
81704. call PROC_GEN_OSSAVE_AREA
81705. /* 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. */
81706. else
81707. if (SAVE # 'NONE' and SAVE_TYPE # 'NONE') or EBJG_SAVETRACE_REQD
81708. then
81709. call PROC_GEN_LOCAL_SAVEAREA
81710. /* Generate local PROC save area according to SAVE_TYPE and SAVE_LENGTH and, if SAVETRACE requested, provide static save area chaining. */
81711. fi
81712. fi

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. SAVETRACE requested. */
81725. call PROC_DEF132DNEW_OSSAVE
81726. /* Generate linkage, out-of-line, or dynamic save area and point to
81727. it with BASEREG or WORKREG; put register name in SAVETRACE. */
81729. if DEBUG_SAVETRACE_REQD and FIRST_PROC
81730. then /* Static chain of local save areas must be linked to OS save areas. */
81731. generate (LABEL II 'ST 13,$FIRSTSV+4')
81732. LABEL := I-
81733. generate ("MVC 8(4,13),A(SFIRSTSV)")
81734. generate ("L 13,$LASTSAV")
81735. PREV_SAVETRACE_PTR := 'FIRSTSV'
81736. PREV_SAVETRACE_AREA := '0'
81737. fi
81738. generate (LABEL II 'ST 13,SAVEREG |(13))
81739. LABEL := I-
81740. generate ("ST 13,SAVEREG")
81741. if USING13
81742. then /* 13 now loaded--issue USING. */
81743. generate ("USING || BLOCK_LABEL_PREFIX II 'ISV,13")
81745. fi
81746. if DEBUG_PROCTRACE_REQD and FIRST_PROC and SAVE(3) ||
81747. then /* Save pointer to PROC trace vector in word 1 of OS save area. */
81749. if WORK = 'ONE' and ~ WORKREG_USED
81750. then
81751. generate ("LR O,WORKREG")
81752. fi
81753. generate ("LA || WORKREG || 'STRACE")
81754. WORKREG.Ptr := base
81755. fi
81757. end
PROCDIVF macros -- 5 July 1973

81759. PROC DIVF_DEFINE

81760. /* Generate ialine, out-of-line, or dynamic save area for
81761. OS_LIN and point to it with the WORKREG or BASEREG. Put base of
81762. save area register in SAVEREG. */
81763. PROC_DEFINE

81764. char
81765. X
81766. X := "SAVEREG := 433REG /* Assumed. */
81767. if SAVE(3) = 'JLNAME'
81768. then
81769. generate (' LABEL := ')
81770. LABEL := ')
81771. if WORK = 'NONE'
81772. then
81773. generate (' LR 0,1)
81774. if LAST # '13' or LAST $ '12'
81775. then
81776. generate (' CNOP 0,4)
81777. LABEL := BLOCK LABEL
81778. BAL := 'I WORKREG || 'I LABEL
81779. if DBJ_IDemonic
81800. then
81801. create (X II ' DC A($TRACE),(' SAVE_LENGTH II '1')p"')
81802. /* Generate inline save area with first word pointing to trace
81803. vector. */
81804. else
81805. generate (' LR 0,1)
81806. call PROC_DEFINI
81807. /* Area has been generated and address is in SAVEREG register. */

81808. " /* Temporary */
81809. X := "
81810. SAVEREG := WORKREG /* Assumed */
81811. if SAVE(3) = 'JLNAME'
81812. then
81813. generate (LABEL := ')
81814. LABEL := ')
81815. if WORK = 'NONE'
81816. then
81817. generate (' LR 0,1)
81818. else
81819. /* User-supplied out-of-line save area. */
81820. if WORK = 'NONE' and WORKREG_USED
81821. then
81822. generate (LABEL := ')
81823. LABEL := ')

81824. /* Generate ialine save area. */
81825. then
81826. generate (LABEL := ')
81827. LABEL := ')

81828. else
81829. /* Generate (LABEL := ')
81830. LABEL := ')

81831. end.
**PROC** Macro

```
 Macro PROC(fn, LOCAL_SAVEAREA)

 /* Generate local PROC save area according to SAVE_TYPE and SAVE_LENGTH
 and, if SAVE_TRACE requested, provide static save area chaining. */
 char FWDPTR /* Name used for next save area. */

 if LABEL = ''
 then
 LABEL := BLOCK_LABEL_PREFIX || 'DD'
 generate ('B' || LABEL)

 generate (BLOCK_LABEL_PREFIX || 'SV DS 00')

 if SAVE_TYPE = 'FULL' or SAVE_TYPE[5,3] = 'HDR'
 then /* Label one should contain PROC count and ID byte. */
 generate ('DC X'PF' || PROC_ID_BYTE || '0000')

 if SAVE_TYPE = 'FULL'
 then
 FWDPTR := BLOCK_LABEL_PREFIX || 'NXT'
 generate ('DC A' || PREV_SAVETRACE_PTR || 'EQU ' ||
 PREV_SAVETRACE_PTR || 'SV')
 generate ('DC A' || PREV_SAVETRACE_AREA || '','')
 generate ('DC P2 || ')'

 PREV_SAVETRACE_PTR := FWDPTR
 /* Save label used as forward pointer. */
 PREV_SAVETRACE_AREA := BLOCK_LABEL_PREFIX || 'SV'
 /* Save name of this save area. */

 generate ('DC ' || SAVE_LENGTH || 'P0')
 call PROC_NLPIRAS2,2N
```

C-86
/* PROC_DEBUG_STJF */

81865. 
81866. PROC_DEBUG_STJF
81867. /* Generate trace and count code for debugging. */
81868. bit PCT_DEBUGGED_VECTOR /* Indicates whether -PCT labeled halfword
81869. which holds PROC counter was generated following the trace vector. */
81870. CHAS COUNT_SPOT /* Suffix of label for PROC counter. */

81871. if DEBUG_PROC_COUNT_REQD
81872. then
81873. if TRACER Vector Generated
81874. then /* Previously generated TRACER vector must be updated. */
81875. if WORK = 'NONE' and WORKREG_used
81876. then
generate (LABEL II ' LR 0,' II WORKREG)
81877. LABEL := '
81878. fi
81879. fi
81880. fi
81881. fi
81882. bit PCT GENNED WITH VECTOR
81883. /* Indicates whether -PCT labeled halfword
81884. which holds PROC counter was generated following the trace vector. */
81885. fi
81886. PCT GENNED WITH VECTOR := 'PCT'
81887. else /* Trace vector must be generated. */
81888. if LABEL = '
81889. then
81890. fi
81891. fi
81892. if DEBUG_PROC_COUNTS_REQD or DEBUG_BLOCKCOUNTS_REQD and
81893. FIRST PROC
81894. then /* PROC counter must be generated since word one of first save area
81895. points to proc trace vector, so we can't keep count there. */
81896. fi
81897. fi
81898. fi
81899. fi
81900. fi
81901. fi
81902. fi
81903. fi
81904. fi
81905. fi
81906. fi
81907. fi
81908. fi
81909. if DEBUG_PROC_COUNTS_REQD or DEBUG_BLOCKCOUNTS_REQD
81910. then /* We must update the count. */
81911. if OS_LINKAGE or (SAVE = 'NONE' and DEBUG_SAVE_TRACE_REQD)
81912. then /* Count will be in BLOCK_LABEL_PREFIX II 'PCT'. */
81913. fi
81914. fi
81915. fi
81916. generate (' NOP 0')
81917. generate (BLOCK_LABEL_PREFIX II 'PCT EQU 256PROC COUNT')
81918. fi
81919. fi
81920. fi
81921. fi
81922. fi
81923. fi
81924. fi
81925. fi
81926. if WORK = 'NONE' and WORKREG_used
81927. then
81928. generate (LABEL II ' LR 0,' II WORKREG)
81929. LABEL := '
81930. fi
81931. fi
81932. fi
81933. generate (LABEL II ' LR 0,' II WORKREG)
81934. BLOCK_LABEL_PREFIX II 'COUNT Spot')
81935. fi
81936. fi
81937. fi
81938. fi
81939. fi
81940. fi
81941. fi
81942. fi
81943. fi
81944. fi

...
`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. `OPERAND1(CURRENT_NEST_LEVEL) := FIRST`  
81950. `OPERAND2(CURRENT_NEST_LEVEL) := LAST`  
81951. `OPERAND3(CURRENT_NEST_LEVEL) := SAFE_LENGTH`  
81952. `OPERAND4(CURRENT_NEST_LEVEL) := PREVIOUS_DEBUG_VECTOR`  
81953. `I := OFFSET_TO_CORP + 50`  
81954. `/* Bias value by 50 and convert to two-digit character string. */`  
81955. `INFORMATION(CURRENT_NEST_LEVEL) :=`  
81956. `I | GS_LINKAGE | I (SAVE(3) = 'DYNAN') | FIRST_VALUEKNOWN |`  
81957. `PROC информации | SPECIAL_PREFIX`  
81958. `CORE`
PROC Macro

PROC MULTIBASE

/* This proc generates any multiple base register adcons if
   needed and not yet generated and notes that such adcons have
   been generated. */

int I, J /* Preparation. */
CALL X

if MULTIBASE
  then /* Multibase adcons required but not yet generated. */
    if LABEL = '
      then /* Search around adcons. */
        LABEL := BLOCK_LABEL_PREFIX || 'FF'
        generate ('B' || LABEL)
      fi
    fi
  fi

I := 2
J := 4096
X := BLOCK_LABEL_PREFIX || 'MBR'

while I < N'BASE do
  generate (X || DC(' || BLOCK_LABEL_PREFIX || 'MBR' ||
    X || 'B')')
  X := ''
  I := I + 1
  J := J + 4096
od /* (Termination: I is incremented during loop, N'BASE is
  fixed; I must eventually exceed N'BASE.) */
MULTIBASE := false
83001.  MACRO  
83002.  /* Defines the end of a procedure block. The register or registers  
83003.  indicated by RESTORE are restored with the exception of those listed  
83004.  in RETURN. If RESTORE is omitted, all saved registers are restored  
83005.  (except those in the RETURN list). The return code is set from the  
83006.  REGISTER and return is made to the address specified by the  
83007.  LINK= operand, unless LINK=NOM is specified. */  
83008.  
83009.  /* (Ground rule: No CORP proc modifies CURRENT_NEST_LEVEL.  
83010.  This can be shown via the cross-reference listing.) */  
83011.  
83012.  int  
83013.  FIRST_SAVERS_REG, /* Register number which may be placed into the  
83014.  first word of the save area. This may be, for example, 14 when  
83015.  the first register saved is some symbolic of unknown value. */  
83016.  OFFSET_TO_GPPO /* Offset in save area (in words) to the storage  
83017.  place for GPRO. This may be positive or negative. */  
83018.  bit  
83019.  ANY_SAVERS, /* Indicates whether any registers were saved in  
83020.  this proc. */  
83021.  DYNAMIC_SAVERS, /* Indicates whether SAVE=(*), DYN,... */  
83022.  was coded on PROC. */  
83023.  CK_AREA_REG, /* Indicates whether CK area is needed. */  
83024.  FIRST_REG_SAVERS, /* Indicates whether FIRST is other than a  
83025.  symbolic. */  
83026.  FIRST_REGISTER_KNOW, /* Indicates whether first register to be  
83027.  restored (in REST1) is other than symbolic. */  
83028.  OS_LINK = /* Indicates whether LINKAGE=(OS,...) was coded  
83029.  on PROC. */  
83030.  
83031.  char  
83032.  LABEL, /* Aiy outstanding label waiting to be generated. */  
83033.  GPRO_OFFSET_SING, /* GPRO_OFFSET to GPRO is character form. */  
83034.  FIRST_REG_SAVERS, LAST_REG_SAVERS, /* First and last registers saved at PROC time. */  
83035.  MULTI, COMP, /* Holds either a '*' and ',,' respectively or else nulls  
83036.  to allow generation of either a "LM" or "L" instruction. */  
83037.  LOCAL_PTRM, OS_PTRM, RESTORE_AREA, /* Instruction segments to generate load instructions from proper  
83038.  save area. */  
83039.  RC_REG, /* Register holding return code before restoring of  
83040.  registers. */  
83041.  LAST_SAVERS, /* Length of save area. */  
83042.  FIRST_REG_SAVERS, LAST_REG_SAVERS, /* First and last register to be restored. */  
83043.  procTo год, /* One-byte hex number used as identifier of current  
83044.  PROC in traces and the like. */  
83045.  PREVIOUS_DEBUG_VECTOR /* Value of debug switches (packed) before  
83046.  encountering this PROC or [if DEBUG=(...,GLOBAL) specified]  
83047.  null. */  
83048.  
83049.  CALL TRACE PRINTER (;
83050.  "CORP")  
83051.  /* Print macro name "CORP" in mnote if tracing on. */  
83052.  
83053.  if CURRENT_NESTLEVEL > NESTING_LIMIT  
83054.  label  
83055.  procTo год BLOCK ( ; )  
83056.  endif  
83057.  call VERIFY_EXPR ( ; "PROC", procTo год )  
83058.  /* Verifies current block has the name specified in the procTo год  
83059.  macro (it any) and that it is in a procTo год block.  
83060.  Various errors receive messages and either intermediate blocks are  
83061.  replaced as a fixup or ERROR_OCCURRED is set.  
83062.  (Errors: if CURRENT_NESTLEVEL > 0 and  
83063.  procTo год = "" or = BLOCK_NAME (CURRENT_NESTLEVEL) ) and  
83064.  BLOCK_TYPE (CURRENT_NESTLEVEL) = "PROC", then  
83065.  ERROR_OCCURED will be set false and CURRENT_NESTLEVEL will not  
83066.  be modified. */  
83067.  if ERROR_OCCURED  
83068.  then  
83069.  endif  
83070.  label  
83071.  /* Generate label at first opportunity. */
COP Macro -- 6 July 1973

83074. call CORP_GET_PROC_INFO
83075. /* Get info saved at PROC macro: FIRST_REG_SAVED, LAST_REG_SAVED,
83076. OS_LINKAGE, FIRST_SAVEAREA_REG, FIRST_VALUEKNOWN, DYNAMIC_SAVEAREA,
83077. SAVE_LENGTH, OFFSET_REG, DS_SAVE, PREVIOUS_DEBUG_VECTOR, PROC_TO_CPP,
83078. SAVEMODE, BLOCK_LABEL_PREFIX, and ANY_REGS_SAVED. */
83079. call CORP_GETRestore_Range
83080. /* Set RESIBZ_AREA to 'SV'. Set REST1 and REST2 to
83081. operand; or, if omitted, then to FIRST_ and LASTREGS_SAVED. */
83082. call CORP_SET_DEBUG_ENVIRONMENT
83083. /* If an X{Y} 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_SAVEAREA
83091. label call CORP_SAVE_DYN_SAVEAREA
83092. /* Issue FREE_MALLOC for dynamic core. */
83093. fi
83094. else
83095. call CORP_DEBUGGING_STORES
83096. /* If DEBUG_CORPVALUES_REQD, copy registers into CRP save area. If
83097. CORVALUES 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_METHOD_CODE
83101. /* If RC= value (or implied zero), load it to GPR15, except that if it
83102. is in in a register other than 15, leave it in that register. */
83103. call CORP_FREE_DYN_SAVEAREA
83104. /* Issue FREE_MALLOC for dynamic core. */
83105. call CORP_SAVEReturning_REGS
83106. /* If any registers are to be restored, do the following: for the
83107. register containing the return code and all those listed in
83108. RETURN, store each register into the appropriate word of the
83109. save area from which the ultimate LN instruction will be issued.
83110. Also set OS POINTER and LOCAL_POINTER to reflect proper save
83111. area. */
83112. if OS_LINKAGE and DEBUG_SAVETRACE_REQD
83113. then
83114. generate (LABEL II 'IVI ' II BLOCK_LABEL_PREFIX II ''SV''X'FF')
83115. LABEL := ''
83116. fi
83117. call CORP_RESTORE_REGS
83118. /* Restore REST1 through REST2 from proper save area if saved. */
83119. if OS_LINKAGE
83213. then generate LABEL II '' IVI '12[13]X'PF''
83214. LABEL := ''
83215. fi
83216. if LINK <> 'NONE'
83217. then generate (LABEL II '3R ' II LINK)
83218. LABEL := ''
83219. fi
83220. label call CORP_SAVE_DCP__ock_areas
83221. /* Generate the CRP and BCK save areas. */
83222. if LABEL <> ''
83223. label generate (LABEL II 'OS OR')
83224. LABEL := ''
83225. fi
83226. label call POP_OLD_BLOCK ( )
83227. /* Delete PROC block from the stack. (Lemma: POP_OLD_BLOCK
83228. decreases CURRENT_NEST_LEVEL by exactly one.) */
83229. label call CORP_RESTORE_DEBUG_ENVIRONMENT
83230. /* Restore value of debugging switches from PREVIOUS_DEBUG_VECTOR. */
83231. label /* (Lemma: If CURRENT_NESTLEVEL > 0 and
83232. [PROC_NAME = 'IVI or = BLOCK_NAME(CURRENT_NESTLEVEL)] and
83233. BLOCK_FLAGS(CURRENT_NESTLEVEL) = "PROC", then
83234. CURRENT_NESTLEVEL will be decremented by exactly one.) */

C-91
"CORP" MACRO -- 6 July 1973

83148. PROC CORP_GET_PROC_INFO
83149. /* Get info saved by PROC macro. */
83150. char X
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. /* In string conversion, absolute value is taken; restore sign. */
83160. else
83161. GPRO_OFFSET_STRING := "+" " " OFFSET_TO_GPRO
83162. fi
83163. OS_LINKAGE := X[3,1]
83164. DYNAMIC_SAVEMARK := X[4,1]
83165. FIRST_VALUE_KNOWN := X[5,1]
83166. SAFE_LENGTH := OPERAND3(CURRENT_NEST_LEVEL)
83167. PREVIOUS/logout_VECTOR := OPERAND4(CURRENT_NEST_LEVEL)
83168. ANY_REGS_SAVED := (FIRST_REG_SAVED X """)
83169. PROC_ID_BYTE := X[6,2]
83170. if X[8,1]
83171. /* Special PROC prefix. */
83172. BLOCK_LABEL_PREFIX := "$P" " " PROC_ID_BYTE
83173. else /* Standard prefix. */
83174. BLOCK_LABEL_PREFIX := "$" " " BLOCK_NUMBER(CURRENT_NEST_LEVEL)
83175. fi
83176. if FIRST_VALUE_KNOWN
83177. then
83178. FIRST_SAVEAREA_REG := FIRST_REG_SAVED /* Convert to integer. */
83179. else
83180. FIRST_SAVEAREA_REG := 14
83181. fi
83182. /* FIRST_SAVEAREA_REG is similar to the variable SAP of PROC. */
83183. C-92
/* Set REST1 and REST2 to RESTORE= operands if present, else to FIRST_ 
and LAST_250_SAVED. Set RESTORE_AREA to "SV". Also set MULT and 
COMMA2 to proper values. */

/* Assume: */
FEST1 := FIRST_REG_SAVED
REST2 := LAST_REG_SAVED
MULT := 'M'
COMMA2 := ', '
FIRST_REG_VALUE_KNOWN := FIRST_VALUE_KNOWN

/* Now find out. */
if ANY_REGS_SAVED
if RESTORE = ""
then 
  REIN (3, ' stools NO REGISTERS SAVED--RESTORE IGNORED')
else 
  REST1 := RESTORE(1)
  REST2 := RESTORE(2)
  FIRST_REG_VALUE_KNOWN := (RESTORE(1) = 'N')

  /* True iff first suboperand is a self-defining term. */

  if REST2 = ""
    then 
      MULT, COMMA2 := ""
  else 
    RESTORE_AREA := "SV"
else 
  ""
endif
endif
C-94
"CORP" Macro -- 6 July 1973

83259. PROC CORP_RESTORE_LSEG13
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_LINKASG is true. */
83264. if DYNAMIC_SAVEAREA
83265. then
83266. generate (LABEL || ' LR 1,13')
83267. fi
83268. if SAVE_TRACE_OA_FLAGS_PROC and PROC_ID_BYTE = '01'
83269. then
83270. generate (LABEL || ' L 13,SPIBST5F+4')
83271. else
83272. generate (LABEL || ' L 13,4(13)')
83273. fi
83274. LABEL := **
83275. CORR
"CORP" Macro -- 6 July 1973

```
CORP

83278. proc CORP_SET_RETURN_CODE
83279.     /* If RC=value (or implied zero), load value into GPR15, but nop if
83280.     RC=(reg). Note in RC_REG what register (if any) contains RC at
83281.     exit */
83282.     RC_REG := '1' /* Indicate no return code. */
83284.     if RC = '1'
83285.         then
83286.             if OS_LINKAGE
83287.                 then
83288.                     generate (LABEL || 'SR 15,15')
83289.                     /* Clear 15 for normal OS return. */
83290.                     LABEL := '1'
83291.                     RC_REG := '15'
83292.         endif
83293.     else
83294.         if RC[1] = '1'
83295.             then /* Register was specified. */
83296.                 RC_REG := RC(1)
83297.             endif /* 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.                         endif
83305.                     else
83306.                         generate (LABEL || 'LA 15,15 RC')
83307.                     endif
83308.                     RC_REG := '15'
83309.                 endif
83310.             endif
83311.         endif
83312.     endif
```

C-96
83314. Bloc 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_SAVED
83323. then
83324. if OS_LINKAGE
83325. then
83326. if OS_POINTER := '(13)'
83327. LOCAL_POINTER := ''
83328. else
83329. LOCAL_POINTER := BLOCK_LABEL_PREFIX RESTORE_AREA '
83330. fi
83331. if RC_REG # 15
83332. then
83333. if RETURNAREA_REG < 14
83334. then
83335. if RC_REG := '(13)'
83336. else
83337. LOCAL_POINTER := BLOCK_LABEL_PREFIX RESTORE_AREA '
83338. fi
83339. the Saved LABEL := 'STRC8302 WARNING--NO CHECK MADE TO INSURE RETURNING REGISTERS ARE AMONG THOSE SAVED IN TRUNCATED SAVE AREA')
83340. fi
83341. I := 1
83342. while I <= N'RETURN do
83343. if RETURN(I) = 'N' /* Self-defining term. */
83344. then
83345. OFFSET := (OFFSET_TO_GPRO - 1) * 4
83346. generate (LABEL : LABEL[I], ST : 15, RC_REG[I], OS_POINTER)
83347. fi
83348. if FIRST_SAVAREA_REG # 14 and N'RETURN > 0
83349. then
83350. generate (LABEL[I], ST : RETURN(I), LOCAL_POINTER[I], OFFSET : OS_POINTER)
83351. fi
83352. I := I + 1
83353. while I <= N'RETURN do
83354. if RETURN(I) = 'N' /* Self-defining term. */
83355. then
83356. OFFSET := (OFFSET_TO_GPRO + RETURN(I) - (RETURN(I) + 2)/16*16) * 4
83357. generate (LABEL[I], ST : RETURN(I), LOCAL_POINTER[I], OFFSET : OS_POINTER)
83358. else
83359. if FIRST_SAVAREA_REG < 14
83360. then
83361. generate (LABEL[I], ST : RETURN(I), LOCAL_POINTER[I], OFFSET : OS_POINTER)
83362. else
83363. generate (LABEL[I], ST : RETURN(I), LOCAL_POINTER[I], OFFSET : OS_POINTER)
83364. fi
83365. fi
83366. fi
83367. fi
83368. fi
83369. fi
83370. endif
83371. fi
83372. fi
83373. endif
83374. fi
83375. endif
83376. endif
83377. endif
83378. /* (Termination: I is incremented, N'RETURN is fixed in loop;
83379. I must eventually exceed N'RETURN. */
83380.
"CORP" Macro -- 6 July 1973

83381. PROC CORP_FREE_DYN_SAVESAPA
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. COLD

C-98
"CORP" Macro -- 6 July 1973

83388. PROC CORP_RESTORE_REGS
83389. /* Restore registers REST1 through REST2 from proper save area if
83390. saved. */
83391. int OFFSET, I /* Temporaries. */
83393. if ANY_REGS_SAVED
83395. then
83396. if FIRST_REST_VALUE_KNOWN
83397. else
83399. endif
83400. if FIRST_SAVEAREA_REGS < 16
83402. then
83404. GORGAN (LABEL | | ' L | | MOLT | | ' | | REST1 | | ', | |
83405. REST2 | | COMMA2 | | LOCAL_POINTER | | '(' | | REST1 | |
83406. GPRO_OFFSET_STRING | | ')#' | | OS_POINTER)
83407. else
83409. GORGAN (LABEL | | ' L | | MOLT | | ' | | REST1 | | ', | |
83410. REST2 | | COMMA2 | | LOCAL_POINTER | | '(' | | REST1 | |
83411. GPRO_OFFSET_STRING | | '-(' | | REST1 | | '*2)/16*16)|' | | OS_POINTER
83412. endif
83413. endif
83414. LABEL := "
83415. C99
"CORP" macro -- 6 July 1973

93418. PROC CORP_RERESTORE_DEBUG_ENVIRONMENT
93419. */ Restore flags which were in progress before the PROC (unless
93420. global raised null raise to suppress restore). Values are packed
93421. in PREVIOUS_DEBUG_VECTOR and need only be unpacked. */
93422.
93423. if PREVIOUS_DEBUG_VECTOR ==
93424. then
93425. DEBUG_BLOCK_MULT_REQD := PREVIOUS_DEBUG_VECTOR[1,1]
93426. DEBUG_PROCNAME_FILE := PREVIOUS_DEBUG_VECTOR[2,1]
93427. DEBUG_LISTLOCKS_REQD := PREVIOUS_DEBUG_VECTOR[3,1]
93428. DEBUG_BLOCK_SHORTS_REQD := PREVIOUS_DEBUG_VECTOR[4,1]
93429. DEBUG_PROCCONTENTS_REQD := PREVIOUS_DEBUG_VECTOR[5,1]
93430. DEBUG_PROCNAME_REQD := PREVIOUS_DEBUG_VECTOR[6,1]
93431. DEBUG_CODECVALUES_REQD := PREVIOUS_DEBUG_VECTOR[7,1]
93432. DEBUG_SAVENAME_REQD := PREVIOUS_DEBUG_VECTOR[8,1]
93433. fi
93434. END
IEEE CORP_LP8_CRP and BCK AREAS
83456. /* If needed, generate CRP and BCK save areas. */
83457. 
83458. /* If rejuicel, generate CRP and BCK save areas. */
83459. 
83460. char
83461. LAST_AREA, /* Label of CRP or BCK area, whichever is generated
83462. last. */
83463. FWD_PTR, /* Label generated as forward pointer in last area. */
83464. TARGEP /* Temporary. */
83465. 
83466. if
83467. /* if OS_LINKAGE and DEBUG_SAVETRACES_REQD */
83468. then /* we need a CRP save area. */
83469. 
83470. if
83471. LINK = '033Z'
83472. then /* We need a CRP save area. */
83473. TARGEP := BLOCK_LABEL_PREFIX II 'FIN'
83474. generate (LABEL II 'B' II TARGET)
83475. LABEL := TARGEP
83476. fi
83477. LAST_AREA := BLOCK_LABEL_PREFIX II 'CRP'
83478. generate (LAST_AREA II 'DS OF')
83479. if DEBUG_SAVETRACES_REQD
83480. then /* We need the BCK save area. */
83481. LAST_AREA := BLOCK_LABEL_PREFIX II 'BCK'
83482. generate (LAST_AREA II 'DS OF')
83483. if DEBUG_SAVETRACES_REQD
83484. then /* We need the BCK save area. */
83485. LAST_AREA := BLOCK_LABEL_PREFIX II 'BCK'
83486. generate (LAST_AREA II 'DS OF')
83487. 
83488. /* if OS_LINKAGE and DEBUG_SAVETRACES_REQD */
83489. then /* we need a CRP save area. */
83490. 
83491. if
83492. LINK = '033Z'
83493. then /* We need a CRP save area. */
83494. TARGEP := BLOCK_LABEL_PREFIX II 'FIN'
83495. generate (LABEL II 'B' II TARGET)
83496. LABEL := TARGEP
83497. fi
83498. LAST_AREA := BLOCK_LABEL_PREFIX II 'CRP'
83499. generate (LAST_AREA II 'DS OF')
83500. if DEBUG_SAVETRACES_REQD
83501. then /* We need the BCK save area. */
83502. LAST_AREA := BLOCK_LABEL_PREFIX II 'BCK'
83503. generate (LAST_AREA II 'DS OF')
83504. if DEBUG_SAVETRACES_REQD
83505. then /* We need the BCK save area. */
83506. fi
83507. 
83508. 
83509. fi
83510. fi
83511. fi
83512. 
83513. /* generate (LABEL II 'DCX"FC" II PROC_ID_BYTE II "0000") */
83514. 
83515. if DEBUG_SAVETRACES_REQD
83516. then /* We need the BCK save area. */
83517. LAST_AREA := BLOCK_LABEL_PREFIX II 'BCK'
83518. generate (LAST_AREA II 'DS OF')
83519. fi
83520. 
83521. if DEBUG_SAVETRACES_REQD
83522. then /* We need the BCK save area. */
83523. LAST_AREA := BLOCK_LABEL_PREFIX II 'BCK'
83524. generate (LAST_AREA II 'DS OF')
83525. fi
83526. 
83527. 
83528. 
83529. 
83530. 
83531. 
83532. 
83533. 
83534. 
83535. 
83536. 
83537. 
83538. 
83539. 
83540. 
83541. 
83542. 
83543. 
83544. 
83545. 
83546. 
83547. 
83548. 
83549. 
83550. 
83551. 
83552. 
83553. 
83554. 
83555. 
83556. 
83557. 
83558. 
83559. 
83560. 
83561. 
83562. 
83563. 
83564. 
83565. 
83566. 
83567. 
83568. 
83569. 
83570. 
83571. 
83572. 
83573. 
83574. 
83575. 
83576. 
83577. 
83578. 
83579. 
83580. 
83581. 
83582. 
83583. 
83584. 
83585. 
83586. 

"CORP" Macro -- 5 July 1973
"CORR" macro -- 6 July 1973

83488. proc disp_new_exit_label
83489. /* If an exit label is required, put it into LABEL (generating any label already there). Issue note regarding EXIT references. */
83490.
83491. if exit_label_size(currentnest_level)
83492. then
83493. if label "+"
83494. then
83495. label := block_label_prefix ++ "EXIT"
83496. if exit_severity = "F"
83497. then
83498. exit_severity := "I"
83499. note (exit_severity,
83500. "SPEC8001 ONE OR MORE EXIT'S REFERENCE THIS POINT")
83501. exit_label_required(currentnest_level) := false
83502. endif
83503. endif
83504. endif
83505. endif
83506. /* Exit label will have been generated by POP_OLD_BLOCK time. */
83507. endif
83508.

C-102
"EXIT_FIND" Macro -- 10 July 1973

91001. MACRO EXIT_FIND ( ; REQD_NAME)
91002. /* Set ULTIMATE_BRANCH_LABEL to exit label for block whose name
91003. is the argument; if no such block, issue message and set
91004. ERROR_OCCURRED. On valid block, that block is marked as needing
91005. an EXIT label. */
91006. 
91008. CALL TRACE_PRINTER ( ; 'EXITFIND')
91009. /* Print macro name "EXITFIND" in mnote if tracing on. */
91010. int I /* Temporary. */
91012. if REQD_NAME = ' ' and '*' then
91013. /* We must search for the right block. */
91014. do I := I - 1 /* (Termination: I is decremented and would eventually become
91015. \leq 0 even if other tests never occurred.) */
91016. if I \leq 0 or
91017. then /* Not found in search. */
91018. if REQD_NAME = BLOCK_NAME(I) and '*' then
91019. then /* EXIT not nested. */
91020. /* Assume all will go well. */
91021. ENTRY_REQD(I) := true
91022. If ENTRY_REQD(I) = 'DO' and
91023. then /* Just use special PROC prefix form. */
91024. ULTIMATE.Branch_LABEL := '$P' II INFORMATION(I)[8,1] II 'EXIT'
91025. else
91026. ULTIMATE.Branch_LABEL := '$$' || BLOCK_NUMBER(I) II 'EXIT'
91027. fi
91028. fi
91029. fi
91030. else /* Found. */
91031. if BLOCK_TYPE(I) = 'PROC' and INFORMATION(I)[8,1] and
91032. then /* Just use special PROC prefix form. */
91033. ULTIMATE.Branch_LABEL := '$P' II INFORMATION(I)[6,2] II 'EXIT'
91034. else
91035. ULTIMATE.Branch_LABEL := '$$' II BLOCK_NUMBER(I) II 'EXIT'
91036. fi
91037. fi
91038. fi
91039. fi
91040. fi
91041. if BLOCK_TYPE(I) = 'PRO' and
91042. then /* Set ERROR_OCCURRED := true */
91043. /* Set ERROR_OCCURRED := true */
91044. /* ERROR_OCCURRED := true */
91045. else
91046. then /* Set ERROR_OCCURRED := true */
91047. /* ERROR_OCCURRED := true */
91048. else
91049. then /* Set ERROR_OCCURRED := true */
91050. /* ERROR_OCCURRED := true */
91051. else
91052. /* ERROR_OCCURRED := true */
91053. /* ERROR_OCCURRED := true */
91054. fi
91055. fi
91056. fi
91057. fi
91058. exit

C-103
"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.
92005. CALL TRACE_PRINTER ( ; "POP")
92006. /* Print macro name "POP" in note if tracing on. */
92007. if CURRENT_NEST_LEVEL < NESTING_LIMIT
92008. then
92009. if END_LABEL_REQD(CURRENT_NEST_LEVEL)
92010. then
92011. GENERATE ('$' || BLOCK_NUMBER(CURRENT_NEST_LEVEL) || 'END DS OH')
92012. if EXIT_LABEL_REQD(CURRENT_NEST_LEVEL) OR OLD_EXIT #
92013. then
92014. if OLD_EXIT #
92015. then
92016. EXIT_SEVERITY := '0'
92017. fi
92018. fi
92019. GENERATE (EXIT_SEVERITY || "STC9201 ONE OR MORE EXIT'S REFERENCE THIS POINT")
92020. if EXIT_LABEL_REQD(CURRENT_NEST_LEVEL)
92021. then
92022. GENERATE ('X' || BLOCK_NUMBER(CURRENT_NEST_LEVEL) || 'XIT DS OH')
92023. if OLD_EXIT #
92024. then
92025. GENERATE (OLD_EXIT || ' DS OH')
92026. fi
92027. fi
92028. if DEBUG_LISTLOCKS_REQD
92029. then
92030. MNOTE (EXIT_SEVERITY, "STC9203 END OF BLOCK ' || BLOCK_NUMBER(CURRENT_NEST_LEVEL)
92031. || ' ' || BLOCK_NAME(CURRENT_NEST_LEVEL) || AT DEPTH ' ||
92032. CURRENT_NEST_LEVEL)
92033. then
92034. MNOTE (*, "**********************************************************************")
92035. fi
92036. fi
92037. CURRENT_NEST_LEVEL := CURRENT_NEST_LEVEL - 1
92038. fi
92039. /* (Lemma: Execution of POP_OLD_BLOCK always decrements
92040. CURRENT_NEST_LEVEL by exactly one. */
92041.

C-104
Macro -- 10 July 1973

93001. MACRO PUSH_NEW_BLOCK (BLOCK_NAME_VALUE):
93002.   BLOCK_TYPE_VALUE=''
93003.   OPERAND1_VALUE=''
93004.   OPERAND2_VALUE=''
93005.   OPERAND3_VALUE=''
93006.   INFORMATION_VALUE=''
93007.   END_LABEL_VALUE=false
93009. /* Define new block; add to stack, save block specifications.
93010. All macro operands unspecified default to null string except
93011. END_LABEL_VALUE defaults to false. */
93013. call TRACEPRINTER ( ; 'PUSH')
93014. /* Print macro name "PUSH" in mnote if tracing on. */
93015. NESTING_LIMIT := 100
93016. /* Ensures an adequate depth of stack is set in variable. Note that stack
93017. depth and this variable must match, but may be changed to any
93018. value. */
93020. /* Same for general CASE stack. */
93021. ERROR_OCCURRED := false
93022. CURRENT_NEST_LEVEL := CURRENT_NEST_LEVEL + 1
93023. if CURRENT_NEST_LEVEL > NESTING_LIMIT
93025. then
93026. mnote (1, 'STACK NESTING LIMIT OF ' || NESTING_LIMIT ||
93027. SPACE9901 NON-CASE BLOCK IMMEDIATELY SURROUNDED BY DOCASE INVALID')
93028. ERROR_OCCURRED := true
93030. LAST_BLOCK_NUMBER := LAST_BLOCK_NUMBER + 1
93031. BLOCK_NUMBER(CURRENT_NEST_LEVEL) := LAST_BLOCK_NUMBER
93032. if CURRENT_NEST_LEVEL > 1 and
93034. BLOCK_TYPE(CURRENT_NEST_LEVEL-1) = 'DOCASE' and
93035. BLOCK_TYPE_VALUE = 'CASE'
93037. then
93039. mnote (3, 'STACK NESTING LIMIT AT DEPTH ' || CURRENT_NEST_LEVEL ||
93041. BLOCK_NAME(CURRENT_NEST_LEVEL) := BLOCK_NAME_VALUE
93042. if BLOCK_NAME_VALUE = ''
93044. then
93046. BLOCK_LABEL_PREFIX := '2' || LAST_BLOCK_NUMBER
93048. fi
93050. BLOCK_NAME(CURRENT_NEST_LEVEL) := 'BLK' || LAST_BLOCK_NUMBER
93052. if DEBUG_LISTBLOCKS true
93054. then
93056. mnote (3, 'STACK NESTING LIMIT AT DEPTH ' || CURRENT_NEST_LEVEL ||
93058. BLOCK_NAME(CURRENT_NEST_LEVEL) || ' AT DEPTH ' || CURRENT_NEST_LEVEL)
93059. fi
93060. END
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. FULL_JUMP_CONDITION,  
94009. OP_CODE)  
94010. /* Generate indicated instruction followed by appropriate conditional  
94011. branch to indicated label. */  
94012. char LOCAL_MASK, LOCAL_REL,  
94013. /* Holds mask or relation for branch. */  
94014. BCTADD /* Label to go on bc instruction. */  
94015. call TRACE_PRINTER (11 SIMPLECOND)  
94016. /* Print macro name "SIMPLCONND" in mnote if tracing on. */  
94017. call SIMPLECOND_WZEN.Mask OR_REL  
94018. /* Extract local task or local REL from OPER's. If LOCAL_REL is an  
94019. external value (GT, GE, EQ, LT, or LE), replace it with the  
94020. proper value (H, AL, E, L, or NH). */  
94021. docase OP_CODE  
94022. of  
94023. case 1 /* Mask or relation only. */  
94024. BC_TAG := LABEL  
94025. esac  
94026. case 2  
94027. BLOAD (3, "STAIR401 INSUFFICIENT OPERANDS FOR TEST " || OP_CODE || "")  
94028. esac  
94029. case 3  
94030. generate (LABEL || " " || OP_CODE || " " || OPER1)  
94031. esac  
94032. case 4  
94033. generate (LABEL || " " || OP_CODE || " " || OPER1 || " " || OPER2)  
94034. esac  
94035. case 5  
94036. generate (LABEL || " " || OP_CODE || " " || OPER1 || " " || OPER2 || " " || OPER3)  
94037. esac  
94038. case misc  
94039. /* RECORD 3, "STAIR402 SUPERFLUOUS OPERANDS FOR TEST " || OP_CODE || "")  
94040. esac  
94041. esacdoc  
94042. endif  
94043. if LOCAL_REL = " "  
94044. if FALLTHRU_CONDITION /* is true: */  
94045. then  
94046. generate (BC_TAG || " " || LOCAL_REL || " " || BRANCH_LABEL)  
94047. endif  
94048. else  
94049. if FALLTHRU_CONDITION /* is true: */  
94050. then  
94051. /* Invert relation. */  
94052. if LOCAL_REL[1,1] = 'N'  
94053. then  
94054. LOCAL_REL := LOCAL_REL ||  
94055. esac  
94056. endif  
94057. generate (BC_TAG || " " || LOCAL_REL || " " || BRANCH_LABEL)  
94058. endif  
94059. endif  
94060. endif  
94061. /* Invert mask. */  
94062. generate (BC_TAG || " " || BCcompareTo || LOCAL_MASK || " " || BRANCH_LABEL)  
94063. endif  
94064. endif  
94065. endif  
94066. endif  
94067. endif  
94068. endif  
94069. endif  

C-106
"SIMPLE_CONDITIONAL" macro -- 10 July 1973

94070. \#include SIMPCOND.Get Mask On Rel
94071. /* Extract LOCAL_ASK or LOCAL_REL from OPER's. If LOCAL_REL is a
94072.   extended value (GT, GE, EQ, LT, or LE), replace it with the
94073.   proper value (H, NL, E, L, or NH). */
94074. 
94075. if OP_COUNT = 0
94076. then
94077.   note (9, "STRING NOT CONDITION SPECIFIED—"MASK-0" ASSUMED")
94078. else
94079.   LOCAL_ASK := '0'
94080. fi
94081.  
94082. if SYSLIST(OP_COUNT)[1,5] = 'MASK=
94083.  then
94084.   LOCAL_ASK := SYSLIST(OP_COUNT)[6,8]
94085.   "LOCAL_REL := SYSLIST(OP_COUNT)[5,8]
94086. else
94087.   LOCAL_REL := SYSLIST(OP_COUNT)
94088. end
94089.  
94090.  
94091.  
94092.  case 'GT'
94093.    LOCAL_REL := 'H'
94094.  esac
94095.  case 'GE'
94096.    LOCAL_REL := 'NL'
94097.  esac
94098.  case 'EQ'
94099.    LOCAL_REL := 'E'
94100.  esac
94101.  case 'LT'
94102.    LOCAL_REL := 'L'
94103.  esac
94104.  case 'LE'
94105.    LOCAL_REL := 'NH'
94106.  esac
94107.  fi
94108.  fi
94109.  
94110.  

C-107
"TRACE_PRINTER" Macro -- 10 July 1973

95001. Macro TRACE_PRINTER ( ; MACRO_NAME)
95002. /* Prints macro name if tracing on. */
95003. if DEBUG_MACRONAMES_REQD
95004. then
95005. include (*, 'SITC9500 ' || MACRO_NAME)
95006. fi
95007. end

*----------------------------------------------------------------*
# VERIFY_END Macro -- 10 July 1973

96001.  **macro** VERIFY_END ( : REQD_TYPE, REQD_NAME)
96002.    /* Verifies current block has name specified by REQD_NAME operand, if
96003.    any, and that it is of type REQD_TYPE. Various errors receive
96004.    messages and either intermediate blocks are BLENDed as a fixup
96005.    or ERROR_OCCURRED is set. */
96006.   **end** 1 /* Temporary. */
"VERIFY_END" Macro -- 10 July 1973

96008. call TRACE_PRINTER ( ; 'VERIFY')
96009. /* Print macro name "VERIFY" in note if tracing on. */
96010. ERROR_OCCURRED := false /* Assumed. */
96011. if REQ_NAME = ' ' then
96012. if CURRENT_NEST_LEVEL < 0
96013. then
96014. note (6, 'STRC9607 NO BLOCKS ACTIVE—MACRO IGNORED')
96015. ERROR_OCCURRED := true
96016. else
96017. if CURRENT_NEST_LEVEL = 0 then
96018. note (6, 'STRC9608 NO BLOCKS ACTIVE—MACRO IGNORED')
96019. ERROR_OCCURRED := true
96020. else
96021. if AT BLOCK_TYPE(CURRENT_NEST_LEVEL) # REQD_TYPE
96022. then
96023. print (6, 'STRC9601 ONE BLEND ASSUMED TO GET TO '''' REQD_TYPE '''' BLOCK')
96024. call BLEND ( )
96025. else
96026. if AT CURRENT_NEST_LEVEL < 1 AND
96027. BLOCK_TYPE(CURRENT_NEST_LEVEL - 1) = REQD_TYPE
96028. then
96029. print (6, 'STRC9602 TWO BLENDS ASSUMED TO GET TO '''' REQD_TYPE '''' BLOCK')
96030. call BLEND ( )
96031. else
96032. print (6, 'STRC9603 CURRENT BLOCK IS NOT '''' REQD_TYPE '''' BLOCK—MACRO IGNORED')
96033. ERROR_OCCURRED := true
96034. fi
96035. fi
96036. fi
96037. fi
96038. endif
96039. endif
96040. fi
96041. /* fi
96042. else */ A block name was specified. */
96043. errorOccurred := true
96044. CURRENT_NAME := 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. print (6, 'STRC9604 NO ACTIVE BLOCK NAMED '''' REQD_NAME '''' BLOCK')
96053. ERROR_OCCURRED := true
96054. else
96055. AT REQD_TYPE # BLOCK_TYPE(I)
96056. then / block named found, but of wrong type. */
96057. print (6, 'STRC9605 BLOCK '''' REQD_NAME '''' IS NOT A '''' REQD_TYPE '''' BLOCK—MACRO IGNORED')
96058. ERROR_OCCURRED := true
96059. else
96060. while CURRENT_NEST_LEVEL > 1
96061. do /* BLEND any intermediate blocks. */
96062. print (6, 'STRC9606 END OF BLOCK '''' REQD_NAME '''' IMPLIES END OF BLOCK '''' BLOCK_NAME(CURRENT_NEST_LEVEL)'''' BLOCK')
96063. call BLEND ( )
96064. /* (Lemma: If CURRENT_NEST_LEVEL > 0 and no BLEND operands
96065. specified, BLEND will decrement CURRENT_NEST_LEVEL
96066. by exactly one.) */
96067. od /* (Termination: On all iterations, I is fixed and
96068. CURRENT_NEST_LEVEL > I > 0. But BLEND decrements
96069. CURRENT_NEST_LEVEL. Therefore, CURRENT_NEST_LEVEL must
96070. eventually become ≤ (actually =) I.) */
96071. fi
96072. fi
96073. fi
96074. fi
96075. fi
96076. fi
96077. endif
96078. /* (Lemma: If CURRENT_NEST_LEVEL > 0 and
96079. [REQ_NAME ≠ '''' or REQ_TYPE ≠ BLOCK_TYPE(CURRENT_NEST_LEVEL)] and
96080. REQ_TYPE = BLOCK_TYPE(CURRENT_NEST_LEVEL), then
96081. ERROR_OCCURRED will always be set false and CURRENT_NEST_LEVEL
96082. will be unmodified. Proof: If the hypothesis conditions are
96083. true, the module calls TRACE_PRINTER and sets ERROR_OCCURRED to
96084. false as its only actions. TRACE_PRINTER modifies no
96085. globals. ] */
**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"
97100.    operand is passed as an actual macro operand. The other variables
97101.    mentioned are globals. */
97102.
97103.    int  
97104.    COND_COUNT,  /* Counts the simple conditionals within the
97105.    conditional expression. */
97106.    DEPTH,  /* Angle bracket nesting depth of simple conditional being
97107.    processed. */
97108.    INDEX,  /* Operand index within the SYSLIST of the simple conditional
97109.    being processed. */
97110.    OP_COUNT,  /* Number of suboperands in the simple conditional being
97111.    processed. */
97112.    NEXT_INDEX,  /* Index of the AND or OR which follows the current
97113.    simple conditional. */
97114.    NEXT_DEPTH,  /* Angle bracket depth of NEXT_INDEX. */
97115.    LA_DEPTH,  /* Angle bracket depth during operand look-ahead. */
97116.    I,  /* Operand index of operand being examined during
97117.    look-ahead. */
97118.    bit
97119.    AND_OR_OUTSTANDING,  /* Indicates whether an AND or OR follows
97200.    the current simple conditional. */
97201.    LOCAL_FALLTHRU_CONDITION,  /* Logical value of the simple conditional
97202.    being processed which is to lead to control falling through the
97203.    test. */
97204.    LOCAL_LABEL_REQD(20),  /* Indicates whether the corresponding
97205.    simple conditionals require a label due to branching logic. */
97206.    char
97207.    LABEL,  /* Outstanding label waiting to be generated. */
97208.    LOCAL_BRANCH_LABEL /* Label for branch target if current simple
97209.    conditional has the opposite truth value from that stored in
97210.    LOCAL_FALLTHRU_CONDITION. */
```
"CONDITIONAL_EXPRESSION_PROCESSOR" Macro -- 9 July 1973

97041. call TRACE_PRINT ( ; 'CEP')
97042. /* Print macro name "CEP" in error 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. /* Stepped INDEX up to next simple conditional, incrementing DEPTH for any
97052. "A" found and setting ERROR_OCCURRED on any S YX error. Increments
97053. CON D_COUNT for the condition found. */
97054. if ERROR_OCCURRED
97055. then
97056. break
97057. end if
97058. and_or_outstanding := false
97059. call CEP_LOOKAHEAD
97060. /* Find LOCAL_BRANCH_LABEL or generate one. If LOCAL_BRANCH_LABEL
97061. is the ULTIMATE_FALLTHRU_LABEL, set FALLTHRU_LABEL USED. Set
97062. LOCAL_FALLTHRU_CONDITION. Also set NEXT_INDEX and NEXT_DEPTH with
97063. the INDEX/DEPTH of the AND OR 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. break
97069. end if
97070. if LOCAL_LABEL_REQ(COND_COUNT)
97071. then
97072. label := BLOCK_LABEL_PREFIX || UNIQUE_LABEL_ID || COND_COUNT
97073. endif
97074. op_count := N'SYSLST(INDEX)
97075. call SIMPLE_CONDITIONAL (label |)
97076. sysslst(INDEX,1),
97077. sysslst(INDEX,2),
97078. sysslst(INDEX,3),
97079. sysslst(INDEX,4),
97080. sysslst(INDEX,5),
97081. LOCAL_BRANCH_LABEL,
97082. LOCAL_FALLTHRU_CONDITION,
97083. op_count)
97084. label := 'T'
97085. index := NEXT_INDEX
97086. depth := NEXT_DEPTH
97087. endif /* 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 (6, 'STRC9701 INSUFFICIENT OPERANDS')
97093. endif
97094. if DEPTH = 0
97095. then
97096. message (6, 'STRC9702 INSUFFICIENT BRACKETS')
97097. endif
97098. end
57100. PROC CEP_FND_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_OCCURRED on any syntax error,
57104. giving message. Increment COND_COUNT by 1. */
57106. ERROR_OCCURRED := false
57107. while INDEX ≤ LAST_INDEX and (SYSLIST(INDEX) = '"' or = '="')
57108. do
57109.   DEPTH := DEPTH + 1
57110.   INDEX := INDEX + 1
57111. od /* (Termination: INDEX is incremented, LAST_INDEX is fixed in
57112.   loop; INDEX would eventually exceed LAST_INDEX even if other
57113.   tests never occur.) */
57114. if SYSLIST(INDEX)[1] = "(" then
57115. mnote(8, 'STRC9706 SYNTAX ERROR--LOOKING FOR SIMPLE ' ||
57116. 'CONDITIONAL, FOUND ' || SYSLIST(INDEX) || '"')
57117. ERROR_OCCURRED := true
57118. fi
57119. COND_COUNT := COND_COUNT + 1
57120. corp
Search operands beyond current simple conditional. If AND/OR found, label found or generated for that spot into LOCAL_BRANCH_LABEL. If same as ULTIMATE_FALLTHRU_LABEL, set FALLTHRU_LABEL_USED. Decide whether this test is to fallthru on true or false and set LOCAL_FALLTHRU_CONDITION. If syntax error found during lookahead, give message and set ERROR_OCCURRED. Also set NEXT_INDEX and NEXT_DEPTH with the index/depth of the AND/OR. */

ERROR_OCCURRED := false

LA_DEPTH := DEPTH

I := INDEX + 1

while I \leq LAST_INDEX and (SYSLIST(I) = '>' or = '/') do

if LA_DEPTH > 0 then
  LA_DEPTH := LA_DEPTH - 1
else
  mnote (8, 'STEC9705 SUPERFLUOUS BRACKET IGNORED')
fi

/* (Termination: I is incremented, LAST_INDEX is fixed in loop; INDEX would eventually exceed LAST_INDEX even if other tests never occur.) */

NEXT_INDEX := I + 1

/* (Lemma: NEXT_INDEX > I > INDEX.) */

NEXT_DEPTH := LA_DEPTH

if I > LAST_INDEX then

  LOCAL_BRANCH_LABEL := ULTIMATE_BRANCH_LABEL
  LOCAL_FALLTHRU_CONDITION := ULTIMATE_FALLTHRU_CONDITION
else

  if SYSLIST(I) = 'AND' or = 'OR'
    then
      CALL CEP_SCAN_FOR_BRANCH
    else
      mnote (8, 'STEC9703 SYNTAX ERROR--LOOKING FOR "AND" OR "OR", ' || SYSLIST(I) || "")

  ERROR_OCCURRED := true

fi

/* (Lemma: The value of NEXT_INDEX returned is greater than the value of INDEX entered.) */

**************
"CONDITIONAL_EXPRESSION_PROCESSOR" Macro -- 9 July 1973

97171. BEG 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 LOCKFOB /* Operation ("AND" or "OR") opposite of SYSLIST(I). */
97181. INK MAX_DEPTH, /* The maximum depth at which the LOOKFOR'ed
97182. operation is a possible branch target. */
97183. TARGET /* Simple conditional number which is the target for
97184. the branch. */
AND_GB_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. */
fi

lockpos := 'AND'

if SYSLIST(I) = 'AND' or SYSLIST(I) = 'OR'
  then
    TARGET := CONDCOUNT + 1
  /* Operation is CR. */
fi

if LA_DEPTH = 0
  then
    I := LAST_INDEX + 1
  /* Advance I to force skip of unnecessary search. */
fi

TARGET := CONDCOUNT + 1

while I <= LAST_INDEX and (SYSLIST(I) # LOCKFOR or LA_DEPTH > MAX_DEPTH)
  do
    if SYSLIST(I) = '<' or SYSLIST(I) = '>'
      then
        TARGET := TARGET + 1
    else
      if SYSLIST(I) = '('
        then
          TARGET := TARGET + 1
      else
        local_branch_label := block_label_prefix || unique_label_id || target
        local_label_required := true
        /* Note that we are relying on the automatic initialization of the
         * local_label_required array to false at start of every invocation of
         * conditional_expression_processor. */
  od

if I > LAST_INDEX
  then
    (lockpos = 'OR' /* Operand was OR. */ and
     ~ ultimate_fallthru_condition) or
    (lockpos = 'AND' /* Operand was OR. */ and
     ultimate_fallthru_condition /= true)
  then
    local_branch_label := ultimate_fallthru_label
    fallthrough_label_used := true
  else
    local_branch_label := ultimate_branch_label
fi

else
  local_branch_label := block_label_prefix || unique_label_id || target
  local_label_required(target) := true
  /* Note that we are relying on the automatic initialization of the
   * local_label_required array to false at start of every invocation of
   * conditional_expression_processor. */
fi

C-116
**TERMINATE_DO_LOOP** Macro  --  26 June 1973

```plaintext
98001.  RACIO  TERMNATE_DO_LOOP ( ; )
98002.  /* Called by DO macro to terminate the current DO block loop by
98003.  generating the necessary loop-terminating branches. If control
98004.  can fall out of the bottom of the code at loop termination, set
98005.  TDL_FALLTHRU_OCCURS to true; else set false. */
98006.  bit
98007.  LB_LABEL_REQ /* Indicates whether label required on looping
98008.  branch instruction. */
98009.  char
98010.  S8011. X, /* Temporary string. */
98012.  LABEL, /* Holds looping branch label. */
98013.  /* One character codes for flow points which are targets for flow points 8, 10, and 11. */
98014.  BRANCH8, BRANCH10, BRANCH11,
98015.  /* to be generated. */
98016.  LB_OPCODE, OPER1, OPER2
98017.  /* Opcode and operands of looping branch
98018.  to be generated. */
98019.  call  TRACE_PRINTER( ; 'TDL')
98020.  /* Extract the following stored
98021.  by DO macro:
98022.  $8023. X := INFOBRMATION(CURRENT
98024.  NEST_LEVEL)
98025.  BRANCH8 := X[1,1]
98026.  BRANCH10 := X[2,1)
98027.  BRANCH11 := X[3,1]
98028.  LBLABEL_BEQ := X[4,1]
98029.  OPER1 := OPERANDI(CURRENT_NESTLEVEL)
98030.  OPER2 := OPERAND2(CURRENT_NESTLEVEL)
98031.  LBOPCODE := OPEBAND3(CURRENTBEST_LEVEL)
98032.  BLOCK_LABEL_PREFIX := '$' I BLOCK_NUMBER(CURRENT_NEST_LEVEL)
98033.  if LB_LABEL_REQ
98034.  then
98035.  LABEL := BLOCK_LABEL_PREFIX || 'LPB')
98036.  fi
98037.  esacall branch cases
98038.  case ('w', 'U')
98039.  generate (' B ' II BLOCK_LABEL_PREFIX II BRANCH8 II '1')
98040.  esac
98041.  case ('BEG')
98042.  generate (' B ' II BLOCK_LABEL_PREFIX II 'BEG')
98043.  esac
98044.  case ('l') /* Nothing to generate; fall through to looping branch. */
98045.  esacased
98046.  TDL_FALLTHRU_OCCURS := false /* Assume we will always branch. */
98047.  if LB_OPCODE = 'B' or 'U'
98048.  then /* A looping branch is required. */
98049.  TDL_FALLTHRU_OCCURS := false
98050.  /* Mark that we will fall out of the looping branch. */
98051.  if BRANCH10 = 'B'
98052.  then
98053.  X := 'BEG'
98054.  else /* BRANCH10 = 'w' or 'U'. */
98055.  X := BRANCH10 || '1'
98056.  fi
98057.  if LB_OPCODE = 'BCT'
98058.  then
98059.  generate (LABEL II ' BCT ' II OPER1 II ', ' II BLOCK_LABEL_PREFIX || X)
98060.  else
98061.  generate (LABEL II ' B ' II LB_OPCODE II ' ' II OPER1 II ' ' II BLOCK_LABEL_PREFIX || X)
98062.  fi
98063.  if BRANCH11 = 'B'
98064.  then
98065.  if it must be 'w' or 'U' looping branch fall-through is not to end of loop; generate branch to proper alternative conditional test. */
98066.  generate (B ' ' II BLOCK_LABEL_PREFIX || BRANCH11 || '1')
98067.  TDL_FALLTHRU_OCCURS := false
98068.  /* Branch defeats fall through. */
98069.  fi
98070.  fi
98071.  fi
98072.  /* (Lemma: TERMINATE_DO_LOOP does not modify
98073.  CURRENT_NEST_LEVEL) */
```

C-117
"HEX" Macro -- 19 December 1973

59001. MACRO HEX ( ; )
59002. /* Converts the integer in VAR to a two-character hex string in
59003. a two-character hex string.
59004. */
59005. I, J /* Temporary variables. */
59006. CALL TRACE_PRINTER ( ; 'HEX')
59007. /* Print macro name "HEX" in anote if tracing on. */
59008. IF HEX(I) ≠ '0'
59009. THEN /* Hex array must be initialized. */
59100. HEX(1) := '0' 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(I)/16
59106. J := HEX(J) - I*16 + 1
59107. HEX_OUT := HEX(I+1) || HEX(J)
59108. END

C-118
CROSS-REFERENCE LISTING

LABEL (cont.)

21048.  • 21049.  21857.  • 21858.  • 31029.  • 31052.  • 31076.
31078.  • 31161.  31162.  • 31166.  • 31170.  • 31171.  • 31178.
31176.  • 31374.  31396.  • 31397.  • 32130.  • 33042.  • 33172.
33123.  • 33125.  33207.  • 33235.  • 33216.  • 33467.
33476.  • 33477.  33507.  • 33529.  • 33531.  • 33552.  • 33573.
33557.  • 33558.  33599.  • 33701.  • 33716.  • 33720.  • 33725.
37233.  • 37242.  37263.  • 37267.  • 37256.  • 37259.
37263.  • 61006.  • 61017.  61022.  • 61024.  • 61038.  • 61039.
41043.  • 41045.  41050.  • 61050.  • 61056.  • 61060.  • 61090.
61094.  01150.  01359.  01366.  • 01636.  • 01638.  • 01643.
81459.  81665.  81968.  • 81973.  • 81974.  • 81977.  • 81979.
81658.  • 81673.  • 81674.  • 81676.  • 81677.  • 81723.  • 81732.
81738.  • 81739.  81769.  • 81773.  • 81773.  • 81779.  • 81800.
81822.  • 81823.  81826.  • 81827.  • 81838.  • 81940.  • 81941.
81877.  • 81878.  81880.  • 81887.  • 81888.  • 81890.  • 81903.
81906.  01914.  01926.  • 01929.  • 01931.  • 01933.  • 01941.
81942.  • 81971.  01973.  • 01974.  • 03031.  • 03071.  • 03114.
83115.  83121.  83123.  • 83127.  • 83128.  • 83132.  • 83134.
83135.  • 83140.  • 83147.  • 83266.  • 83267.  • 83271.  • 83273.
83275.  • 83286.  • 83290.  • 83303.  • 83305.  • 83307.  • 83339.
83340.  • 83344.  • 83346.  • 83360.  • 83365.  • 83366.  • 83375.
83383.  • 83388.  • 83393.  • 83399.  • 90401.  • 90405.  • 90432.
83505.  • 83508.  • 83509.  • 83948.  • 90401.  • 90405.  • 90432.
94035.  • 94037.  • 97036.  • 97043.  • 97072.  • 97075.  • 97084.
99011.  • 99013.  99050.  99061.  • 99074.  • 99075.  • 99076.
99012.  • 99013.  99050.  99061.  • 99074.  • 99075.  • 99076.
99013.  • 99014.  99050.  99061.  • 99074.  • 99075.  • 99076.
99014.  • 99015.  99050.  99061.  • 99074.  • 99075.  • 99076.
99015.  • 99016.  99050.  99061.  • 99074.  • 99075.  • 99076.
99016.  • 99017.  99050.  99061.  • 99074.  • 99075.  • 99076.
99017.  • 99018.  99050.  99061.  • 99074.  • 99075.  • 99076.
99018.  • 99019.  99050.  99061.  • 99074.  • 99075.  • 99076.
99019.  • 99020.  99050.  99061.  • 99074.  • 99075.  • 99076.
99020.  • 99021.  99050.  99061.  • 99074.  • 99075.  • 99076.
99021.  • 99022.  99050.  99061.  • 99074.  • 99075.  • 99076.
99022.  • 99023.  99050.  99061.  • 99074.  • 99075.  • 99076.
99023.  • 99024.  99050.  99061.  • 99074.  • 99075.  • 99076.
99024.  • 99025.  99050.  99061.  • 99074.  • 99075.  • 99076.
99025.  • 99026.  99050.  99061.  • 99074.  • 99075.  • 99076.
99026.  • 99027.  99050.  99061.  • 99074.  • 99075.  • 99076.
99027.  • 99028.  99050.  99061.  • 99074.  • 99075.  • 99076.
99028.  • 99029.  99050.  99061.  • 99074.  • 99075.  • 99076.
99029.  • 99030.  99050.  99061.  • 99074.  • 99075.  • 99076.
99030.  • 99031.  99050.  99061.  • 99074.  • 99075.  • 99076.
99031.  • 99032.  99050.  99061.  • 99074.  • 99075.  • 99076.
99032.  • 99033.  99050.  99061.  • 99074.  • 99075.  • 99076.
99033.  • 99034.  99050.  99061.  • 99074.  • 99075.  • 99076.
99034.  • 99035.  99050.  99061.  • 99074.  • 99075.  • 99076.
99035.  • 99036.  99050.  99061.  • 99074.  • 99075.  • 99076.
PROC_DACIVL_SAVE_TYPE 8144. 81530.
PROC_DEFINE_NEW_OSSAVE 81725. 81759.
PROC_ESTABLISH_EASE 81704. 81611.
PROC_GEN_LOCAL_SAVEAREA 81713. 81833.
PROC_GSE_OSSAVE_AREA 81705. 81729.
PROC_GSE_SAVEAREA 81706. 81699.
PROC_HEADER 81068. 81341.
PROC_INFO_SAVE 81101. 81962.
PROC_MULTIFASE_GEN 81082. 81817.
PROC_NAME #83001. 83057.
PROC_REG_SAVE 81071. 81814.
PROC_SCAN_OPTIONS 81063. 81105.
PROC_SET_SAVE_INFO 81037. 81477.
PROC_NEW_BLOCK 11022. 11054. 21077. 31042. 33059. 41009. 81053.
PROC_HEADER 81001. 81394. 81398. 81406. 81406. 81410. 81410.
PROC_REG_SAVE 81071. 81814.
PROC_SCAN_OPTIONS 81063. 81105.
PROC_SET_SAVE_INFO 81037. 81477.
RESTORE_AREA 83037. 83254. 83331.
RESTORE 83013. 83202. 83220. 83397. 83399. 83404. 83405.
RETURN 83011. 83202. 83220. 83397. 83399. 83404. 83405.
SAD 83105. 83154. 83155. 83155. 83155. 83155. 83155. 83155.
SAVE 81030. 81054. 81440. 81500. 81500. 81500. 81500. 81500.
SAVE_LENGTH 81811. 81861. 81953. 83091. 83166. 83252.
SAVE_TYPE 83037. 83037. 83037. 83037. 83037. 83037. 83037. 83037.
SAVE 81030. 81054. 81440. 81500. 81500. 81500. 81500. 81500.
SAVEVREG 83103. 83154. 83155. 83155. 83155. 83155. 83155. 83155.
SAVECACHE_CHECK 81177. 81210. 81320. 81320. 81320. 81320. 81320. 81320.
SAVECACHE_1ST_PROC 81175. 81210. 81320. 81320. 81320. 81320. 81320. 81320.
SAVECACHE_VALUE 81175. 81210. 81320. 81320. 81320. 81320. 81320. 81320.
SEC 81135. 81373. 81737. 81737. 81737. 81737. 81737. 81737.
SIMCCD_GET_MASK_OR_REL 84010. 84610. 84610.
SIMPLE_CONDITIONAL 84010. 90705.
SPECIALPREFIX #81016. 81147. 81151. 81359.
SOFFSE #83227. 83227. 83227. 83227. 83227. 83227. 83227. 83227.
SYL #83113. 83113. 83113. 83113. 83113. 83113. 83113. 83113.
SYL #83113. 83113. 83113. 83113. 83113. 83113. 83113. 83113.
SYL #83113. 83113. 83113. 83113. 83113. 83113. 83113. 83113.
SYL #83113. 83113. 83113. 83113. 83113. 83113. 83113. 83113.
SYL #83113. 83113. 83113. 83113. 83113. 83113. 83113. 83113.
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=namel SPECIFIED ON IF BLOCK name2</td>
<td>The current IF block (whose name is name2) included ELSE=namel 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>
<tr>
<td>8</td>
<td>STRC1501</td>
<td>ELSE BLOCK <em>elsename</em> NOT FOUND</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>The operand ELSE=<em>elsename</em> was coded on the IF macro, but the FI has occurred before the ELSE occurred.</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>STRC2101</td>
<td>OPERANDS AFTER SECOND &quot;WHILE&quot; IGNORED</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>The keyword &quot;WHILE&quot; appears more than once in the DO's operands.</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>STRC2102</td>
<td>OPERANDS AFTER SECOND &quot;UNTIL&quot; IGNORED</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>The keyword &quot;UNTIL&quot; appears more than once in the DO's operands.</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>STRC2103</td>
<td>WARNING—&quot;WHILE,(BCT,...&quot; WILL LOOP ONE LESS TIME THAN VALUE IN REGISTER</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>STRC2104</td>
<td>WARNING—LOOPING BRANCH MAY NOT BE EXECUTED ON EVERY ITERATION</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>STRC2105</td>
<td>TWO LOOPING BRANCHES INVALID IN &quot;DO&quot;—&quot;WHILE&quot; IGNORED</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Both the WHILE and UNTIL looping groups contain looping branches (BCTs, BXHs, or BXLE,); the WHILE looping group has been ignored.</td>
<td></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>
<tr>
<td>8</td>
<td>STRC2106</td>
<td>INVALID NUMBER OF OPERANDS FOR opcode</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>The looping branch <em>opcode</em> has the wrong number of operands; BCT should have one, BXLE or BXH should have two.</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>STRC2107</td>
<td>xxx INVALID AFTER LOOPING BRANCH—&quot;AND&quot; INSERTED</td>
<td></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>
<td></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>
<td></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>
<td></td>
</tr>
<tr>
<td>8</td>
<td>STRC2109</td>
<td>WHILE TEST IS VOID—IGNORED</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>No looping group follows the keyword &quot;WHILE&quot;; the keyword has been discarded.</td>
<td></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>
<td></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>
<td></td>
</tr>
<tr>
<td>8</td>
<td>STRC2111</td>
<td>UNTIL TEST IS VOID—IGNORED</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>No looping group follows the keyword &quot;UNTIL&quot;; the keyword has been discarded.</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>STRC2112</td>
<td>PARENTHESES OMITTED AROUND opcode</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>The looping branch <em>opcode</em> was not specified as a sublist.</td>
<td></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>
<tr>
<td></td>
<td></td>
<td><strong>Severity</strong> Message-Number Explanation</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>STRC2113</td>
<td>REL= OR MASK= NOT IN PARENTHESES—IGNORED</td>
<td></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>
<td></td>
</tr>
<tr>
<td>8</td>
<td>STRC2114</td>
<td>SUPERFLUOUS LOOPING GROUP IGNORED</td>
<td></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>
<td></td>
</tr>
<tr>
<td>8</td>
<td>STRC2301</td>
<td>MORE THAN ONE &quot;ATEND&quot; IN BLOCK</td>
<td></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>
<td></td>
</tr>
<tr>
<td>8</td>
<td>STRC2501</td>
<td>MORE THAN ONE &quot;ONEXIT&quot; IN BLOCK</td>
<td></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>
<td></td>
</tr>
<tr>
<td>8</td>
<td>STRC2502</td>
<td>NO EXIT FOR THIS &quot;DO&quot;</td>
<td></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>
<td></td>
</tr>
<tr>
<td>4</td>
<td>STRC3101</td>
<td>WARNING—xxx ASSUMED AS INDEX: USE &quot;DOCASE,xxx&quot; FOR RANGE SPEC</td>
<td></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>
<td></td>
</tr>
<tr>
<td>8</td>
<td>STRC3102</td>
<td>xxx INVALID SECOND OPERAND—IGNORED</td>
<td></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>
<td></td>
</tr>
</tbody>
</table>

D-4
<table>
<thead>
<tr>
<th>Severity</th>
<th>Message-Number</th>
<th>Message-Text</th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>
| 8                | STRC3305       | **xxx INVALID—yyy ASSUMED**  
xxx appears as the operand of a CASE macro, but the format is invalid. Usually this occurs when a range is coded whose second value is less than the first (such as "(15,10)"). The operand yyy replaces xxx. |
| 4                | STRC3306       | **EARLIER UNEXPECTED OPERAND IMPLIES THIS TO BE CASE xxx**  
CASE macro has no operands but earlier CASEs for the same DOCASE contained operands other than their ordinal position numbers. The operand xxx has been assumed. |
| 8                | STRC3307       | **OPERAND INVALID VALUE ON SIMPLE CASE xxx**  
The ordinal position number of this CASE is xxx, but SIMPLE was coded on the DOCASE and an operand other than xxx on the CASE. The operand is ignored. |
| 8                | STRC3308       | "DOCASE...,ONLY" INVALID WITH MISC  
A CASE MISC has been found in a DOCASE with the ONLY range option. Since these are mutually exclusive, the ONLY has been ignored. |
| 8                | STRC3309       | **OPERAND MUST BE SELF-DEFINING TERM OR OMITTED ON SIMPLE CASE xxx**  
The ordinal position number of this CASE is xxx and SIMPLE was coded on the DOCASE, but an operand has been specified which is not a self-defining term. It has been ignored. |
| 8                | STRC3310       | **REL= OR MASK= NOT IN PARENTHESES—IGNORED**  
The REL= and MASK= operands must be a part of a simple conditional and thus must be inside parentheses. The keyword has been ignored. |
8 STRC3311  MULTIPLE MISC CASES IN THIS DOCASE—
   THIS BLOCK IS DEAD CODE
More than one "CASE MISC" has occurred in the
same DOCASE. Only the first is executable.

8 STRC3312  xxx INVALID—ONLY FIRST TWO SUBOPERANDS
   PROCESSED
xxx is an operand list containing more than two sub-
operands in a CASE macro for a DOCASE which
specified an index. Operands after the first two
are ignored.

* STRC3313  CASE DEBUG ID=X'hh'
Debug block counts are being kept for this CASE
macro. When executed, this block will store
X'hh' into the last-case variable in the immedi-
ately surrounding DOCASE block.

8 STRC3701  DOCASE CONTAINS NO VALID CASES
No valid CASE macros were found as immediate
sub-blocks of the DOCASE.

8 STRC4301  NO BLOCKS ACTIVE—"BLEND" IGNORED
A BLEND macro was coded but no blocks were
active (the current nest level was zero). The
macro has been ignored.

8 STRC4302  NO BLOCK ACTIVE NAMED xxx— "BLEND" IGNORED
A BLEND macro was issued for block xxx, but no
block by that name is active.

8 STRC5301  BLEND OF biname ASSUMED
The block biname has not yet been terminated; a
BLEND is being issued for it by the FINAL macro.
<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>STRC8101</td>
<td>LINKAGE=xxx INVALID-&quot;OS&quot; ASSUMED</td>
<td>The first suboperand of the LINKAGE= keyword is invalid; &quot;OS&quot; has been substituted.</td>
</tr>
<tr>
<td>8</td>
<td>STRC8102</td>
<td>SECOND LINKAGE OPERAND IGNORED</td>
<td>The second suboperand of the LINKAGE= keyword must be either &quot;CSECT&quot; or omitted. It is invalid and has been ignored.</td>
</tr>
<tr>
<td>4</td>
<td>STRC8103</td>
<td>WARNING—SAVETRACE REQUIRES &quot;FINAL&quot; MACRO</td>
<td>The SAVETRACE debug option has been specified on the first PROC; warning is printed to indicate need for FINAL macro.</td>
</tr>
<tr>
<td>8</td>
<td>STRC8104</td>
<td>DEBUG=xxx INVALID—IGNORED</td>
<td>An invalid debug specification is present; the first eight characters of the invalid operand are listed as xxx. That option is ignored.</td>
</tr>
<tr>
<td>8</td>
<td>STRC8105</td>
<td>SAVETRACE MUST BE SPECIFIED ON FIRST PROC</td>
<td>The SAVETRACE debug option must be enabled on the first PROC. The operand has been ignored.</td>
</tr>
<tr>
<td>8</td>
<td>STRC8106</td>
<td>SAVETRACE REQUIRES FIRST PROC TO BE LINKAGE=OS</td>
<td>The SAVETRACE debug option is valid only if the first PROC includes the LINKAGE=OS specification. The operand has been ignored.</td>
</tr>
<tr>
<td>4</td>
<td>STRC8107</td>
<td>REG 1 MUST BE AMONG THOSE SAVED</td>
<td>Register 1 was destroyed during the GETMAIN for a dynamic save area and registers 14 through 12 were not specified (or defaulted) as being saved and WORK=NONE was specified. No further check is made to assure register one was among those saved.</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>
<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.</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>STRC8108</td>
<td>PROC proc-name, DEBUG ID=X'hh'</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>The save-trace, proc-trace, or proc-count debug option has been specified for this proc. The hex id byte hh will be used to identify this proc in the labels and dumps. This message is generated so that the user will know the proc id number even if &quot;PRINT NOGEN&quot; has been specified.</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>STRC8109</td>
<td>REGISTER 13 IS INVALID—IGNORED</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>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>
<td></td>
</tr>
<tr>
<td>8</td>
<td>STRC8301</td>
<td>NO REGISTERS SAVED—RESTORE IGNORED</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>RESTORE= was coded on the CORP macro, but SAVE=NONE was coded on the PROC. The operand is ignored.</td>
<td></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</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>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>
<td></td>
</tr>
<tr>
<td>0</td>
<td>STRC8303</td>
<td>ONE OR MORE EXIT'S REFERENCE THIS POINT</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>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>
<td></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>
<tr>
<td>8</td>
<td>STRC9103</td>
<td>EXIT TO IMMEDIATELY SURROUNDING BLOCK INVALID</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>The block name specified as the EXIT target is the block immediately</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>surrounding the EXIT.</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>STRC9104</td>
<td>EXIT TO DO BLOCK INVALID WITHIN ATEND OR ONEXIT</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>An EXIT macro specifies (or implies) a DO block as its target, but the</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>EXIT macro is nested within the ATEND or ONEXIT segment of the DO. The</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>macro is ignored.</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>STRC9201</td>
<td>ONE OR MORE EXIT'S REFERENCE THIS POINT</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>This block is the target of one or more EXIT macros (or EXIT= operands of</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>IF macros). The severity code of this message may be modified by specifying</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>the EXIT= operand of a PROC macro.</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>STRC9301</td>
<td>BLOCK NESTING LIMIT OF limit EXCEEDED—MACROS MUST BE MODIFIED</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>The current static nesting level has exceeded the stack limit in the</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>macros of limit. Either the program or macros must be modified.</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>STRC9302</td>
<td>NON-CASE BLOCK IMMEDIATELY SURROUNDED BY DOCASE INVALID</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>The block being defined is not a CASE block but is immediately surrounded</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>by a DOCASE block. The result is undefined.</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>STRC9401</td>
<td>INSUFFICIENT OPERANDS FOR TEST &quot;opcode&quot;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>The parenthesized list which is supposed to be a simple conditional contains</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>two items, the operation code opcode and a bc-spec.</td>
<td></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>
<tr>
<td>8</td>
<td>STRC9402</td>
<td>SUPERFLUOUS OPERANDS FOR TEST &quot;opcode&quot;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>The parenthesized list which is supposed to be a simple conditional contains more than 5 items.</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>STRC9403</td>
<td>NO CONDITION SPECIFIED—&quot;MASK=0&quot; ASSUMED</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>A void simple conditional has been specified.</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>STRC9601</td>
<td>ONE BLEND ASSUMED TO GET TO &quot;type&quot; BLOCK</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>STRC9602</td>
<td>TWO BLENDS ASSUMED TO GET TO &quot;type&quot; BLOCK</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>STRC9603</td>
<td>CURRENT BLOCK IS NOT &quot;type&quot; BLOCK—MACRO IGNORED</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>In a block terminating macro (such as FI), the current block was not of the corresponding type (such as IF).</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>STRC9604</td>
<td>NO ACTIVE BLOCK NAMED &quot;blname&quot;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>The request to terminate block blname has been ignored because no block named blname is in the nest.</td>
<td></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>
<tr>
<td>8</td>
<td>STRC9605</td>
<td>BLOCK &quot;blname&quot; IS NOT A type BLOCK—MACRO IGNORED</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>The request to terminate a block named blname of type type has been ignored</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>because the indicated block is of a different type.</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>STRC9606</td>
<td>END OF BLOCK &quot;blname1&quot; IMPLIES END OF BLOCK &quot;blname2&quot;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Request to terminate block blname1 must first terminate block blname2 which</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>is nested inside block blname1.</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>STRC9607</td>
<td>NO BLOCKS ACTIVE—MACRO IGNORED</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>The request to terminate a block has been ignored since no blocks are</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>outstanding.</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>STRC9701</td>
<td>INSUFFICIENT OPERANDS</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>The conditional expression ends with a logical operation (AND or OR)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>outstanding.</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>STRC9702</td>
<td>INSUFFICIENT BRACKETS</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>More left brackets (&quot;&lt;&quot; or &quot;+&quot;) than right brackets (&quot;-&gt;&quot; or &quot;/&quot;) are</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>present.</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>STRC9703</td>
<td>SYNTAX ERROR—LOOKING FOR &quot;AND&quot; OR &quot;OR&quot;, FOUND xxx</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>The operand xxx was found where a logical operator was expected.</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>STRC9704</td>
<td>SYNTAX ERROR—xxx INVALID</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Invalid operand xxx in conditional expression.</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>STRC9705</td>
<td>SUPERFLUOUS BRACKET IGNORED</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>More right brackets (&quot;&lt;&quot; or &quot;+&quot;) have been found in the conditional</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>expression than left brackets (&quot;-&gt;&quot; or &quot;/&quot;).</td>
<td></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>
<tr>
<td>8</td>
<td>STRC9706</td>
<td>SYNTAX ERROR—LOOKING FOR SIMPLE CONDITIONAL, FOUND &quot;xxx&quot;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>The operand xxx was found in a conditional expression where a simple</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>conditional sublist was expected.</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>STRC9902</td>
<td>START OF BLOCK nn (blname) AT DEPTH level</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>In response to the debug option LISTBLOCKS, the message indicates the start</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>of the block whose sequential number is nn. The block name is blname;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>if no name was specified on the block initiation macro, blname is an</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>internal name of the form BLKnn. The current static nest level is level.</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>STRC9903</td>
<td>END OF BLOCK nn (blname) AT DEPTH level</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>In response to the debug option LISTBLOCKS, the message indicates the end</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>of the block whose sequential number is nn. The block name is blname;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>if no name was specified on the block initiation macro, blname is an</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>internal name of the form BLKnn. The current static nest level is level.</td>
<td></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) RUF DRAFT</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 RUF DRAFT. It is provided for those installations which do not have a TN (upper and lower case) print chain available. RUF DRAFT will translate the SIMPL-M listings of data set three for printing on HN, PN, or QN print trains. For instructions on using RUF DRAFT, 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 1</th>
<th>Page 2</th>
<th>Page 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>abnormal block exit</td>
<td>27</td>
<td></td>
<td>31, A-10</td>
</tr>
<tr>
<td>abstract programming</td>
<td>8</td>
<td></td>
<td>35, A-30</td>
</tr>
<tr>
<td>language</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addressability</td>
<td>42</td>
<td></td>
<td></td>
</tr>
<tr>
<td>angle brackets</td>
<td>19, A-2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>asynchronous IF block</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ATEND</td>
<td>29, A-3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ATEXIT</td>
<td>30, A-4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>base register</td>
<td>34, A-32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bc-spec</td>
<td>A-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BCT</td>
<td>21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BLEND</td>
<td>16, 19, 23, 27, 38, A-5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BLISS</td>
<td>6, 7, 10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BLOCK</td>
<td>16, A-6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>block counts</td>
<td>40</td>
<td></td>
<td></td>
</tr>
<tr>
<td>block exit</td>
<td>27</td>
<td></td>
<td></td>
</tr>
<tr>
<td>blocks</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BXH</td>
<td>21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BXLE</td>
<td>21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CASE</td>
<td>7, 23, A-7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>case ranges</td>
<td>24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CASEEND</td>
<td>27, A-9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COBOL</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>conditional expression</td>
<td>19, 21, 26, A-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>conditional relations</td>
<td>19</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ E \ - \ 3 \]
### INDEX (continued)

<table>
<thead>
<tr>
<th>Term</th>
<th>Page</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESACOD</td>
<td>23, A-20</td>
<td>looping branches</td>
<td>21, A-13</td>
</tr>
<tr>
<td>evoking procs</td>
<td>36-38</td>
<td>looping group</td>
<td>A-13</td>
</tr>
<tr>
<td>EXIT</td>
<td>27, A-21</td>
<td>messages</td>
<td>D-1</td>
</tr>
<tr>
<td>exit severity</td>
<td>38</td>
<td>MISC</td>
<td>23, A-7</td>
</tr>
<tr>
<td>FI</td>
<td>18, A-22</td>
<td>miscellaneous case</td>
<td>23, A-7</td>
</tr>
<tr>
<td>FINAL</td>
<td>38, A-23</td>
<td>modifying structured programs</td>
<td>13</td>
</tr>
<tr>
<td>FOREVER</td>
<td>28, A-12</td>
<td>modularity</td>
<td>13</td>
</tr>
<tr>
<td>FORTRAN</td>
<td>10</td>
<td>modules</td>
<td>30</td>
</tr>
<tr>
<td>GEDANKEN</td>
<td>10</td>
<td>multiple decisions</td>
<td>23</td>
</tr>
<tr>
<td>goto controversy</td>
<td>3</td>
<td>nest level</td>
<td>17</td>
</tr>
<tr>
<td>goto-less programming</td>
<td>1, 2-10</td>
<td>nesting blocks</td>
<td>17</td>
</tr>
<tr>
<td>IF</td>
<td>18, 28, A-24</td>
<td>OD</td>
<td>21, A-27</td>
</tr>
<tr>
<td>IFANY</td>
<td>27, A-15</td>
<td>ONEND</td>
<td>30, A-28</td>
</tr>
<tr>
<td>IFEND</td>
<td>19, A-26</td>
<td>ONEXIT</td>
<td>29, A-29</td>
</tr>
<tr>
<td>immediately</td>
<td></td>
<td>ONLY</td>
<td>27, A-15</td>
</tr>
<tr>
<td>surrounding</td>
<td>17</td>
<td>operand format for DO</td>
<td>B-7</td>
</tr>
<tr>
<td>in-line identifier</td>
<td>34, A-31</td>
<td>OREGANO</td>
<td>10</td>
</tr>
<tr>
<td>infinite loop</td>
<td>28</td>
<td>PL/I</td>
<td>10</td>
</tr>
<tr>
<td>installing STRCMACS</td>
<td>E-1</td>
<td>PROC</td>
<td>31, A-30</td>
</tr>
<tr>
<td>ISWIM</td>
<td>10</td>
<td>OS vs non-OS</td>
<td>31</td>
</tr>
<tr>
<td>iteration</td>
<td>21</td>
<td>procs</td>
<td>30</td>
</tr>
<tr>
<td>labels</td>
<td>42</td>
<td>PROCEND</td>
<td>39, A-35</td>
</tr>
<tr>
<td>link register</td>
<td>36, A-11</td>
<td>properly nested</td>
<td>17</td>
</tr>
<tr>
<td>linkage of procs</td>
<td>36-38</td>
<td>proving program</td>
<td></td>
</tr>
<tr>
<td>LISP</td>
<td>10</td>
<td>correctness</td>
<td>6</td>
</tr>
<tr>
<td>literals</td>
<td>26, 42</td>
<td>quasi-goto-less code</td>
<td>3, 8</td>
</tr>
</tbody>
</table>
INDEX (continued)

<table>
<thead>
<tr>
<th>Term</th>
<th>Page(s)</th>
<th>Term</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>range option</td>
<td>27</td>
<td>SIMPL-M (con't)</td>
<td>B-4</td>
</tr>
<tr>
<td>recursive code</td>
<td>34, 42</td>
<td>implied relations</td>
<td>B-3</td>
</tr>
<tr>
<td>reentrant code</td>
<td>34, 42</td>
<td>initialization</td>
<td>B-2</td>
</tr>
<tr>
<td>register save areas</td>
<td>32, A-31</td>
<td>K'</td>
<td>B-4</td>
</tr>
<tr>
<td>user supplied</td>
<td>33</td>
<td>macros</td>
<td>B-1</td>
</tr>
<tr>
<td>dynamic</td>
<td>33</td>
<td>N'</td>
<td>B-4</td>
</tr>
<tr>
<td>length of</td>
<td>33</td>
<td>procs</td>
<td>B-1</td>
</tr>
<tr>
<td>CORPVALUES</td>
<td>41</td>
<td>SYSLIST</td>
<td>B-4</td>
</tr>
<tr>
<td>BACK</td>
<td>41</td>
<td>T'</td>
<td>B-4</td>
</tr>
<tr>
<td>restoring registers</td>
<td>32, A-10</td>
<td>type conversion</td>
<td>B-2</td>
</tr>
<tr>
<td>return address register</td>
<td>36, A-10</td>
<td>SIMPL-X</td>
<td>10, B-1</td>
</tr>
<tr>
<td>return code</td>
<td>35, A-10</td>
<td>simple conditionals</td>
<td>18</td>
</tr>
<tr>
<td>returning values</td>
<td>32, A-10</td>
<td>stepwise refinement</td>
<td>1,</td>
</tr>
<tr>
<td>RUF DRAFT</td>
<td>E-1</td>
<td></td>
<td>11-13</td>
</tr>
<tr>
<td>save areas</td>
<td>32, A-31</td>
<td>STRCMACS</td>
<td>E-1</td>
</tr>
<tr>
<td>user supplied</td>
<td>33</td>
<td>super-instructions</td>
<td>12</td>
</tr>
<tr>
<td>dynamic</td>
<td>33</td>
<td>surrounding</td>
<td>17</td>
</tr>
<tr>
<td>length of</td>
<td>33</td>
<td>top-down programming</td>
<td></td>
</tr>
<tr>
<td>CORPVALUES</td>
<td>41</td>
<td>see stepwise refinement</td>
<td></td>
</tr>
<tr>
<td>BACK</td>
<td>41</td>
<td>trace vector</td>
<td>40</td>
</tr>
<tr>
<td>SIMPL-M arrays</td>
<td>8, B-1</td>
<td>UNTIL</td>
<td>21</td>
</tr>
<tr>
<td>assignments</td>
<td>B-3</td>
<td>USING</td>
<td>35</td>
</tr>
<tr>
<td>comments</td>
<td>B-4</td>
<td>WHILE</td>
<td>21</td>
</tr>
<tr>
<td>data types</td>
<td>B-2</td>
<td>work register</td>
<td>35, A-33</td>
</tr>
</tbody>
</table>

E - 5