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

PRICES SUBJECT TO CHANGE

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>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>GOTO-LESS PROGRAMMING</td>
<td>2</td>
</tr>
<tr>
<td>STEPWISE REFINEMENT</td>
<td>11</td>
</tr>
<tr>
<td>EFFICIENCY OF STRUCTURED CODE.</td>
<td>14</td>
</tr>
<tr>
<td>INTRODUCTION TO STRCMACS</td>
<td>16</td>
</tr>
<tr>
<td>Defining blocks</td>
<td>16</td>
</tr>
<tr>
<td>Decision making</td>
<td>18</td>
</tr>
<tr>
<td>Iteration</td>
<td>21</td>
</tr>
<tr>
<td>Multiple decisions</td>
<td>23</td>
</tr>
<tr>
<td>Abnormal block exit</td>
<td>27</td>
</tr>
<tr>
<td>Defining modules</td>
<td>30</td>
</tr>
<tr>
<td>Special services</td>
<td>38</td>
</tr>
<tr>
<td>STRCMACS debugging aids</td>
<td>38</td>
</tr>
<tr>
<td>Addressability, labels, and reentrant code</td>
<td>42</td>
</tr>
<tr>
<td>APPENDIX A — MACRO INSTRUCTION DESCRIPTIONS</td>
<td>A-1</td>
</tr>
<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>CASEND</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>DEND</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>IF</th>
<th>A-24</th>
</tr>
</thead>
<tbody>
<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 if and while) replacing the branch instruction ("goto-less programming"); the other is the process of developing a program by progressively refining descriptions of components in terms of more primitive components (called "stepwise refinement" or "top-down programming"). In addition to discussing what these techniques are, we will try to show why their use is advised and how both can be implemented in OS assembly language by the use of a special macro instruction package.

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

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

---

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

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

```
IF (I-5) 10, 20, 20
10 J = 5
GO TO 30
20 J = 1
30 CONTINUE
```

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

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

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

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

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

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

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

IF (q) GO TO 20
IF (r) GO TO 10
A
GO TO 60

10 B
GO TO 60

20 IF (s) GO TO 80
C
D
E

30 IF (.NOT. (t)) GO TO 40
F
GO TO 30

40 IF (u) GO TO 70
G

50 IF (.NOT. (v)) GO TO 60
H

60 I
RETURN

70 J
GO TO 50

80 IF (w) GO TO 90
C
D
GO TO 110

90 C
D
K

100 IF (.NOT. (x)) GO TO 110
L
GO TO 100

110 IF (.NOT. (w)) GO TO 120
M

120 IF (.NOT. (v)) GO TO 60
H
GO TO 60
goto-less programming

IF q
THEN
C
D
IF s
THEN
IF w
THEN
K
WHILE x
DO
L
OD
M

IF v
THEN
H
FI

ELSE

IF u
THEN
J
ELSE
G
FI
IF v
THEN
H
FI

ELSE

IF r
THEN
B
ELSE
A
FI
FI
I
in goto-less form. Alan Perlis has referred to similar situations as the 'Turing Tarpit' in which everything is possible, but nothing is easy.\textsuperscript{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."\textsuperscript{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.

\textsuperscript{2}Ibid. p. 795.
It should be mentioned that the languages in which goto-less is really feasible are more than bare-minimum languages. The theoretical considerations show that the required constructs are:

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

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

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

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

```

: OUTER: begin;

: INNER: begin;

  : if I \neq 0 then begin; alpha end;
  end;

  : if I \neq 0 then begin; beta end;
  end;

  : gamma
```

Notice, however, that by using the 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 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\textsubscript{1} for some abstract super-machine with arbitrarily complex instructions. Most of the instructions of B\textsubscript{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\textsubscript{1}, it would surely accomplish the task A. We can now take the instructions of B\textsubscript{1} (call them the A\textsubscript{2,i}) and for whichever are not understandable to our real computer repeat the problem-solving process producing a program B\textsubscript{2,i} — the "how" for each A\textsubscript{2,i} in more primitive terms. This process is continued until eventually all instructions are in terms intelligible to our computer.

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

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

By using this method of programming, the modules developed during designing are both the natural modules for coding and also the modules of documentation. By limiting each module to about 50 lines (one page), one not only helps such typographical aid as the indentation of control structures but also limits the breadth of the activity of the module to a reasonable size, improving the overall intelligibility of the program.
Designing a large program from the top down is not all new; nor is the breaking of code into modules. Such techniques have been used under the name "modularity" for some time. The extension here is to break up the modules by stepwise refinement and code them in the same fashion. In addition, the modularity is carried down to much lower levels. The requirements ("what") of each module are well-defined and the method by which these requirements are fulfilled (the "how") is limited in detail to about a page.

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

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

---

EFFICIENCY OF STRUCTURED CODE

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

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

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

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

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

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

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

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

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

Defining blocks

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

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

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

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

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

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

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

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

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

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

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

Decision making

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

(LTR,3,3,Z)

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

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

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

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

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

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

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

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

- H or GT (High) or N H or LE (Not High)
- L or LT (Low) or N L or GE (Not Low)
- E or EQ (Equal) or NE (Not Equal)
- O (Ones or Overflow) or NO (Not Ones or Not Overflow)
- P (Plus) or NP (Not Plus)
- M (Minus) or NM (Not Minus)
- Z (Zero or Zeros) or NZ (Not Zero or Not Zeros)

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

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

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

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

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

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

The entire QTEST block is bypassed unless either register 5 is zero or register 3 differs from the halfword at HWORD. If the QTEST block is executed, another conditional expression is evaluated at MORECHK. Note the use of angle brackets to group operands. These must be coded as separate macro operands—i.e.,
\(\langle \text{CR}, 7, 5, E\rangle, \text{OR}, \langle \text{SR}, 3, 1, Z\rangle\) 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 "\((\text{CR}, 7, 5, E), \text{OR}, \langle, (\text{SR}, 3, 1, Z), \text{AND}, (\text{LTR}, 1, 1, \text{MASK=SYMMAK}), \rangle\)".) 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
   .
   .
IN DCB ...,EODAD=INPUTEND
```
The asynchronous IF generates an unconditional branch around the block. Note that if a label occurs on the IF macro, it will be defined on the branch instruction. As a consequence, the label "INPUTEND" is specified on the first instruction inside the block rather than on the IF macro itself.

**Iteration**

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

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

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

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

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

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

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

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

The 360/370 provides three instructions which are particularly well suited for the construction of loops: EXH, BXLE, and BCT. Use of these looping
branches is provided for in the DO macro either in place of or in addition to conditional expressions.

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

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

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

The above loop will execute only four times.

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

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

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

```asm
X  B  α
γ  LTR 4,4
    BNZ α
    TM  FLAG,X'80'
    BZ  β
α  DS 0H
block code
    BCT 5,γ
β  DS 0H
```
Appendix B shows the code generated for all possible combinations of DO operands.

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

Multiple decisions

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

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

Example:

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

24
FIVE EQU 5
FOURTEEN EQU X'E'
SIXTEEN EQU FIVE+11

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

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

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

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

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

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

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

In addition, the DOCASE macro will automatically optimize for the case where all of the CASE macros specify operands which are exact multiples of 4.
Another form of the DOCASE allows the selection to be performed on the basis of character strings. The CASE macros may specify selection values in any of the ways shown:

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

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

```
DOCASE
  CASE   (LTR, 3, 3, Z)
  .
  .
ESAC
CASE   (CR, 1, 2, EQ), OR, (TM, FLAG, X'80', O)
  .
  .
ESAC
CASE   (S, 5, WORD, P)
  .
  .
ESAC
ESACOD
```
The conditional expressions are evaluated until one is found that is true. That case block is then executed and the rest are bypassed. Note that no index is specified.

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

One other pair of options is provided which is of use mainly when the DOCASE is implemented by a branch vector (that is, when an index is specified, neither SPARSE nor SIMPLE is specified, and one or more CASEs are for self-defining terms in the range 0-255) and no miscellaneous case is present, although it may be specified in any DOCASE. The options IFANY or ONLY may be coded as the second or third operand. When IFANY is specified, code is included to bypass all CASE blocks if the index is out of the range of the branch vector. ("Do case I, if any such case exists; else do nothing.") When ONLY is coded, the range test is not included and the result if the index takes on an out-of-range value is undefined — and invariably 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\nSCAN    DO     WHILE, (TM, FLAG, ENDOCARD, Z)
    :     :
WHOOPS  IF     (CLI, DELIMITR, C', ', NE)
          \ Print 'BAD DELIMITER' message\n           EXIT
FI
    :
OD
OD    DOINFILE
In this code segment, a delimiter other than the comma will cause abnormal termination of the control card scan loop after printing a message. Since the EXIT macro has no operand, the exit is to the end of the block containing the block containing the EXIT (the surrounding block whose nest level is one less than the current nest level; in our example, the block SCAN). Any surrounding block's name may be specified as the EXIT's operand to cause transfer to the end of that block. In our example, adding the operand DOINFILE to the EXIT would skip the rest of the input when the error occurred.

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

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

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

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

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

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

```
DO FOREVER
```
or just

```
DO
```

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

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

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

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

SAVE=3

or

SAVE=(15,7)

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

```plaintext
PROC   SAVE=(R5, LAST)

R5   EQU 5
LAST  EQU 9
```

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

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

```plaintext
CORP  RESTORE=(first, last)
```
or limited to a single saved register by coding

```plaintext
CORP  RESTORE=reg
```

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

```plaintext
X  PROC

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

\[
\text{CORP RESTORE}=(2,7), \text{RETURN}=(5,9)
\]

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:

\[
\text{Z PROC LINKAGE=OS,SAVE}=,(,\text{MYSAVE})
\]

\[
\text{PROC1 PROC LINKAGE=OS,SAVE}=(14,2,\text{MYSAVE})
\]

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

\[
\text{SAVE}=(,\text{DYNAM})
\]

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:

\[
\text{RENTPROC PROC LINKAGE=OS,SAVE}=(,\text{DYNAM,WORKSIZE})
\]

\[
\text{USING WORKSECT,13}
\]
WORKSECT DSECT
DS 18F New save area.

... Other work variables.

WORKSIZE EQU (*-WORKSECT+3)/4, Length, in words, rounded up.

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

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

ID=char-string

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

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

PROC BASE=7

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

PROC BASE=(7,8,9)

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

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

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

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

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

X PROC LINKAGE=(OS,CSECT)

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

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

PROC WORK=5

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

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

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

   CORP RC=(reg)

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

The last instruction normally generated by a CORP is a

   BR 14

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

   CORP LINK=linkreg

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

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

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

   B PROC
       .
       .
       .
       BAL 14,C
       .
       .
       .
       CORP
The alternate method is to define the procs as user macros to perform the linkage at assembly time:

MACRO
BMAC
B PROC
CMAC (*)
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

B PROC DEBUG=(BLOCKNAMES, PROCTRACE, GBL)

C PROC DEBUG=(NOPROCTRACE, PROCCOUNTS)

D PROC DEBUG=(ALL, NOSAVETRACE)

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

We will now discuss each of the options in turn.

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

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

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

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

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

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

and is generated in the first proc which requests PROCTRACE.

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

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

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

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

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

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

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

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

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

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

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

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

**Addressability, labels, and reentrant code**

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

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

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

SUBPRG  PROC  LINKAGE=(DS, CSECT), DEBUG=(ALL, LOCAL, BLOCKS, GLOBAL)

    LA  5, R
    DO  UNTIL, (RCT, 3)
        BAL 14, SUBK
        BAL 14, SUBZ
    OD

*               :
               :
CORP  EJECT
SUBK  PROC  SAVE=(3, 5)

*               :
               :
L     3, XID

*               :
               :
CORP  SUBK  EJECT
SUBY  PROC  DEBUG=NDCOPVALUES

*               :
               :
L     3, YID
       CALL  NEXTPRG

*               :
               :
CORP  RETURN=3  EJECT
SUBZ  PROC

*               :
               :
L     3, ZID
LRI  5, 3
    IF  (C.5=F'11', EQ)
        BAL 14, SUBY
    FI

*               :
               :
CORP  RETURN=6  EJECT
FINAL

DS  OF
XID  DC  C'XXXX'
YID  DC  C'YYYY'
ZID  DC  C'ZZZZ'
LTORG
SPACE 3
END

43
### 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>E7FC D1DA</td>
<td>0012E</td>
<td></td>
<td>61 SUBI</td>
<td>PROC SAVE=(3,5)</td>
</tr>
<tr>
<td>0001EA</td>
<td>E28C257</td>
<td>64+</td>
<td>DC</td>
<td>B $PO2AA</td>
<td>* STBC0108 PROC SUBX, DEBUG ID=X'02'</td>
</tr>
<tr>
<td>0001EE</td>
<td>90BC D1P0</td>
<td>3D204</td>
<td>64+</td>
<td>DC CLA'SUBX'</td>
<td></td>
</tr>
<tr>
<td>0001F2</td>
<td>47FC D22C</td>
<td>0D490</td>
<td>66+</td>
<td>DC $PO2DD</td>
<td></td>
</tr>
<tr>
<td>0001F6</td>
<td>PP020000</td>
<td>67+</td>
<td>DS</td>
<td>DP</td>
<td></td>
</tr>
<tr>
<td>0001FA</td>
<td>0000000000000270</td>
<td>68+</td>
<td>DC</td>
<td>X'020000'</td>
<td></td>
</tr>
<tr>
<td>000240</td>
<td>0000000000000000</td>
<td>71+</td>
<td>DC</td>
<td>(15)'0'</td>
<td></td>
</tr>
<tr>
<td>000244</td>
<td>E28C257</td>
<td>0D490</td>
<td>72+</td>
<td>L 1,=A($TRACE)</td>
<td></td>
</tr>
<tr>
<td>000248</td>
<td>9202 1100</td>
<td>00100</td>
<td>73+</td>
<td>MVC 0(256,1),1(1)</td>
<td></td>
</tr>
<tr>
<td>000252</td>
<td>4111 0001</td>
<td>001FA</td>
<td>74+</td>
<td>MVI 256(1),X'02'</td>
<td></td>
</tr>
<tr>
<td>000256</td>
<td>4010 1155</td>
<td>001FA</td>
<td>75+</td>
<td>LA 1,$PO2SV+2</td>
<td></td>
</tr>
<tr>
<td>00025A</td>
<td>92FC D184</td>
<td>001FA</td>
<td>77+</td>
<td>STH 1,$PO2SV+2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5830 D484</td>
<td>00498</td>
<td>78+</td>
<td>MVI $PO2SV,X'FE'</td>
<td></td>
</tr>
<tr>
<td>00025E</td>
<td>90BC D260</td>
<td>0027C</td>
<td>79+</td>
<td>:</td>
<td></td>
</tr>
<tr>
<td>000264</td>
<td>92FC D264</td>
<td>001FA</td>
<td>80+</td>
<td>:</td>
<td></td>
</tr>
<tr>
<td>000266</td>
<td>922F D184</td>
<td>0027C</td>
<td>81</td>
<td>L 3,XID</td>
<td></td>
</tr>
<tr>
<td>00026A</td>
<td>9835 2200</td>
<td>001FA</td>
<td>82+</td>
<td>:</td>
<td></td>
</tr>
<tr>
<td>00026E</td>
<td>07PB</td>
<td>83+</td>
<td>:</td>
<td>:</td>
<td></td>
</tr>
<tr>
<td>000270</td>
<td>FC020000</td>
<td>84</td>
<td>:</td>
<td>:</td>
<td></td>
</tr>
<tr>
<td>000274</td>
<td>000001F8000002C9</td>
<td>85+</td>
<td>L</td>
<td>3,5,$PO2SV+32</td>
<td></td>
</tr>
<tr>
<td>00027C</td>
<td>0000000000000000</td>
<td>86+</td>
<td>DC</td>
<td>X'FC020000'</td>
<td></td>
</tr>
<tr>
<td>00027E</td>
<td>000000000002C9</td>
<td>87+</td>
<td>DC</td>
<td>A($PO2SV,$PO2FWD)</td>
<td></td>
</tr>
<tr>
<td>000280</td>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
<td></td>
</tr>
</tbody>
</table>

**Page 3**

23 JAN 74
### Example of Debug Facilities

<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJECT CODE</th>
<th>ADDR1</th>
<th>ADDR2</th>
<th>STMT</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>000288</td>
<td>47F0 D2AC</td>
<td>002C0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0002BC</td>
<td>E22W288</td>
<td></td>
<td>0324</td>
<td>99+SP03AA</td>
<td>STM 14,12,SP03SV+12</td>
</tr>
<tr>
<td>0002C4</td>
<td>47F0 D2FC</td>
<td>00310</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0002C8</td>
<td></td>
<td>101+SP03SV</td>
<td>DS</td>
<td>OP</td>
<td></td>
</tr>
<tr>
<td>0002C9</td>
<td>FF030000</td>
<td>102+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0002C9</td>
<td></td>
<td>103+SP02PDW</td>
<td>EQU</td>
<td>SP03SV</td>
<td></td>
</tr>
<tr>
<td>0002CC</td>
<td>0000027000000360</td>
<td>104+</td>
<td></td>
<td>DC A(SP02CRP,SP03NXT)</td>
<td></td>
</tr>
<tr>
<td>0002D4</td>
<td>0000000000000000</td>
<td>105+</td>
<td></td>
<td>DC (15)F'0'</td>
<td></td>
</tr>
<tr>
<td>000310</td>
<td>5810 D49C</td>
<td>00480</td>
<td>106+SP03DD</td>
<td>L</td>
<td>,A(STRACE)</td>
</tr>
<tr>
<td>000314</td>
<td>92FF 1000 1001 00000</td>
<td>00001</td>
<td>107+</td>
<td>MVC</td>
<td>0(256,1),1(1)</td>
</tr>
<tr>
<td>00031A</td>
<td>9203 1000</td>
<td>00130</td>
<td>108+</td>
<td>SPF</td>
<td>256(1),X'03'</td>
</tr>
<tr>
<td>00031B</td>
<td>4810 D288</td>
<td>002CA</td>
<td>109+</td>
<td>LA</td>
<td>1,1(1)</td>
</tr>
<tr>
<td>000322</td>
<td>4111 0001</td>
<td>00001</td>
<td>110+</td>
<td>MST</td>
<td>1,SP03SV+2</td>
</tr>
<tr>
<td>000326</td>
<td>4010 D286</td>
<td>002CA</td>
<td>111+</td>
<td>MVI</td>
<td>SP03SV,X'0E'</td>
</tr>
<tr>
<td>00032A</td>
<td>9200 D2B4</td>
<td>002C8</td>
<td>112+</td>
<td>DC</td>
<td></td>
</tr>
<tr>
<td>00032E</td>
<td>5830 D488</td>
<td>0049C</td>
<td>113+</td>
<td>MKP</td>
<td></td>
</tr>
<tr>
<td>000332</td>
<td>0700</td>
<td>114+</td>
<td></td>
<td>CALL</td>
<td>NEXTPROG</td>
</tr>
<tr>
<td>000334</td>
<td>47F0 D329</td>
<td>003JC</td>
<td>115+</td>
<td>CNOP</td>
<td>0,4</td>
</tr>
<tr>
<td>000339</td>
<td>00000000</td>
<td>116+</td>
<td></td>
<td>RES</td>
<td></td>
</tr>
<tr>
<td>00033C</td>
<td>5800 D324</td>
<td>003J8</td>
<td>117+</td>
<td>ISP0040B</td>
<td>DC V(NEXTPROG) ENTRY POINT ADDRESS</td>
</tr>
<tr>
<td>000340</td>
<td>058F</td>
<td>118+</td>
<td></td>
<td>L</td>
<td>15,ISP0040B LOAD 15 WITH ENTRY ADR</td>
</tr>
<tr>
<td>000344</td>
<td>5030 D2D4</td>
<td>00288</td>
<td>119+</td>
<td>BPH</td>
<td>14,15 BRANCH TO ENTRY POINT</td>
</tr>
<tr>
<td>000346</td>
<td>92FF D284</td>
<td>002C8</td>
<td>120+</td>
<td>ST</td>
<td>3,SP03SV+32</td>
</tr>
<tr>
<td>000346</td>
<td>92FF D284</td>
<td>002C8</td>
<td>121+</td>
<td>MVI</td>
<td>SP03SV,X'FF'</td>
</tr>
<tr>
<td>000348</td>
<td>983C D2C0</td>
<td>002D4</td>
<td>122+</td>
<td>MVI</td>
<td>14,12,SP03SV+12</td>
</tr>
<tr>
<td>00034E</td>
<td>07FB</td>
<td>123+</td>
<td></td>
<td>BR</td>
<td>14</td>
</tr>
</tbody>
</table>
EXAMPLE OF DEBUG FACILITIES

<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJECT CODE</th>
<th>ADDR1</th>
<th>ADDR2</th>
<th>STMT</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>000350</td>
<td>47FP D34</td>
<td>00358</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000354</td>
<td>824C D39</td>
<td>0036C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00035C</td>
<td>47FP D34</td>
<td>003A8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000360</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000364</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00036C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00039A</td>
<td>5810 D34</td>
<td>004A0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00039C</td>
<td>5950 D40</td>
<td>004B4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00039E</td>
<td>4B10 D34</td>
<td>003B0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0003A4</td>
<td>4111 D01</td>
<td>003B1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0003BA</td>
<td>9204 D34</td>
<td>00362</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0003C2</td>
<td>92FF D34</td>
<td>00360</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0003C6</td>
<td>5830 D4C</td>
<td>004A0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0003CA</td>
<td>1863</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0003CC</td>
<td>5950 D40</td>
<td>004B4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0003DE</td>
<td>4111 D01</td>
<td>003B1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0003F0</td>
<td>9204 D34</td>
<td>00362</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0003FC</td>
<td>92FF D34</td>
<td>00360</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000400</td>
<td>9853 D2A</td>
<td>002B8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000408</td>
<td>458C D40</td>
<td>004A4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00040C</td>
<td>000036000000000</td>
<td>004C5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000414</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000416</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000418</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000420</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000428</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00043C</td>
<td>FC040000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000440</td>
<td>000000000000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00044C</td>
<td>000000000000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000450</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000454</td>
<td>PB040000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00045C</td>
<td>000000000000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* STRC0106 PROC SUBZ, DEBUG ID=X'04'

** SOURCE STATEMENT **

130 SUBZ PROC
131 *
132 SUBZ B $P04AA
133 DC CL$SUBZ
134 SP04AA STM 14,12,$          
135 B $P04DD
136 SP04SV DS OP
137 DC X'0040000' FLAG (FF=FINISHED, FE=ENTERED), ID, COUNT
138 SP03XNT EQU $P04SV
139 DC A($P04SV,$P03XNT)
140 DC (15)F'0'
141 SP04DD L 1,=A($TRACE)
142 MVC 0(256,1),1(1)
143 NMI 256(1),X'TOA'
144 LA 1,$P04SV+2
145 SP04SV STM 14,12,$          
146 DC X'0040000' FLAG (FF=FINISHED, FE=ENTERED), ID, COUNT
147 DC X'0040000', FLAG (FF=FINISHED, FE=ENTERED), ID, COUNT
148 MVI $P04SV,X'0040000'
149 B $6END
150 L 1,21D
151 LR 6,3
152 IF (C,5,F'11',IQ)
153 C 5,F'11'
154 BNE $6MND
155 L 1,56IFC
156 LA 1,1(1)
157 STH 1,$6IFC
158 B $6MND
159 DC C'BK6',OH'1'
160+$6IFC DC H'0' IF COUNT
161+$6GO DS OH
162 BAL 14,SUBY
163 FI
164+$6END DS OH
165 : ;
166 + : ;
167 COEP RETURN=6
168 STM 14,12,$P04CRP+12
169 MVC $P04BCY+12(15+4),SP04SV+12
170 ST 6,$P04BCY+12
171 NMI $P04SV,X'FF'
172 LM 14,12,$P04BCY+12
173 BR 76
174+$P04CRP DS OP
175 DC X'FC040000'
176+$P04XNT EQU $P04CRP
177 DC A($P04SV,$P04XNT)
178 DC (15)F'0'
179+$P04ACK DS OP
180 DC X'PB040000'
181 DC A($P04CRP,$P04FWD)
182 DC (15)F'0'
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>184</td>
<td>FINAL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000</td>
<td>185+$LASTSAV</td>
<td>EQU</td>
<td>$04BCK</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00496</td>
<td>186+$PO4PWD</td>
<td>EQU</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2727E727</td>
<td>187</td>
<td>DS</td>
<td>OF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3B288228</td>
<td>188</td>
<td>XID</td>
<td>DC</td>
<td>'XXIX'</td>
<td></td>
</tr>
<tr>
<td>09299989</td>
<td>189</td>
<td>YID</td>
<td>DC</td>
<td>'YYYY'</td>
<td></td>
</tr>
<tr>
<td>000418</td>
<td>190</td>
<td>ZID</td>
<td>DC</td>
<td>'ZZZZ'</td>
<td></td>
</tr>
<tr>
<td>00001F8</td>
<td>191</td>
<td>LTORG</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000450</td>
<td>192</td>
<td>=A($FIRSTSV)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00004500</td>
<td>193</td>
<td>=A($LASTSAV)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00004500</td>
<td>194</td>
<td>=A($TRACE)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00004500</td>
<td>195</td>
<td>=F'7'</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00052000</td>
<td>197</td>
<td>END</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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:

\[(\text{opcode, op1, \ldots, opn, bc-spec}) \text{ or } (\text{bc-spec})\]

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

A conditional expression consists of one or more simple conditionals separated by the logical connectors AND or OR (and also by the commas required
in macro syntax). In addition, angle brackets "<" and ">"* may be specified as operands for the grouping of subexpressions. For example:

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

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` sym

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

**ATEXIT—Define Abnormal Loop Termination Code**

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

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

A-4
BLEND—Terminate Current Block

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

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

\text{block-name}

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

\text{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.

<table>
<thead>
<tr>
<th>[blname]</th>
<th>BLOCK</th>
</tr>
</thead>
</table>

**blname**

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

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

<table>
<thead>
<tr>
<th>[blname]</th>
<th>CASE</th>
<th>MISC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>index-list</td>
</tr>
<tr>
<td></td>
<td></td>
<td>char-index-list</td>
</tr>
<tr>
<td></td>
<td></td>
<td>conditional-test</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).
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.

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

A-9
**CORP—Terminate a Procedure**

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

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

*label* 

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

*proc-name*

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

*RESTORE=(first,last)*

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

*RETURN=reg-list*

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

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

RC=value

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.

\[
\begin{align*}
\text{[biname]} & \quad \text{DO} & \quad \left\{ \begin{array}{c}
\text{FOREVER} \\
\text{WHILE,looping-group} & \left\{ \begin{array}{c}
\text{[AND]} \\
\text{OR}
\end{array} \right\}, \text{UNTIL,looping-group} \\
\text{UNTIL,looping-group} & \left\{ \begin{array}{c}
\text{[AND]} \\
\text{OR}
\end{array} \right\}, \text{WHILE,looping-group}
\end{array} \right\}
\end{align*}
\]

\*biname \*sym

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

FOREVER

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

WHILE,looping-group

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

UNTIL,looping-group

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

The order of the WHILE and UNTIL is not significant.
AND

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

OR

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

looping-group

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

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

looping-branch

One of the special looping instructions specified as:

(BCT, regl)
(BXH, regl, reg2)
(BXLE, regl, 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, (BCT, regl) will loop one time less than the initial value in 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), (INDEX-REG, SIMPLE), IFANY, , ONLY), SPARSE</th>
<th>, , ONLY, SPARSE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>(index, H, W)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>dec dig, sym</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>RX-type</td>
<td></td>
</tr>
</tbody>
</table>

bname

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

index-word

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

(index-reg)

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

(index,W)

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

(index,H)

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

(index, B)

Indicates the DOCASE index is located in the byte at address \textit{index}.

(index, length)

Indicates the DOCASE is to select a CASE on the basis of character strings; the "index" string is at address \textit{index} and of length \textit{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.

\textbf{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, \ldots, \(n\). 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).

\textbf{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.

\textbf{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.

\textbf{ONLY}

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

**DOCASEND**—*Terminate Alternative Selection*

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

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

A-16
**DOEND—Terminate Iteration Block**

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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DOEND</td>
<td>[block-name]</td>
</tr>
</tbody>
</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.

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

else-name

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

BLEND=if-name

Indicates that this ELSE is intended to match the IF block named if-name. If coded, checks will be made to assure that it is the current block.
ESAC—Terminate a DOCASE Alternative

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

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

`block-name` sym

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

ESACOD—Terminate a Selection Among Alternatives

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

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

[block-name]

Indicates the ESACOD is intended to match the outstanding DOCASE block named \textit{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.

\[
\begin{array}{|c|c|}
\hline
label & EXIT & block-name \\
\hline
\end{array}
\]

\textit{label} \quad \textit{sym}

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

\textit{block-name} \quad \textit{sym}

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

*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></th>
<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 -- Insure Structures are Terminated**

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

**IF—Define Conditional Block**

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

| blname     | IF          | {
|            |            | \{ASYNCH\}
|            |            | \{cond-test\}
|            |            | [EXIT= \{exit-block\}]\n|            |            | [ELSE=else-block]\n
blname

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` sym

Indicates the OD is intended to match the outstanding DO block named `block-name`. If coded, checks will be made to assure that it is the current block.
ONEND—Define Normal Loop Termination Code

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

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

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

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

*block-name*  

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

PROC—Define a Proc

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

<table>
<thead>
<tr>
<th>proc-name</th>
<th>PROC</th>
<th>LINKAGE=(OS, CSECT)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>ID=None</td>
</tr>
<tr>
<td></td>
<td></td>
<td>id-string</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SAVE=None</td>
</tr>
<tr>
<td></td>
<td></td>
<td>first, last, NONE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>savearea, length</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BASE=None</td>
</tr>
<tr>
<td></td>
<td></td>
<td>basereg</td>
</tr>
<tr>
<td></td>
<td></td>
<td>baselist</td>
</tr>
<tr>
<td></td>
<td></td>
<td>WORK=None</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.
PROCSAVE=(, ,NONE)

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

SAVE=(, savearea )
sym

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

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

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

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

BASE=NONE

Indicates that no base register loading is to be performed.

BASE=basereg
dec dig, sym

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

BASE=(baselist)
dec dig, sym

A list of base registers may be supplied. Each register in the list will be loaded 4096 bytes beyond the previous and USINGs will be issued for all registers in the list. If the first suboperand of the list is omitted (by coding "BASE=(, reg2, reg 3, . . . regn)") , the default base register will be assumed. (See below.) Only registers 2 through 12 may be specified.

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

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

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

A-32
PROC

For non-OS procs—BASE=NULL

(If the first suboperand of baselist is omitted
for non-OS procs, it defaults to 12.)

WORK=NULL

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

WORK=workreg
dec dig, sym

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

If the WORK-operand is omitted, defaults are:

For OS procs: WORK=2
For Non-OS procs: WORK=1

EXIT=severity
dec dig or "*"

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

DEBUG=options-list

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

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

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

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

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

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

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

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

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

SAVE_TRACE[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 SAVE_TRACE 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., "NOPROC_TRACE" or "NPT"). When the CORP is generated, options will revert to their status before the PROC macro. To avoid the restoring of the options' status at CORP time, include "GLOBAL" (or "GBL") in the list. "ALL" or "NONE" may be specified to turn on or off all options; either may be followed by exceptions. (e.g., "DEBUG=(ALL,NST)" turns on all options except the save-trace.)
PROCEND—Terminate a Proc

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

<table>
<thead>
<tr>
<th>label</th>
<th>PROCEND</th>
<th>proc-name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>[label]</td>
</tr>
<tr>
<td>NONE</td>
<td>RC=</td>
<td>(RC= [{NONE}])</td>
</tr>
<tr>
<td>value</td>
<td></td>
<td>(value)</td>
</tr>
<tr>
<td>reg</td>
<td></td>
<td>(reg)</td>
</tr>
<tr>
<td></td>
<td>RESTORE=</td>
<td>[RESTORE= (first [, last])]</td>
</tr>
<tr>
<td></td>
<td>RETURN=</td>
<td>[RETURN= reglist]</td>
</tr>
<tr>
<td>NONE</td>
<td>LINK=</td>
<td>[LINK= {NONE}]</td>
</tr>
<tr>
<td>linkreg</td>
<td></td>
<td>(linkreg)</td>
</tr>
</tbody>
</table>
OS macro assembly language is an insufficiently powerful language for doing structured programming. As a result, the programming of the STRCMACS was performed in an abstract programming language called SIMPL-M. This is an imaginary language which is a hybrid of SIMPL-X (a high-level structured programming language developed at the University of Maryland), OS macro assembly language, and the STRCMACS themselves. After the code was written in SIMPL-M, it was translated by hand to OS macro assembly language. The SIMPL-M program is considered the "source" code and all updates are performed in it. It is much easier to read than the macro assembly language source. The SIMPL-M source for the macros is listed in Appendix C. In this appendix, we will give a brief description of the SIMPL-M language. In addition, a decision table for the DO macro formats is included in this appendix to complete the source documentation.

Introduction to SIMPL-M

SIMPL-M is a high-level language for the specifying of assembly language macros. In some ways it resembles ALGOL or PL/I; it provides for arbitrary nesting of control structures such as if, while...do, and docase. Two types of modules are allowed: macros and procs. The macros are not macros in the sense that they are expanded when the SIMPL-M source is "translated"; they are macros in the sense that the translated version defines and may be evoked as OS assembly language macros. The operands which are specified for macros closely parallel the allowable operands of OS macro prototype statements (that is, a name field operand and a list of positional and/or keyword operands). The procs are parameterless modules constructed during the stepwise refinement of each of the macros of the STRCMACS. They are expanded in-line in the translation to the assembly language macro definition. Both macros and procs are shown as being evoked by call instructions. The distinction is obvious since the macro calls always have argument lists (possibily 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.)

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

**SIMPL-M:**

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

**OS MACRO:**

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

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

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

- `int` - Integers
- `bit` - Logical variables
- `char` - Character strings

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

- `int` to `bit`: 0 → 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

B-2
bit to char:  false → '0';  true → '1'

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

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

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

```c
int X(20)
```
and are referred to as

```c
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:

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

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

Relations may include implied operands. For example:

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

```c
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:

```c
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:

```
START
  0
  1
  $nnU1 COND-TEST B
  2
  3
  4
  $nnW1 COND-TEST A
  5
  6
  $nnBEG BLOCK'S CODE
  7
  8
  9
  $nnLPB LOOPS BRANCH
  10
  11
  12
  $nnEND END
```
The following decision table shows the connections which must be made for the various formats. Those shown lightly shaded occur without branching (control falls through to the indicated node). Boxes shown cross-hatched do not occur for that operand combination. An example follows the table.
<table>
<thead>
<tr>
<th>FOREVER</th>
<th>✓</th>
</tr>
</thead>
<tbody>
<tr>
<td>WHILE</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>looping-branch</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>AND/OR</td>
<td>A O A A O O A O A O A</td>
</tr>
<tr>
<td>cond-test A</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>AND/OR</td>
<td>A O A O A A O O A O</td>
</tr>
<tr>
<td>UNTIL</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>looping-branch</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>AND/OR</td>
<td>O A O A O A</td>
</tr>
<tr>
<td>cond-test B</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
</tbody>
</table>

| Entry point - 0 | 7 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 7 | 7 | 7 |

| B | 'Branch' leg 2 | Condition | T | F | T | F | F | F | T | T | T | T | T | T | T | T | To node | 12 | 7 | 12 | 12 | 7 | 7 | 12 | 9 | 12 | 9 | 12 | 9 | 12 | 12 |

| 'Fall thru' leg 3 | Condition | F | T | F | T | T | F | F | F | F | F | F | F | F | F | F | To node | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 9 | 7 | 9 | 7 | 9 | 7 | 7 |

| A | 'Branch' leg 5 | Condition | T | T | T | T | T | T | T | T | T | T | T | T | T | T | T | To node | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 |

| 'Fall thru' leg 5 | Condition | T | T | T | T | T | T | T | T | T | T | T | T | T | T | T | To node | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 |

| After Code - 8 | 7 | 4 | 1 | 1 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 1 | 1 | 9 | 9 | 1 | 1 | 1 | 1 | 9 | 1 | 9 | 9 |

| Looping Branch | 'Branch' leg 10 | To node | 4 | 4 | 1 | 4 | 1 | 7 | 7 | 7 | 7 | 7 | 4 | 7 | 4 | 4 | 7 | 7 | 7 | 1 | 7 |

| 'Fall thru' leg 11 | To node | 12 | 4 | 12 | 1 | 4 | 1 | 12 | 12 | 12 | 4 | 12 | 12 | 4 | 4 | 12 | 12 | 12 | 12 | 12 | 12 |

| OPERAND FORMAT | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 |
Example of the code generated for the macro:

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

Operand format number 6.

Code generated:

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

<Block's code>

BCT  5,$18U1
$18END  DS  OH
APPENDIX C

ABSTRACT SOURCE LISTING OF STRCMACS
Global Definitions — 14 July 1973

135. /* GENERAL PURPOSE GLOBALS. */
136.
137. PROC_COUNTER, /* Number for last special proc label "$Pep". The
138. --
139. LAST_BLOCK_NUMBER, /* Number used in labels of most recently generated
140. block. */
141. HEX_IN /* Input value to XHEX macro. */
142.
143. int
144. PROC_COUNTER, /* Number for last special proc label "$Pep." The
145. Ep is PROC_COUNTER in hex. */
146. LAST_BLOCK_NUMBER, /* Number used in labels of most recently generated
147. block. */
148. HEX_IN /* Input value to XHEX macro. */
149. bit
150. ERROccurred, /* General purpose error flag used by various
151. inner macros to report failure. */
152. TD1_FAILED_OCCURS, /* Set true by TERMINATE_DO_LOOP if Do may
153. fall through bottom is terminating (when a looping branch is present); else set false. */
154. NOT_FIRST_PROC, /* False until processing begins on first PROC
155. macro. */
156. SAVETRACE_ON_FIRST_PROC, /* False unless first PROC macro included
157. DEBUG=(--.,SAVETRACE,-) operand. */
158. TRACE_VECTOR_GIVEN /* False until the trace vector (GETRACE)
159. is generated on the first PROC which includes the
160. DEBUG=(**,PROCTRACE,-*) operand. */
161. char
162. BLOCK_LABEL_PREFIX, /* Unique character string for each block
163. for use in generating labels. */
164. EXIT_SEVERITY, /* Mnote severity for EXIT target message. Can be
165. set by PROC macros. */
166. HEX(16), /* Constants used in converting decimal to hex by XHEX. */
167. HEX_OUT, /* Output value from XHEX macro. */
168. PREV_SAVETRACE_AREA, /* Holds label generated on last local PROC
169. save area to be used in producing the static chain for SAVETRACE. */
170. PREV_SAVETRACE_PTR /* Holds label generated as forward pointer
171. within last local PROC save area for static chain for SAVETRACE. */
172.
173. bit
174. DEBUG_FLAGS, /* Causes code and counters to keep execution
175. counts on all blocks. */
176. DEBUG_BlockNAMES_REQD, /* Causes block names to be generated as
177. inline character constants to aid in locating within dumps. */
178. DEBUG_CORPVALUES_REQD, /* Causes register values at end of procs (at
179. CORP macro start) to be saved in inline save areas for reference. */
180. DEBUG_DUMPMACROS_REQD, /* Causes various intermediate values to be
181. printed during macro processing for debugging the macros. */
182. DEBUG_LISTBLOCKS_REQD, /* Causes mnotes to be generated at the start
183. and end of all blocks listing their name, number, and static
184. nesting depth. */
185. DEBUG_MACRONAMES_REQD, /* Causes mnotes to be generated whenever any
186. macro is entered (including inner macros) which list the macro's
187. name; for debugging the macros. */
188. DEBUG_PROCCOUNTS_REQD, /* Causes code and counters to keep execution
189. counts on PROC blocks only. */
190. DEBUG_PROCMACROS_REQD, /* Causes PROC names to be generated as
191. inline character constants to aid in locating within dumps. */
192. DEBUG_PROCTRACE_REQD, /* Causes a trace vector to be generated and
193. code to be generated to keep track of the last 257 PROCs entered. */
194. DEBUG_SAVETRACE_REQD /* Causes all local save areas to be statically
195. chained together and code to be generated to link the chain to the
196. OS save area to provide OS formatting within ABEND dumps. */
Global Definitions -- 14 July 1973

int CURRENT_NEST_LEVEL, /* Current depth of static nesting of
tblocks; stack pointer. */

NESTING_LIMIT, /* Holds dimension of main stack. */

BLOCK_MNBR(100) /* Block number of the Ith block. */

bool END_LABEL_REQD(100), /* Indicates whether Ith block needs an END label
generated during POPOLD_BLOCK. */

EXIT_LABEL_REQD(100) /* Indicates whether Ith block needs an XIT label
generated during POPOLD_BLOCK. */

char BLOCK_NAME(100), /* Block name of Ith block, either USER_NAME specified
in macro label field or generated name "BLKnn" where nnn is the
sequential block number. */

BLOCK_TYPE(100), /* Macro name which generated the Ith block
(IF, DO, DOCASE, CASE, BLOCK, or PROC). */

OPERAND1(100), OPERAND2(100), OPERAND3(100), OPERAND4(100),
/* These hold various data which are needed to close the blocks
generated. Specific contents vary according to the type of
block generated. See individual macros. */

INFORMATION(100) /* Similar to the OPERAND stacks above, the
INFORMATION stack holds information for the closing of the block.
Often the individual characters within the variables are used for
different values, packed together into INFORMATION. */

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

int MAX_CASE_VALUE(9), /* Maximum branch vector value found. */

NEXT_CMP_LABEL_NO(9), /* Case number for next comparison case label
to be generated. */

GCASE_NEST_LEVEL, /* Current depth of stacking in the GCASE stack;
number of nested DOCASES with either GENERAL, SPARSE, or CHARCOMP
operand formats. */

GCASE_NEST_LIMIT /* Maximum depth of nesting of GCASE stack; must
be equal to stack dimension. */

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

/* CONDITIONAL_EXPRESSION_PROCESSOR PSEUDC-PARAMETERS. */

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

bit ULTIMATE_FALLTHRU_CONDITION, /* Logical value upon which conditional
expression is to pass control (or fall through) to the ULTIMATE_
FALLTHRU_LABEL. */

FALLTHRU_LABEL_USED /* CEP sets this true if a branch is generated
to the ULTIMATE_FALLTHRU_LABEL (else no change occurs). */

char ULTIMATE_BRANCH_LABEL, /* Indicates label to be used as branch target
when conditional test does not have logical value stored in
ULTIMATE_FALLTHRU_CONDITION. */

ULTIMATE_FALLTHRU_LABEL, /* Indicates label available as branch target
when conditional test has logical value stored in ULTIMATE_
FALLTHRU_CONDITION. If used as a branch target, FALLTHRU_
LABEL_USED must be set true (by CEP) to insure next sequential
instruction following conditional expression receives label
definition. */

bit UNIQUE_LABEL_ID /* One character unique to this call of CEP used to
insure labels generated by this call will differ from all other
labels, even others within the same macro (particularly for DO). */
IF Macro

11001. BASED IF (USER_NAME, REL=, MASK=, EXIT=, ELSE=)

11002. /* Initiate a block in the structure. Save any information needed
11003. by ELSE or FL. For ASYNCH type, generate branch around block.
11004. for normal IF (EXIT= not specified), generate conditional expression
11005. tests with branch around block (or to ELSE) for false and fall
11006. through for true; if EXIT= specified, then generate branch to
11007. proper block end for true, fall through for false, and delete
11008. IF block from structure. Put USER_NAME on first executable
11009. instruction if one specified. */

11010. bit VALID_EXIT

11011. /* VALID_EXIT is true if EXIT= was specified and no errors have
11012. been found to cause the EXIT to be ignored. */

11013. char EXIT_LABEL,

11014. /* Label for EXIT= branch, when deferred until
11015. after block count has been incremented. */

11016. LABEL /* Outstanding label, waiting to be generated. */

11017. call TRACE_PRINTER (", "IF")

11018. /* Prints macro name "IF" in note if tracing ca. */

11019. call PUSH_NEW_BLOCK (USER_NAME;

11020. BLOCK_TYPE_VALUE='IF',

11021. OPERAND1 VALUE=ELSE,

11022. END LABEL_VALUE=true)

11023. /* Define new block; add to stack. Initialize block specifications.
11024. Assume block will require an END label. Note block type and save
11025. name of ELSE block if one specified here. Set up unique
11026. BLOCK_LABEL_PREFIX for use in generating unique labels. */

11027. if ERROR_OCCURRED

11028. then

11029. mexit

11030. fi

11031. if REL '' or MASK 

11032. then

11033. mnote (8, 'STRC1102 BEL= OR MASK= NOT IN PARENTHESES--IGNORED')

11034. fi

11035. LABEL := USER_NAME

11036. /* Generate USER_NAME at first opportunity. */

11037. VALID_EXIT := (EXIT $$)

11038. /* Set VALID_EXIT to the truth of whether EXIT= was specified. */

11039. if SYSLIST(1,1) = 'ASYNCH'

11040. then /* Either "IF ASYNCH" or "IF (ASYNCH)" was entered. */

11041. call IF_CONDITIONAL_GENERATOR /* Generate branch around block. */

11042. else /* Either "IF ASYNCH" or "IF (ASYNCH)" was entered. */

11043. call IF_CONDITIONAL_GENERATOR /* Generate conditional tests according to IF macro operands and
11044. the current conditional test specs. */

11045. fi

11046. call IF_CONDITIONAL_GENERATOR /* Generate conditional tests according to IF macro operands and
11047. the current conditional test specs. */

11048. if LABEL 

11049. then

11050. call IF_CONDITIONAL_GENERATOR /* Generate conditional tests according to IF macro operands and
11051. the current conditional test specs. */

11052. fi

11053. call IF_CONDITIONAL_GENERATOR /* Generate conditional tests according to IF macro operands and
11054. the current conditional test specs. */

11055. fi

11056. call IF_CONDITIONAL_GENERATOR /* Generate conditional tests according to IF macro operands and
11057. the current conditional test specs. */

11058. call IF_CONDITIONAL_GENERATOR /* Generate conditional tests according to IF macro operands and
11059. the current conditional test specs. */

11060. call IF_CONDITIONAL_GENERATOR /* Generate conditional tests according to IF macro operands and
11061. the current conditional test specs. */

11062. call IF_CONDITIONAL_GENERATOR /* Generate conditional tests according to IF macro operands and
11063. the current conditional test specs. */

11064. if LABEL 

11065. then

11066. call IF_CONDITIONAL_GENERATOR /* Generate conditional tests according to IF macro operands and
11067. the current conditional test specs. */

11068. fi

11069. if VALID_EXIT

11070. then

11071. /* No IF block remains after completion of "IF EXIT=..." macro;
11072. simulate presence of FI macro. */

11073. call FI ( ; )

11074. run
11076. **blk IF_ASYNCH_BRANCH**
11077. /* Give error message if EXIT specified. Generate branch to
11078. end of IF block. */
11079. if VALID_EXIT
11080. then
11081. note (8, 'STRC1101 EXIT= IGNORED WITH "ASYNCH"')
11082. VALID_EXIT := FALSE
11083. fi
11084. generate (LABEL || ' B ' || BLOCK_LABEL_PREFIX || 'END')
11085. /* Branch around asynchroneous IF block. */
11086. LABEL := ""
"If" Macro -- 21 June 1973

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

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

C-6
**IF** Macro

**21 June 1973**

```plaintext
11116. proc IF_EXIT_SPECS
11117. /* An EXIT= operand has been specified; insure ELSE= not also
11118. specified. If valid, change conditional test specs to standard
11119. for EXIT-type IF including the assigning of the branch target
11120. as the XIT label of the block specified by the EXIT keyword. */
11121. char HOLD /* Temporary. */
11122. if ELSE # /* If ELSE= was not omitted... */
11123. then
11124. note (@, "STRC1103 EXIT= IGNORED WITH ELSE")
11125. VALID_EXIT := false
11126. else
11127. HOLD := ULTIMATE_BRANCH_LABEL
11128. /* Save old branch label, we may need it yet. */
11129. call EXIT_FIND ( ; EXIT)
11130. /* Sets ULTIMATE_BRANCH_LABEL to XIT label of block whose name
11131. is specified in the argument; if none specified ("EXIT=*"),
11132. use block surrounding IF macro; if no such block, issue message,
11133. change ULTIMATE_BRANCH_LABEL unmodified, and set ERROR_OCCURRED to
11134. true. Mark target block as requiring XIT label. */
11135. if DEBUG_BLOCKCOUNTS_REQD
11136. then
11137. EXIT_LABEL := ULTIMATE_BRANCH_LABEL
11138. ULTIMATE_BRANCH_LABEL := HOLD
11139. /* Make EXIT-type IF act like regular IF (i.e., fall through on true)
11140. so we can count the number of times the exit is taken; save the
11141. EXIT_LABEL for a branch after the count is made and make the
11142. ULTIMATE_BRANCH_LABEL whatever it would have been had this been
11143. a regular IF. */
11144. else
11145. ULTIMATE_FALLTHRU_LABEL := BLOCK_LABEL_PREFIX
11146. ULTIMATE_FALLTHRU_CONDITION := false /* Fall through on false. */
11147. END_LABEL_REQD(CURRENT_NEST_LEVEL) := false
11148. if ERROR_OCCURRED /* on EXIT_FIND... */
11149. then
11150. /* Exit point not found and message has been issued. Make branch
11151. point same as fall-through point and clear error (i.e., fix up
11152. and continue). */
11153. ULTIMATE_BRANCH_LABEL := BLOCK_LABEL_PREFIX
11154. FALLTHRU_LABEL_USED := TRUE
11155. /* ...since it's also the branch label. */
11156. ERROR_OCCURRED := false
11157. fi
11158. fi
11159. fi
11160. goto C-7
```
IF Macro — 21 June 1973

11163. PROC IF_CONDITIONAL_GENERATOR
11164. /* Generate code to pass control to the ULTIMATE_FALLTHRU_LABEL (or
11165. logical value which is stored in ULTIMATE_FALLTHRU_CONDITION;
11166. else to pass control to the ULTIMATE_BRANCH_LABEL. Also generate
11167. fall-through label definition if FALLTHRU_LABEL_USED was ever
11168. turned on. */
11169. */
11170. /* Set up further specifications required by CONDITIONAL_EXPRESSION_  
11171. PROCESSOR. */
11172. FIRST_INDEX := 1  
11173. LAST_INDEX := N'SYSLIST
11174. /* The operands of SYSLIST to be processed are operand 1 through the
11175. last operand [SYSLIST(N'SYSLIST)], i.e., all of them. */
11176. UNIQUE_LABEL_ID := 'I'
11177. /* Used by CONDITIONAL_EXPRESSION_PROCESSOR to produce unique
11178. labels. */
11179. CALL CONDITIONAL_EXPRESSION_PROCESSOR (LABEL; SYSLIST)
11180. /* Generate code corresponding to the operands of the IF (referred
11181. to collectively as SYSLIST). Only the SYSLIST can be passed
11182. directly as arguments; the following variables are effectively
11183. arguments but are passed in global variables:
11184. arguments but are passed in global variables:
11185. FIRST_INDEX,
11186. LAST_INDEX,
11187. ULTIMATE_BRANCH_LABEL,
11188. ULTIMATE_FALLTHRU_LABEL,
11189. ULTIMATE_FALLTHRU_CONDITION,
11190. UNIQUE_LABEL_ID,
11191. FALLTHRU_LABEL_USED.
11192. Process operands of the SYSLIST beginning with SYSLIST(FIRST_INDEX)
11193. through SYSLIST(LAST_INDEX) [for the IF macro, this is the entire
11194. SYSLIST], generating the indicated tests to pass control as
11195. indicated above. The UNIQUE_LABEL_ID is used to insure unique
11196. labels. If a branch is made to the ULTIMATE_FALLTHRU_LABEL, then
11197. FALLTHRU_LABEL_USED is set, else it is unaltered. */
11198. IF FALLTHRU_LABEL_USED
11199. THEN
11200. LABEL := ULTIMATE_FALLTHRU_LABEL
11201. ELSE
11202. LABEL := **
11203. END
11204. END
DECLARE IF_BLOCK_COUNT.

/* If debugging in progress, generate block name and/or count of block execution. Note that ASYNCH blocks cannot be counted. */

char TARGET /* Temporary. */

if SYSLIST(1,1) = 'ASYNCH'

then

if DEBUG_BLOCKNAMES_REQD

then

generate ("C'1' BLOCK_NAME(CURRENT_NEST_LEVEL) )

endif

endif

else /* Not ASYNCH. */

if DEBUG_BLOCKCOUNTS_REQD or DEBUG_BLOCKNAMES_REQD

then

if DEBUG_BLOCKNAMES_REQD

then

generate (LABEL II '*' BLOCK_LABELPREFIX |II 'IFC')

endif

endif

endif

endif

endif

endif

if EXIT_LABEL

then

TARGET := BLOCK_LABELPREFIX |II 'GO'

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

if DEBUG_BLOCKNAMES_REQD

then

generate ("C'1' BLOCK_NAME(CURRENT_NEST_LEVEL) )

endif

endif

endif

if DEBUG_BLOCKCOUNTS_REQD

then

generate (BLOCK_LABELPREFIX |II 'IFC')

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif

endif
"ELSE" Macro -- 26 June 1973

13001. **B**EGIN**E**lse (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 /* XIT 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 then
13013. mexit fi
13014. Call VERIFY_END( ; 'IF', BLEND) /* Verifies current block has the name specified
13015. by the BLEND= operand on the ELSE macro (if any) and that it is an IF block.
13016. Various errors receive messages and either intermediate blocks are
13017. BLENDed as a fixup or ERROR_OCCURRED is set. */
13018. if ERROR_OCCURRED then
13019. mexit fi
13020. if OPERAND2(CURRBENT_NEST_ LEVEL) = 'ELSE'
13021. then
13022. nnote (8, 'STRC1302 ELSE HAS ALREADY BEEN GENERATED FOR CURRENT IF')
13023. mexit fi
13024. ELSE BLOCK NO := LAST BLOCK NUMBER + 1 /* Get block number of upcoming ELSE block. */
13025. Generate (ELSEBLOCKNO || 'END') /* Generate branch to end of ELSE block. */
13026. if EXIT_LABEL_REQD(CURRENT_NESTLEVEL) then
13027. IF_EXIT_LABEL := $ | BLOCK_NUMBER(CURRENT_NEST_LEVEL) || 'XIT'
13028. EXIT_LABEL_REQD(CURRENT_NEST_LEVEL) := false /* Exit label has been postponed to FI. */
13029. fi
13030. Call POP OLD BLOCK ( ; ) /* Remove IF block from stack. */
13031. Call PUSH NEW BLOCK(USER_NAME;
13032. BLOCK_TYPE_VALUE='IF', /* Must be marked as "IF" block for FI's
13033. call to VERIFY_END. */
13034. END_LABEL_VALUE=TRUE,
13035. OPERAND2_VALUE='ELSE',
13036. OPERAND1_VALUE=IF_EXIT_LABEL)
13037. if OPERAND1(CURRENT_NEST_LEVEL) # # USER_NAME
13038. then
13039. Note (6, 'STRC1301 ELSE='
13040. OPERAND1(CURRENT_NEST_LEVEL)
13041. ' SPECIFIED ON IF BLOCK' || BLOCK_NAME(CURRENT_NEST_LEVEL)) /* ELSE=elsename specified on IF macro but different (or no) label
13042. field on ELSE macro. */
13043. Note (6, 'STRC1301 ELSE='
13044. OPERAND2(CURRENT_NEST_LEVEL))
13045..EXIT_LABEL_REQD(CURRENT_NEST_LEVEL) := false
13046. fi
13047. Note (8, 'STRC1301 ELSE HAS ALREADY BEEN GENERATED FOR CURRENT IF')
13048. mexit fi
13049. if OPERAND2(CURRENT_NEST_LEVEL) = 'ELSE'
13050. then
13051. Note (6, 'STRC1302 ELSE HAS ALREADY BEEN GENERATED FOR CURRENT IF')
13052. mexit fi
13053. if OPERAND1(CURRENT_NEST_LEVEL) # # USER_NAME
13054. then
13055. Generate (ELSEBLOCKNO || 'END') /* Generate branch to end of ELSE block. */
13056. if EXIT_LABEL_REQD(CURRENT_NESTLEVEL) then
13057. IF_EXIT_LABEL := $ | BLOCK_NUMBER(CURRENT_NEST_LEVEL) || 'XIT'
13058. EXIT_LABEL_REQD(CURRENT_NEST_LEVEL) := false /* Exit label has been postponed to FI. */
13059. fi
13060. Call POP OLD BLOCK ( ; ) /* Remove IF block from stack. */
13061. Call PUSH NEW BLOCK(USER_NAME;
13062. BLOCK_TYPE_VALUE='IF', /* Must be marked as "IF" block for FI's
13063. call to VERIFY_END. */
13064. END_LABEL_VALUE=TRUE,
13065. OPERAND2_VALUE='ELSE',
13066. OPERAND1_VALUE=IF_EXIT_LABEL)
13067. if OPERAND1(CURRENT_NEST_LEVEL) # # USER_NAME
13068. then
13069. Generate (USER_NAME || 'DS OR') /* Define outstanding label. */
13070. fi
13071. 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" is an if tracing on. */
15007. `if` CURRENT_NEST_LEVEL > NESTING_LIMIT
15008. `then`
15009. `call` POP_OLD_BLOCK ( ; )
15010. `exit`
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`
15025. `if` OPERAND1(CURRENT_NEST_LEVEL) != ''
15026. `then`
15027. `adpt` (8, 'STRC1501 ELSE BLOCK " ' OPERAND1(CURRENT_NEST_LEVEL)
15028. " NOT FOUND')
15029. `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

```
21001.  MACRO DO (USER_NAME; REL=, NASK=)
21002.  /* Create a block in the structure. Save any information needed by
21003.  OD. Generate conditional tests to loop according to UNTIL and WHILE
21004.  conditional expressions. Insure UNTIL tests are not checked on first
21005.  execution. Generate USER_NAME, if any, on first executable
21006.  instruction. Order of generated code follows. (Each section may be
21007.  omitted in certain cases; see decision table in documentation for
21008.  details.)
21009.  */
21010.  FLOW POINT 0: Branch to entry point for total block.
21011.  FLOW POINT 1: Start of UNTIL tests, labeled $nnnUl.
21012.  FLOW POINT 2: Branch on success of UNTIL tests.
21013.  FLOW POINT 3: Fall-through on failure of UNTIL tests.
21014.  FLOW POINT 4: Start of WHILE tests, labeled $nnnW1.
21015.  FLOW POINT 5: Branch on failure of WHILE tests.
21016.  FLOW POINT 6: Fall-through on success of WHILE tests.
21017.  FLOW POINT 7: Start of internal looping code (user code between DO
21018.  and termination of DO loop by OD, ATEND, or
21019.  TERMINATEDOLOOP. */
```

```
21030.  int
21031.  WHILE_INDEX, /* Index within SYSLIST of start of WHILE operands;
21032.  initially points to WHILE keyword but eventually points to start of
21033.  conditional test. */
21034.  WHILE_END_INDEX, /* Index within SYSLIST of end of WHILE operands. */
21035.  UNTIL_INDEX, UNTIL_END_INDEX, /* Same as WHILE counterparts. */
21036.  OPERAND_FORMAT, /* Column number of DC decision table (see documenta-
21037.  tion) which indicates what operands are present. */
21038.  I, /* Temporary work variable. */
21039.  LB, /* Index within SYSLIST of the looping branch (BCT, BXH, or
21040.  BILE). */
21041.  LASTOP /* Index of last positional operand of DO macro which is
21042.  considered valid. */
21043.  int
21044.  WHILE_PRESENT, UNTIL_PRESENT,
21045.  /* Indicates a looping group (a looping branch and/or a
21046.  conditional test) of the indicated type is present. */
21047.  LS_LABEL_REQ,
21048.  /* Indicates whether the looping branch requires a label. */
21049.  WHILE_COND_TEST, UNTIL_COND_TEST,
21050.  /* Indicates a conditional test of the given type is present. */
21051.  THIS_CONDITIONAL_REQ,
21052.  /* Indicates the currently processed type (WHILE or UNTIL) includes
21053.  a conditional test. */
21054.  char
21055.  LB_OPCODE_ID,
21056.  /* OpCode of looping branch: BCT, BXH, or BILE. */
21057.  LB_OPCODE1, LB_OPCODE2,
21058.  /* Operands of the looping branch, if present. */
21059.  LB_LOGIC_OP,
21060.  /* Logical operator ("AND" or "OR") which connects the looping branch
21061.  and the conditional test, if both are present. */
21062.  LOOPING_BRANCH_TYPE,
21063.  /* "NONE", "WHILE", or "UNTIL": indicates position of looping
21064.  branch. */
21065.  LABEL,
21066.  /* Any outstanding label waiting to be generated. */
21067.  FIRST_ID,
21068.  /* Label required at start of conditional test being processed. */
21069.  MAIN_OP,
21070.  /* Logical operator ("AND" or "OR") which connects WHILE and UNTIL
21071.  looping groups, if both are present. */
```

C-12
call TRACE_PRINTER ( ; 'DO')
	/* Prints macro name 'DO' in mnote if tracing on */
call PUSH_NEW BLOCK (USER_NAME; BLOCK_TYPE_VALUE='EO')
	/* Define new block; add to stack. Initialize block specifications. */
if ERROR_OCCURRED
then
exit
fi
if REL = ' ' or MASK = ' '
then
 note (b. 'STRC2113 REL= OR MASK NOT IN PARENTHESES-IGNORED')
fi
LABEL := USER_NAME
call DO SCAN OPERANDS
	/* Collect scanning information and looping branch (BCT, BXH, and BXLE) information from operands. Set OPERAND_FORMAT based on these values. */
if OPERAND_FORMAT = 0 and # 10 and # 12 and # 19
then
END_LABEL_REQD(CURRENT_NEST_LEVEL) := true
fi
if DEBUG BLOCKCOUNTS_REQD
then
/* Generate reset of current loop count. */
qenerate (LABEL || ' SR 1,1')
LABEL := ' ' || BLOCK_LABEL_PREFIX || 'DOL')
fi
if OPERAND_FORMAT = 0
then
/* Not infinite loop. */
call DO_BRANCH_FOR LOOP ENTRY
	/* Generate flow point 0 branch, if required, to proper label to insure UNTIL tests are not made before first loop. */
call DO GENERATE_ALL_CONDITIONAL_TESTS
	/* Cause WHILE and UNTIL tests to be generated (flow points 1 through 6) with proper labels. */
fi
call DO_LABEL BLOCK
	/* Store begin label (flow point 7) into LABEL. */
call DO_INFO_SAVE
	/* Insert into stack all information required by TERMINATE_DO_LOOP to close loop (flow points 8 through 12). */
call DO_TRACE_COUNTERS
	/* Generate any debugging counters, etc. */
if LABEL = ''
then
generate (LABEL || ' DS 0H')
fi
mend
"DO" Macro -- 21 June 1973

21127. **PROC** DO_SCAN_OPERANDS
21128. /* 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_END_INDEX, LB_OPCODE_ID, LB_OPERAND1, LB_OPERAND2, LB_LABEL_ADDR, and LB_LOGIC_OP) which must be passed to TERMINATE_DO_LOOP to close loop; and set OPERAND_FORMAT (case number code from decision table). */

21137. **CALL** DO_FIND_KEYWORDS_AND_PRESENCE
21138. /* Put operand index of "WHILE" and "UNTIL" keywords into 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. */

21142. **CALL** DO_FIND_END_INDEXES_AND_MAIN_OP
21143. /* For each type xxxx (WHILE and UNTIL) which is present, put index of the last operand of looping group for that type into xxxx_END_INDEX; if both present, find logic operator which connects them and put it into MAIN_OP, else put in null string. */

21147. **CALL** DO_LOOPING_BRANCH_AND_FIRST_OPERAND
21148. /* Collect looping branch information and step WHILE_INDEX and UNTIL_INDEX to first operand of conditional test (not including looping branch and following operator) or set to zero if not present. Also set WHILE and UNTIL_COND_TEST to indicate presence of conditional tests. */

21153. **CALL** DO_SET_FORMAT
21154. /* Set type of operands according to decision table, using WHILE_PRESENT, UNTIL_PRESENT, MAIN_OP, WHILE_COND_TEST, UNTIL_COND_TEST, LOOPING_BRANCH_TYPE, and LB_LOGIC_OP to make decision. */

21156. **END**
DO FIND KEYWORDS AND PRESENCE

21160. DEFN DO_FIND_KEYNORDS AND PRESENCE

21161. /* Find operand index of 'WHILE' and 'UNTIL' keywords; set to zero

21162. if omitted. Put index of last valid operand in LASTOP.

21163. Set WHILE_PRESENT and UNTIL_PRESENT true if corresponding looping

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

21166. WHILE_INDEX, UNTIL_INDEX := 0

21167. if SYSLIST(1) = 'WHILE' and 'UNTIL' and

21168. then

21169. Rnote (8, 'STRC2108 FIRST OPERAND MUST BE "WHILE", "UNTIL", "FOREVER", OR OMITTED')

21170. fi

21171. /*

21172. Assume they're all valid. */

21173. I := 1

21174. LASTOP := N'SYSLIST

21175. while I ≤ LASTOP

21176. do /* Search for WHILE and UNTIL keywords. */

21177. if SYSLIST(I) = 'WHILE'

21178. then

21179. if WHILE_INDEX = 0

21180. then /* No WHILE found before. */

21181. WHILE_INDEX := I

21182. else

21183. Rnote (8, 'STRC2101 OPERANDS AFTER SECOND "WHILE" IGNORED')

21184. LASTOP := I - 1

21185. fi

21186. else /* Operand was not "WHILE". */

21187. if SYSLIST(I) = 'UNTIL'

21188. then

21189. if UNTIL_INDEX = 0

21190. then /* No UNTIL found before. */

21191. UNTIL_INDEX := I

21192. else

21193. Rnote (8, 'STRC2102 OPERANDS AFTER SECOND "UNTIL" IGNORED')

21194. LASTOP := I - 1

21195. fi

21196. fi

21197. fi

21198. I := I + 1

21199. do /* (Termination: I is incremented and N'SYSLIST is fixed during

21200. loop: LASTOP ≤ N'SYSLIST. I would eventually exceed N'SYSLIST

21201. so it must eventually exceed LASTOP.) */

21202. if WHILE_INDEX > 1 AND UNTIL_INDEX > 0

21203. then /* Garbage operands are present. */

21204. Rnote (8, 'STRC2114 SUPERFLUOUS LOOPING GROUP IGNORED')

21205. fi

21206. fi

21207. /* Decide whether WHILE and UNTIL looping groups are present. The

21208. possible operand formats are:

21209. DO UNTIL, (looping-group)

21210. DO WHILE, (looping-group)

21211. DO WHILE, (looping-group), (and/or), UNTIL, (looping-group)

21212. DO WHILE, (looping-group), (and/or), WHILE, (looping-group)

21213. DO [No operand or single operand "FOREVER"] /*

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

21222. DOFIND_END_INDEXES_AND_MAIN_OP
21223. /* For each type xxxxx (WHILE and UNTIL), put index of last operand of
21224. looping groups for that type into xxxxx_END_INDEX; if both are
21225. present, find logic operator which connects them and put it into
21226. MAIN_OP; else MAIN_OP := ''; WHILE_INDEX and UNTIL_INDEX currently
21227. point to the corresponding keyword or are zero if the corresponding
21228. keyword is omitted or implied (due to error). */

21229. MAIN_OP := '';
21230. WHILE_INDEX, UNTIL_INDEX := LASTOP /* As initial guess. */
21231. if LASTOP = 1 and SYSLIST(I) = 'FOREVER'
21232. then WHILE_PRESENT, UNTIL_PRESENT := false
21233. else if WHILE_PRESENT
21234. then all UNTIL_PRESENT
21235. fi
21236. if WHILE_INDEX = WHILE_END_INDEX
21237. then /* One of the following was entered:
21238. "DO WHILE,UNTIL,-*" or
21239. "DO AND,UNTIL,--" */
21240. mnote (8, 'STRC2109 WHILE TEST IS VOID-IGNORED')
21241. WHILE_PRESENT := false
21242. fi
21243. if UNTIL_INDEX = UNTIL_END_INDEX
21244. then /* One of the following was entered:
21245. "DO UNTIL,WHILE,-*" or "DO UNTIL,AND,WHILE,-,.*"
21246. mnote (8, 'STRC2111 UNTIL TEST IS VOID-IGNORED')
21247. UNTIL_PRESENT := false
21248. fi
21249. else if WHILE_INDEX = WHILE_END_INDEX
21250. then /* One of the following was entered:
21251. "DO WHILE,UNTIL,-*" or "DO WHILE,AND,UNTIL,-,.*"
21252. mnote (8, 'STRC2112 WHILE TEST IS VOID-IGNORED')
21253. WHILE_PRESENT := false
21254. fi
21255. else
21256. if WHILE_INDEX = WHILE_END_INDEX
21257. then /* One of the following was entered:
21258. "DO UNTIL,WHILE,-*" or "DO AND,UNTIL,--" */
21259. mnote (8, 'STRC2113 WHILE TEST IS VOID-IGNORED')
21260. WHILE_PRESENT := false
21261. fi
21262. else if UNTIL_INDEX = UNTIL_END_INDEX
21263. then /* One of the following was entered:
21264. "DO UNTIL,WHILE,-*" or "DO UNTIL,AND,WHILE,-,.*"
21265. mnote (8, 'STRC2114 UNTIL TEST IS VOID-IGNORED')
21266. UNTIL_PRESENT := false
21267. fi
21268. else /* WHILE_PRESENT but not UNTIL_PRESENT. */
21269. if WHILE_INDEX = WHILE_END_INDEX /* Which is equal to LASTOP. */
21270. then /* "DO WHILE" with no other operands. */
21271. WHILE_PRESENT := false /* Ignore to get infinite DO. */
21272. fi
21273. else
21274. if UNTIL_INDEX = UNTIL_END_INDEX /* "DO UNTIL" with no other operands. */
21275. then /* Ignore to get infinite DO. */
21276. fi
21277. fi
21278. else /* WHILE_PRESENT and UNTIL_PRESENT */
21279. fi
21280. if WHILE_PRESENT and UNTIL_PRESENT
21281. then /* "DO UNTIL" with no other operands. */
21282. fi
21283. if WHILE_PRESENT and UNTIL_PRESENT
21284. then /* "DO WHILE" with no other operands. */
21285. fi
21286. else
21287. MAIN_OP := 'AND' /* Assumed. */
21288. if SYSLIST(I) = 'OR'
21289. then
21290. MAIN_OP := 'OR'
21291. else
21292. if SYSLIST(I) = 'AND'
21293. then
21294. mnote (8, 'STRC2115 LOGIC OPERATOR BETWEEN "WHILE" AND "UNTIL"')
21295. fi
21296. fi
21297. fi
21298. else
21299. fi
21300. core
DO Macro -- 21 June 1973

```
21302. PROC DO_LOOPING_BRANCH_AND_FIRST_OPERAND
21303. /* Step WHILE_INDEX and UNTIL_INDEX to first operand of conditional
21304. test or set to zero if not present. Collect all looping branch
21305. information. Set WHILE_ and UNTILCOND_TEST to true if appropriate
21306. conditional test is present (as opposed to only a looping branch). */
21307. */ Assume no looping branch. */
21308. LB := 0
21309. LOOPING_BRANCH_TYPE := 'NCNE'
21310. if WHILE_PRESENT
21311. then
21312. I, UNTIL_INDEX := UNTIL_INDEX + 1
21313. /* Move UNTIL_INDEX from pointing at "UNTIL" to pointing at first
21314. operand. */
21315. UNTIL_operand.*/
21316. if SYSLIST(I,1) = 'BCT' or = 'BXLE' or = 'BXH'
21317. then
21318. LOOPING_BRANCH_TYPE := 'UNTIL'
21319. UNTIL_CONDTEST := (UNTIL_END_INDEX > I)
21320. LB := I /* UNTIL_INDEX is still pointing at the looping branch; we aren't
21321. sure how far to advance it yet. */
21322. else
21323. UNTIL_COND_TEST := true
21324. fi
21325. else
21326. UNTIL_INDEX, UNTIL_END_INDEX := 0
21327. UNTIL_COND_TEST := false
21328. /* Turn off all UNTIL stuff. */
21329. fi
21330. if WHILE_PRESENT
21331. then
21332. I, WHILE_INDEX := WHILE_INDEX + 1
21333. /* Move WHILE_INDEX from pointing at "WHILE" to pointing at first
21334. operand. */
21335. WHILE_operand.*/
21336. if SYSLIST(I,1) = 'BCT' or = 'BXLE' or = 'BXH'
21337. then
21338. if LOOPING_BRANCH_TYPE = 'NONE'
21339. then /* There is no WHILE looping branch. */
21340. LOOPING_BRANCH_TYPE := 'WHILE'
21341. while_CONDTEST := (WHILE_END_INDEX > I)
21342. LB := I /* WHILE_INDEX is still pointing at looping branch; we aren't sure
21343. how far to advance it yet. */
21344. if SYSLIST(I,1) = 'BCT'
21345. then
21346. END warning (4, 'STRC2103 WARNING—"WHILE,(BCT,..." WILL LOOP ONE
21347. LESS TIME THAN VALUE IN REGISTER')
21348. if MAIN_OP = 'OR'
21349. then
21350. END warning (4, 'STRC2104 WARNING—LOOPING BRANCH MAY NOT BE
21351. EXECUTED ON EVERY ITERATION')
21352. else /* There is also an UNTIL looping branch. */
21353. END warning (4, 'STRC2105 TWO LOOPING BRANCHES INVALID IN "DO"—
21354. "WHILE" IGNORED')
21355. WHILE_PRESENT, WHILE_COND_TEST := false
21356. MAIN_OP := '1'
21357. WHILE_INDEX, WHILE_END_INDEX := 0
21358. fi
21359. else
21360. WHILE_COND_TEST := true
21361. fi
21362. fi
21363. WHILE_INDEX, WHILE_END_INDEX := 0
21364. WHILE_COND_TEST := false
21365. call DO LOOPING_BRANCH_PROCESS
21366. /* Collect looping branch information and advance WHILE_ or
21367. UNTIL_INDEX over looping branch operands. */
```

C-IV
DO Macro -- 21 June 1973

21374. PROC DO_LOOPING_BRANCH_PROCESS
21375. /* Collect looping branch information:
21376.   LB_OPCODE_ID 'BCT', 'BLE' or 'BGE'
21377.   LB_OPCODE2 Second operand, null (or garbage) for BCT
21378.   LB_LOGIC_OP Logic operand connecting looping branch to
21379.       rest of WHILE or UNTIL.
21380.   LB_LABEL_REQ Indicates whether looping branch will need
21381.     a label.
21382. Also step WHILE or UNTIL_INDEX over looping branch. If any looping
21383. branch is present, LB contains its index; else, LB = 0. */
21385. int OP_COUNT /* Number of operands looping branch needs. */
21387. LB_OPCODE_ID, LB_OPERAND1, LB_OPERAND2, LB_LOGIC_OP := '
21388.   /* Assume no looping branch is present. */
21389. if LB ≠ 0
21390. then
21391. LB_OPCODE_ID := SYSLIST(LB,1)
21392. if LB_OPCODE_ID ≠ 'BCT'
21393. then
21394.   OP_COUNT := 2
21395. else
21396.   OP_COUNT := 3
21397. fi
21398. if LB_OPCODE_ID ≠ LB_OPCODE_ID
21399. then /* Not a sublist */
21400.   if SYSLIST(LB) ≠ 1
21401.   else /* Given as a sublist. */
21402.     LE_OPCODE1 := SYSLIST(LB,2)
21403.     LE_OPCODE2 := SYSLIST(LB,3)
21404.     if SYSLIST(LB) ≠ OP_COUNT
21405.     then
21406.       /* SYSLIB(LB) INVALID NUMBER OF OPERANDS FOR */
21407.       LB_OPCODE_ID
21408.     fi
21409.     fi
21410.     LB := LB + 1 /* Step LB past logical operator which connects
21411.     branching to conditional test, if both are present. */
21412.     if (LOOPING_BRANCH_TYPE = 'WHILE' and WHILE_COND_TEST)
21413.     or (LOOPING_BRANCH_TYPE = 'UNTIL' and UNTIL_COND_TEST)
21414.     then
21415.     LB_LOGIC_OP := SYSLIST(LB)
21416.     if LB_LOGIC_OP ≠ 'AND' or ≠ 'OR'
21417.     then
21418.       LB := LB + 1 /* Step LB past logic operator. */
21419.     else
21420.       /* SYSLIB(LB) INVALID AFTER LOOPING BRANCH—"AND" INSERTED */
21421.     LB_LOGIC_OP := 'AND'
21422.     fi
21423.     if LOOPING_BRANCH_TYPE = 'WHILE'
21424.     then
21425.       WHILE_INDEX := LB
21426.     else
21427.       UNTIL_INDEX := LB
21428.     fi
21429.     /* Set xxxx_INDEX to point at start of conditional test, if any. */
21430.     fi
21431.     fi
21432.     LB_LABEL_REQ := (LOOPING_BRANCH_TYPE = 'WHILE')
21434. ENDP
DO Macro -- 21 June 1973

21436. PROC DO_SET_FORMAT
21437. /* Set OPERAND_FORMAT according to decision table (see documenta-
21438. tion). */
21439.
21440. if WHILE_PRESENT
21441. then
21442. if LOOPING_BRANCH_TYPE = 'WHILE'
21443. then
21444. if WHILE_COND_TEST
21445. then
21446. if LB.Logic_OP = 'AND'
21447. then
21448. if UNTIL_PRESENT
21449. then
21450. if MAIN_OP = 'AND'
21451. then
21452. OPERAND_FORMAT := 15
21453. else
21454. OPERAND_FORMAT := 16
21455. fi
21456. else
21457. OPERAND_FORMAT := 13
21458. fi
21459. else
21460. if UNTIL_PRESENT
21461. then
21462. if MAIN_OP = 'AND'
21463. then
21464. OPERAND_FORMAT := 17
21465. else
21466. OPERAND_FORMAT := 18
21467. fi
21468. else
21469. OPERAND_FORMAT := 14
21470. fi
21471. else
21472. if UNTIL_PRESENT
21473. then
21474. if MAIN_OP = 'AND'
21475. then
21476. OPERAND_FORMAT := 11
21477. else
21478. OPERAND_FORMAT := 12
21479. fi
21480. else
21481. OPERAND_FORMAT := 10
21482. fi
21483. fi
21484. fi
/* if WHILE_PRESENT then
   if LOOPING_BRANCH_TYPE = 'WHILE' then *** */

else
   if LOOPING_BRANCH_TYPE = 'UNTIL'
then UNTIL_COND_TEST
   if LD_LOGIC_OP = 'AND'
    then if MAIN_OP = 'AND'
        then OPERAND_FORMAT := 7
        else OPERAND_FORMAT := 9
    else OPERAND_FORMAT := 8
   fi
else
   if MAIN_OP = 'AND'
    then OPERAND_FORMAT := 2
    else OPERAND_FORMAT := 5
fi
else
   if UNTIL_PRESENT
    then if MAIN_OP = 'AND'
        then OPERAND_FORMAT := 3
        else OPERAND_FORMAT := 1
    else OPERAND_FORMAT := 1
fi
fi
21533. */ /* WHILE_PRESENT then */
21535. else
21536. if UNTIL_PRESENT
21537. then
21538. if LOOPING_BRANCH_TYPE = 'UNTIL'
21539. then
21540. if UNTIL_COND_TEST
21541. then
21542. if LB_LOGIC_OP = 'AND'
21543. then
21544. OPERAND_FORMAT := 22
21545. else
21546. OPERAND_FORMAT := 21
21547. fi
21548. else
21549. OPERAND_FORMAT := 19
21550. fi
21551. else
21552. OPERAND_FORMAT := 20
21553. fi
21554. else
21555. OPERAND_FORMAT := 0
21556. fi
21557. fi
21558. Corp
"DO" Macro -- 21 June 1973

21560. PROC DO_BRANCH_FOR_LOOP_ENTRY
21561. /* Generate branch at flow point 0, if required, to proper label to
21562. ensure UNTIL tests are not made before first loop. */
21563.
21564. "DO"
21565. OPERAND_FORMAT IF
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.
21571. CASE (10-18)
21572. /* Branch to WHILE looping branch first. */
21573. GENERATE (LABEL || 'B' || BLOCK_LABEL_PREFIX || 'LPB')
21574. LABEL := '
21575.
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.
21581. CASE
21582. CODE

C-22
DO Macro -- 21 June 1973

21564. PROC DO_GENERATE_ALL_CONDITIONAL_TESTS
21565.      /* Cause WHILE and UNTIL conditional tests to be generated with proper
21566.      labels. */
21567.      INT PASS /* Looping index. */
21568.      PASS := 1
21569.      WHILE PASS <= 2
21570.      do
21571.      if PASS = 1
21572.      then
21573.      CALL DO_UNTIL_PREPROCESS
21574.      GOTO
21575.      CALL DO_WHILE_PREPROCESS
21576.      fi
21577.      /* The DO_xxxCONDITIONAL_PROC proc must set:
21578.      TESTCONDITIONAL_REQD from xxxxCOND_TEST
21579.      FIRST_INDEX from xxxx_INDEX
21580.      LAST_INDEX from xxxx_LAST_INDEX
21581.      UNIQUE_LABEL_ID with the first letter of xxxx
21582.      ULTIMATE_BRANCH_LABEL with the branch target
21583.      ULTIMATE_FALLTHRU_LABEL with the fallthru name
21584.      ULTIMATE_FALLTHRU_CONDITION with the proper value
21585.      FIRST_ID with the first label
21586.      to insure proper test generation. */
21587.      if THISCONDITIONAL_REQD
21588.      then
21589.      CALL DO_GENERATE_CONDITIONAL_SET
21590.      /* Generate code to pass control to the ULTIMATE_FALLTHRU_LABEL (or
21591.      to fall through to it) if the conditional test specified by
21592.      SYSLIST(FIRST_INDEX) through SYSLIST(LAST_INDEX) has the logical
21593.      value stored in ULTIMATE_FALLTHRU_CONDITION; else pass control to
21594.      the ULTIMATE_BRANCH_LABEL. If a branch is generated to the
21595.      ULTIMATE_FALLTHRU_LABEL, set FALLTHRU_LABEL_USED to true;
21596.      else set it false. Include definition of any LABEL outstanding
21597.      before generating code. */
21598.      if PASS = 1
21599.      then
21600.      CALL DO_UNTIL_PCSTPROCESS
21601.      /* For those cases where the ULTIMATE_FALLTHRU_LABEL was not to
21602.      follow the conditional test as the next sequential instruction,
21603.      generate an unconditional branch to the ULTIMATE_FALLTHRU_LABEL
21604.      and clear FALLTHRU_LABEL_USED. */
21605.      if FALLTHRU_LABEL_USED
21606.      then
21607.      LABEL := ULTIMATE_FALLTHRU_LABEL
21608.      /* Generate label at next opportunity. */
21609.      fi
21610.      fi
21611.      fi
21612.      /* Termination: PASS incremented only (not modifiable by called
21613.      procs), must eventually exceed 2. */
"DO" MACRO -- 21 June 1973

21638.  DO UNTIL PREPROCESS
21639.  /* Must set up THIS_CONDITIONAL_REQD, FIRST_INDEX, LAST_INDEX,
21640.  UNIQUE_LABEL_ID, ULTIMATE_BRANCH_LABEL, ULTIMATE_FALLTHRU_LABEL,
21641.  ULTIMATE_FALLTHRU_CONDITION, and FIRST_ID. */

21642.  THIS_CONDITIONAL_REQD := UNTIL_COND_TEST
21643.  if UNTIL_COND_TEST
21644.  then
21645.  FIRST_INDEX := UNTIL_INDEX
21646.  LAST_INDEX := UNTIL_END_INDEX
21647.  ULTIMATE_BRANCH_LABEL := BLOCK_LABEL_PREFIX || 'END'
21648.  /* Flow point 2 normally connects to flow point 12. */
21649.  ULTIMATE_FALLTHRU_LABEL := BLOCK_LABEL_PREFIX || 'W1'
21650.  /* Flow point 3 usually falls through to flow point 4. */
21651.  ULTIMATE_FALLTHRU_CONDITION := false
21652.  UNIQUE_LABEL_ID := 'U'
21653.  FIRST_ID := BLOCK_LABEL_PREFIX || 'U'
21654.  docase OPERAND_FORMAT happ
21655.  or
21656.  case (3, 6, 9) /* UNTIL test ORed with WHILE test. */
21657.  ULTIMATE_FALLTHRU_CONDITION := IFMS
21658.  ULTIMATE_BRANCH_LABEL := BLOCK_LABEL_PREFIX || 'BEG'
21659.  qbar
21660.  case (11, 15, 17) /* UNTIL test ANDed with WHILE looping branch. */
21661.  ULTIMATE_FALLTHRU_LABEL := BLOCK_LABEL_PREFIX || 'LPB'
21662.  qbar
21663.  case (12, 16, 18) /* UNTIL test ORed with WHILE looping branch. */
21664.  ULTIMATE_BRANCH_LABEL := BLOCK_LABEL_PREFIX || 'LPB'
21665.  qbar
21666.  ULTIMATE_FALLTHRU_LABEL := BLOCK_LABEL_PREFIX || 'BEG'
21667.  qbar
21668.  case (20, 21, 22) /* UNTIL conditional test only. */
21669.  ULTIMATE_FALLTHRU_LABEL := BLOCK_LABEL_PREFIX || 'BEG'
21670.  qbar
21671.  qbar
21672.  endif
21673.  endcase
DO Macro -- 21 June 1973

21675. DECLARE DO_WHILE_PROCESS
21676.  /* Must set up THIS_CONDITIONAL_REQD, FIRST_INDEX, LAST_INDEX,
21677.       UNIQUE_LABEL_ID, ULTIMATE_BRANCH_LABEL, ULTIMATE_FALLTHRU_LABEL,
21678.       ULTIMATE_FALLTHRU_CONDITION, and FIRST_ID. */
21679.
21680.   THIS_CONDITIONAL_REQD := WHILE_COND_TEST
21681.   if WHILE_COND_TEST
21682.     then
21683.       FIRST_INDEX := WHILE_INDEX
21684.       LAST_INDEX := WHILE_END_INDEX
21685.       ULTIMATE_BRANCH_LABEL := BLOCK_LABEL_PREFIX || 'END'
21686.       /* Flow point 5 always branches to flow point 12. */
21687.       ULTIMATE_FALLTHRU_LABEL := BLOCK_LABEL_PREFIX || 'BEG'
21688.       /* Flow point 6 always falls through to flow point 7. */
21689.       ULTIMATE_FALLTHRU_CONDITION := true
21690.       UNIQUE_LABEL_ID := 'W'
21691.       FIRST_ID := BLOCK_LABEL_PREFIX || 'W1'
21692.   fi
21693.  end
"DC" Macro -- 21 June 1973

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

21704. FALLTHRU_LABEL_USED := false
21705. if LABEL #/ then /* A label is waiting to be generated. */
21706. if FIRST_ID = ' ' then
21707. /* No special label is required at the beginning of this conditional
21708. test. */
21709. FIRST_ID := LABEL /* Put the label on the conditional test. */
21711. else
21712. /* We also have a label waiting for the conditional test. */
21713. generate (LABEL || ' DS OH')
21714. /* Get the LABEL label out of the way. */
21716. fi
21717. fi
21718. call CONDITIONAL_EXPRESSION_PROCESSOR (FIRST_ID; SYSLIST)
21720. /* Generate code corresponding to the operands of the current set
21721. (WHILE or UNTIL) of the DO operands (referred to collectively as
21722. SYSLIST). Only the SYSLIST can be passed directly as arguments;
21723. the following variables are effectively arguments but are passed
21724. in global variables:
21725. FIRST_INDEX,
21726. LAST_INDEX,
21727. ULTIMATE_BRANCH_LABEL,
21728. ULTIMATE_FALLTHRU_LABEL,
21729. UNIQUE_LABEL_ID,
21730. FALLTHRU_LABEL_USED.
21731. process operands of the SYSLIST beginning with SYSLIST(FIRST_INDEX)
21732. through SYSLIST(LAST_INDEX), generating the indicated tests to pass
21733. control as indicated above. If a branch is made to the
21734. ULTIMATE_FALLTHRU_LABEL, then FALLTHRU_LABEL_USED is set, else
21735. it is unaltered. */
21736. Corp
21738. ELOG DO_UNTIL_POSTPROCESS
21739. /* Generate where required a branch to follow the UNTIL tests to
21740. transfer control to a non-sequential ULTIMATE_FALLTHRU_LABEL.
21741. See decision table, flow point 3. Insure FALLTHRU_LABEL_USED
21742. is turned off so the label will not be generated on the next
21743. sequential instruction. */
21744. dogase OPERAND_FORMAT ifany
21745. of
21746. case (11, 15-18)
21747. generate ('B  // ULTIMATE_FALLTHRU_LABEL)
21748. FALLTHRU_LABEL_USED := false
21749. esac
21750. esacdo
21751. gopro
21752. core
```
"DO" Macro -- 21 June 1973

21754. proc do_label_block
21755. /* If a begin label is required, generate it. */
21757. case operand_format
21758. of
21759. case (0,3,5,8-12,14,16-22)
21760. if label # block_label_prefix || 'BEG'
21761. then
21762. /* Begin label must be generated. */
21763. if label != ''
21764. then
21765. generate (label || ' DS OH')
21766. fi
21767. label := block_label_prefix || 'BEG'
21769. fi
21770. esac
21771. endproc
```
PROC DC_INFO_SAVE

/* Insert into stack all information required to close loop at
TERMINATE_DO_LOOP. */

case 80, 810, 811
/* One character codes indicating flow point to follow points 8,
10, and 11. */

88 := 'W' /* Assume branch at point 8 is to WHILE group (point 4). */
810, 811 := 'O'
/* Assume no looping branch (thus no branches at 10 and 11). */

/* Set 88. */

docase OPERAND_FORMAT only
of

case (2, 3, 11, 12, 15-18, 20) /* UNTIL conditional test but no UNTIL looping branch. */

80 := 'O' /* To flow point 1. */

case

case (9-10, 13, 14, 15, 21, 22) /* UNTIL looping branch or no UNTIL but WHILE looping branch. */

88 := 'L' /* Fall through to point 9. */

case

case (0) /* Infinite loop. */

80 := 'O' /* To flow point 7. */


case 8m366d0

if LOOPING_BRANCH_TYPE = 'NONE'
then

/* Set 810. */

docase OPERAND_FORMAT only
of

case (4, 7, 13, 15, 16)

810 := 'W' /* To flow point 4. */


case (5-9, 12, 16-17, 19, 22)

810 := 'B' /* To flow point 7. */


case

case (6, 8, 21)

811 := 'U' /* To flow point 1. */


case 8m366d0

/* Set 811. */

docase OPERAND_FORMAT only
of

case (4, 6-10, 13, 15, 16, 19, 21)

811 := 'W' /* Fall through to flow point 12 (end of DO block). */


case

case (5, 8, 14, 17, 18)

811 := 'U' /* To flow point 6. */


case

case (7, 9, 22)

811 := 'U' /* To flow point 1. */


case 8m366d0

/* Set 810. */

docase OPERAND_FORMAT only
of

case (4, 7, 13, 15, 16, 19, 21)

810 := 'W' /* To flow point 4. */


case

case (5-9, 12, 16-17, 19, 22)

810 := 'B' /* To flow point 7. */


case

case (6, 8, 21)

811 := 'U' /* To flow point 1. */


case 8m366d0

/* Set 811. */

docase OPERAND_FORMAT only
of

case (4, 6-10, 13, 15, 16, 19, 21)

811 := 'W' /* Fall through to flow point 12 (end of DO block). */


case

case (5, 8, 14, 17, 18)

811 := 'U' /* To flow point 6. */


case

case (7, 9, 22)

811 := 'U' /* To flow point 1. */


case 8m366d0

/* Set 810. */
**DO** Macro -- 21 June 1973

```
'8280. PROC DO_TRACE_COUNTERS
'8281. */ If debugging, generate block name and/or counters for block and
'8282. loop execution. */
'8283. if DEBUG_BLOCKCOUNTS_REQD or DEBUG_BLOCKNAMES_REQD
'8284. then
    if DEBUG_BLOCKCOUNTS_REQD
      generate (LABEL || ' ' || BLOCK_LABEL_PREFIX || 'DOL')
      LABEL := $01
      generate (' ' || BLOCK_LABEL_PREFIX || 'DOL')
    endif
    generate (' ' || BLOCK_LABEL_PREFIX || 'DOL')
    generate (' ' || BLOCK_LABEL_PREFIX || 'DOL')
    generate (' ' || BLOCK_LABEL_PREFIX || 'DOL')
    generate (' ' || BLOCK_LABEL_PREFIX || 'DOL')
  endif
  /* Generate branch around block name and/or block counts. */
  generate (LABEL || ' ' || BLOCK_LABEL_PREFIX || 'GO')
  LABEL := BLOCK_LABEL_PREFIX || 'GO'
'8285. if DEBUG_BLOCKNAMES_REQD
      generate (' ' || BLOCK_LABEL_PREFIX || 'DOL')
    endif
'8286. if DEBUG_BLOCKCOUNTS_REQD
      generate (LABEL || ' ' || BLOCK_LABEL_PREFIX || 'DOL')
    endif
  endif
'8287. if DEBUG_BLOCKNAMES_REQD
      generate (LABEL || ' ' || BLOCK_LABEL_PREFIX || 'DOL')
    endif
  endif
'8288. if DEBUG_BLOCKCOUNTS_REQD
      generate (LABEL || ' ' || BLOCK_LABEL_PREFIX || 'DOL')
    endif
  endif
'8289. if DEBUG_BLOCKNAMES_REQD
      generate (LABEL || ' ' || BLOCK_LABEL_PREFIX || 'DOL')
    endif
  endif
'8290. /*
```
ATEND macro

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 generated to the OD for the ONEXIT block. */

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

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

if CURRENT_NEST_LEVEL > NESTING_LIMIT then
mexit
fi

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

if ERROR_OCCURRED then
mexit
fi

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

if ATEND_GENNED then
rnote(8, ISTRC2301 MCRE THAN CNE "ATEND" IN BLOCK')
mexit
fi

BLOCK_LABEL_PREFIX := ''II BLOCK_NUMBER(CURRENT_NEST_LEVEL)
TDL_GENNED := INFO[5,1]
/* See if we've already generated the loop-terminating code. */
FIN_LABEL_REQD := INFO[8,1]
/* Note whether a FIN label has already been referenced. */

if ~TDL_GENNED then

call TERMINATE_DO_LOOP ( ; )
/* Terminate the loop by generating any necessary back branches. */
endif

FIN_LABEL_REQD := true
/* Terminate the ONEXIT block. */

if END_LABEL_REQD(CURRENT_NEST_LEVEL) then

END_LABEL_REQD(CURRENT_NEST_LEVEL) := false
/* If normal block termination required an END label, provide it and note that we no longer require it. */
endif

INFORMATION(CURRENT_NEST_LEVEL) := INFO[1,4] ||
FIN_LABEL_REQD(CURRENT_NEST_LEVEL)
/* Forward FIN_LABEL_REQD to OD. */

mend
"ONEXIT" Macro -- 31 October 1973

25001. Macro ONEXIT ( : USER_NAME)
25002. /* The ONEXIT macro causes the generation of the loop-terminating code
25003. for the surrounding DO block if such code has not yet been generated.
25004. The target for abnormal loop termination (EXIT macros) is then defined
25005. to allow the code which follows to be executed upon abnormal loop
25006. termination. If the ONEXIT has been preceded by an ATEND macro,
25007. the branch is generated to the OD for the ATEND block. */
25008. bit ONEXIT_GENNED, /* Indicates whether ONEXIT has been generated
25009. previously for this block. */
25010. TDL_GENNED, /* Indicates whether the TERMINATE_DO_LOOP macro has
25011. been evoked for this DO by a previous macro (properly, only
25012. by an ATEND). */
25013. FIN_LABEL_REQD /* Indicates a branch to the label "SnnnFIN" has
25014. been generated and must be defined at OD time. */
25015. char INFO /* Holds copy of INFORMATION(CURRENT_NEST_LEVEL).*/
25016. call TRACE PRINTER ( ; 'ONEXIT')
25017. if CURRENT_NEST_LEVEL > NESTING_LIMIT
25018. then
25019. mexit
25020. fi
25021. call VERIFY_END ( ; 'DO' USER_NAME)
25022. /* Verifies current block has the name specified
25023. by the USER_NAME
25024. operand of the ONEXIT macro (if any) and that it is a DO block.
25025. Various errors receive messages and either intermediate blocks are
25026. BLENDed as a fixup or ERROR_OCCURRED is set. */
25027. if ERROR_OCCURRED
25028. then
25029. mexit
25030. fi
25031. INFO := INFORMATION (CURRENT_NEST_LEVEL)
25032. ONEXIT GENNED := INFO[7,1]
25033. /* See if we've already generated an CNEXII. */
25034. if ONEXIT GENNED
25035. then
25036. anote (8, 'SRC2501 MORE THAN ONE "ONEXIT" IN BLOCK')
25037. mexit
25038. fi
25039. if -EXIT_LABEL_REQD(CURRENT_NEST_LEVEL)
25040. then
25041. mnote (8, 'SRC2502 NO EXIT FOR THIS "DO"')
25042. fi
25043. FIRLABEL_BEQD := INFO[8,1]
25044. /* Note whether a FIN label has already been referenced. */
25045. BLOCK_LABEL_PREFIX := IS' ' | BLOCKNUMBER(CURRENT_NEST_LEVEL)
25046. TDL_GENNED := INFO[5,1]
25047. /* See if we've already generated the loop-terminating code. */
25048. if -TDL_GENNED
25049. then
25050. call TERMINATE_DO_LOOP ( ; )
25051. /* Looping branch expects to fall through to END label. */
25052. generate (' B ' | BLOCK_LABEL_PREFIX | 'END')
25053. else
25054. /* TERMINATE_DO_LOOP must have been done by previous ATEND. */
25055. generate (' B ' | BLOCK_LABEL_PREFIX | 'FIN')
25056. FIN_LABEL_REQD := true
25057. /* Provide branch to FIN for ATEND block. */
25058. generate (BLOCK_LABEL_PREFIX | 'XIT DS OD')
25059. EXIT_LABEL_REQD(CURRENT_NEST_LEVEL) := false
25060. /* Provide target for EXIT branch and note that it is no longer
25061. needed. */
25062. INFORMATION(CURRENT_NEST_LEVEL) := INFO[1,4] | INFO[6,1] | INFO[8,1]
25063. TDL := INFO
25064. FIN_LABEL_REQD := false
25065. /* Forward FIN_LABEL_REQD to OD. */
25066. mend

25000. C-32
/* Terminate DO loop if ATEND or ONEXIT have not done so and do
standard block closing. */

TDL_GENNED, /* Indicates whether the looping code has been generated
yet. */

FIN_LABEL_REQD /* Indicates whether a "$nnnFIN" label is
required. (It is used at the end of the ATEND or ONEXIT code. */

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

/* Prints macro name "OD" in mnote if tracing on. */

if CURRENT_NEST_LEVEL > NESTING_LIMIT
then
  call VERIFY_END ( ; 'DO', USER_NAME)
  /* Verifies current block has the name specified by the USER_NAME
operand of the OD 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.
(Lemma: IF CURRENT_NEST_LEVEL > 0 and
  [USER_NAME = '' or = BLOCK_NAME(CURRENT_NEST_LEVEL)] and
  BLOCK_TYPE(CURRENT_NEST_LEVEL) = 'DO', then
  ERROR_OCCURRED will be set false and CURRENT_NEST_LEVEL will
not be modified.) */
if ERROR_OCCURRED
then
  mexit
else
  /* ATEND or ONEXIT occurred; we may need FIN label. */
  if FIN_LABEL_REQD
    generate ('$' II BLOCK_NUMBER(CURRENT_NEST_LEVEL) II 'FIN DS OH')
  fi
fi
fi

call POP_OLD_BLOCK ( ;)
/* Delete current block from the stack.
(lemma: POP_OLD_BLOCK decrements CURRENT_NEST_LEVEL by exactly
one.) */

*DOCASE* Macro -- 26 June 1973

```markdown
31001. **MACRO** DOCASE (USER_NAME; INDEX, OPTION, RANGE)
31002. /* 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. */
31005. bit
31006. 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). */
31009. 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. */
31114. char
31116. INDEX_REG, /* Name of register containing DOCASE index, if
31117. in a register. */
31119. INDEX_LENGTH, /* Length (or symbol indicating length) of index for
CHARCOMP operand. */
31121. 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). */
31123. 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). */
31129. LABEL, /* Any outstanding label, to be generated on next executable
instruction. */
31131. INDEX_ADDR /* Symbolic address of byte-type or CHARCOMP index, if
any; else null. */
31134. /* 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 all but falling through to the first CASE). */
```

C-34
CALL TRACE_PRINTER ( ; "DOCASE")
/* Print macro name "DOCASE" in mnote if tracing on. */
CALL PUSH_NEW_BLOCK (USER_NAME;
  BLOCK_TYPE_VALUE='DOCASE';
  END_LABEL_VALUE=TRUE)
/* Define new block; add to stack. Initialize block specifications.
Note that block will need an END label. Set up unique
BLOCK_LABEL_PREFIX for generating unique labels. */
IF ERROR_OCCURRED THEN BEXIT
LABEL := USER_NAME /* Generate macro's label at first opportunity. */
CALL EXTRACT_OPERANDS
/* Validate operands and issue any error-messages; set INDEX_REG,
INDEX_TYPE, INDEX_RANGE_ASSURED, INDEX_LENGTH and CASE_FORMAT. */
IF CASE_FORMAT = "CONDTEST" THEN
  CALL INDEXTO_REGI /* If case format is GENERAL, SPARSE, or CHARCOMP and the
index is a byte, save symbolic address of the index in INDEX_ADDR, otherwise
set INDEX_ADDR to null and generate code to put index into GPR1. */
  IF CASE_FORMAT = "SIMPLE" THEN
    CALL GENERAL_SETUP /* Generate branch to general handler for GENERAL format. In any
case (GENERAL, SPARSE, or CHARCOMP), advance GCASE_NEST_LEVEL for the
CASE stack and initialize the GCASE globals. */
  FI
  CALL DEBUGSTUFF /* Generate last-case variable and block-name constant if required. */
  CALL INFO_SAVE
  /* Store in stack all data needed by CASE and ESACOD to complete
CASE processing. */
  IF LABEL = "*" THEN
    GENERATE (LABEL | " DS GH")
  FI
FI
MEND

*C-35*
"DOCASE" macro -- 26 June 1973

31082.  ESEG DOCASE_EXTRACT_OPERANDS
31083.  /* Validate operands and issue any error messages; set INDEX_REG,
31084.  INDEX_TYPE, INDEX_RANGE_ASSURED, INDEX_LENGTH, and CASE_FORMAT. */
31086.  IF OPTION = 'SIMPLE' OR = 'SPARSE'
31087.  THEN
31088.  CASE_FORMAT := OPTION
31089.  ELSE
31090.  IF OPTION = 'ONLY'
31091.  THEN /* Allow range specification as second operand of macro, also. */
31092.  INDEX_RANGE_ASSURED := TRUE
31093.  ELSE
31094.  IF OPTION = 'ADD # 'IFANY'
31095.  THEN
31096.  WRITE (6, 'STRC3102 ' || OPTION ||
31097.  ' INVALID SECOND OPERAND—JNGNEED')
31098.  IF INDEX = '1'
31099.  THEN
31100.  CASE_FORMAT := 'CONTEST' /*
31101.  ELSE
31102.  CASE_FORMAT := 'GENERAL'
31103.  IF INDEX(1) = 'IFANY' OR = 'ONLY'
31104.  THEN
31105.  WRITE (6, 'STRC3101 WARNING—"" || INDEX(1) ||
31106.  ' "" ASSUMED AS INDEX; USE "DOCASE " || INDEX(1) ||
31107.  ' FOR RANGE SPEC')
31108.  IF RANGE = 'ONLY'
31109.  THEN
31110.  INDEX_RANGE_ASSURED := TRUE
31111.  ELSE
31112.  IF INDEX = 'ADD # 'IFANY'
31113.  THEN
31114.  WRITE (6, 'STRC3101 "" || RANGE ||
31115.  ' "" ASSUMED AS INDEX; USE "DOCASE " || INDEX ||
31116.  ' FOR RANGE SPEC')
31117.  IF INDEX = '1'
31118.  THEN
31119.  INDEX := '0' /* Assume not CHARCOMP */
31120.  IF INDEX = 'ADD INDEX(1) = '1'
31121.  THEN /* A one-element sublist was specified; we take it to be a
31122.  INDEX_REG := INDEX(1) /* Index is specified as a register. */
31123.  ELSE
31124.  INDEX_REG := '1'
31125.  END
31126.  END /* Get index type specified; should be "w" (word), "h" (halfword),
31127.  "b" (byte), or length of CHARCOMP index. */
31128.  IF INDEX_TYPE = 'w' AND = 'n' AND = 'h'
31129.  THEN
31130.  INDEX_LENGTH := INDEX_TYPE /* Operand two is length specification. */
31131.  CASE_FORMAT := 'CHARCOMP' /* Change format to CHARCOMP. */
31132.  ELSE
31133.  INDEX_TYPE := 'w' /* No type specified; "w" is default. */
31134.  END
31135.  END
31136.  END
31137.  END
31138.  END
31139.  END
31140.  END
31141.  END
31142.  END
31143.  END
31144.  END
31145.  END
31146.  END

C-36
"DOCASE" Macro  --  26 June 1973

31149. PROG DOCASE_INDEX_TO_REG1
31150. /* If case format is GENERAL, SPARSE, or CHARCGMP 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 COBTEST format. */
31154.
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
31161. generate (LABEL 11 ' LR 1,' || INDEX_REG)
31162. LABEL := ''
31163. fi
31164. esac
31165. CASE ('W') /* Word index. */
31166. generate (LABEL 11 ' L 1,' || INDEX(1))
31167. LABEL := ''
31168. esac
31169. CASE ('H') /* Halfword index. */
31170. generate (LABEL 11 ' LH 1,' || INDEX(1))
31171. LABEL := ''
31172. esac
31173. CASE ('B') /* Byte index. */
31174. IF CASE_FORMAT = 'SIMPLE'
31175. then
31176. generate (LABEL 11 ' SR 1,' || INDEX(1))
31177. LABEL := ''
31178. esac
31179. generate (' IC 1,' || INDEX(1))
31180. INDEX_ADDR := INDEX(1)
31181. /* Postpone loading of index into register; we may want to do CLI's. */
31182. fi
31183. esac
31184. esac
31185. C-37
31187. PROC DOCASE_GENERAL_SETUP
31188. /* Generate branch to beginning of general handler for general format
31189.   DOCASE. In any case, advance GCASE_NEST_LEVEL for the GCASE stack
31190.   and initialize the GCASE globals. It is assumed that this proc is
31191.   called only for GENERAL, SPARSE, and CHARCOMP case formats. */
31192.   int I, J /* Temporaries. */
31193.
31194. if CASE_FORMAT = 'GENERAL'
31195.   then
31196.     generate (LABEL || 'B' || BLOCK_LABEL_PREFIX || 'BEG')
31197.     /* Generate branch to general handler which is defined at ESACOD. */
31198.   else
31199.     branch_to_case := false /* Alternatively. */
31200.   fi
31201. GCASE_NEST_LEVEL := GCASE_NEST_LEVEL + 1 /* Advance GCASE stack. */
31202.   if GCASE_NEST_LEVEL < GCASE_NEST_LIMIT
31203.     then /* Clear GCASE globals. */
31204.       MAX_CASE_VALUE(GCASE_NEST_LEVEL) := -1
31205.     /* Maximum branch vector value found. */
31206.       NEXT_COMP_LABEL_NO(GCASE_NEST_LEVEL) := 1
31207.     /* Case number for next comparison case label to be generated. */
31208.     J := GCASE_NEST_LEVEL * 256
31209.   if CASE_FORMAT = 'GENERAL'
31210.     then /* Clear CASE_OCCURS bits. */
31211.       while I <= J 
31212.         CASE_OCCURS(I) := false
31213.         I := I + 1
31214.       od /* (Termination: I is incremented, J is fixed during loop; I
31215.       must eventually exceed J.) */
31216.   else /* GCASE stack overflow. */
31217.     mnote (12, 'STRC3104 GENERAL/SPARSE/CHARCOMP GCASE NESTING LEVEL ' ||
31218.     GCASE_NEST_LEVEL || ' EXCEEDS MAXIMUM OF ' ||
31219.     GCASE_NEST_LIMIT || '--MACROS MUST BE MODIFIED')
31220.   fi
31221.  end

"DOCASE" macro -- 26 June 1973

C-38
"DOCASE" Macro -- 26 June 1973

31226. DEFINE DOCASE_DEBUG_STUFF
31227. /* Generates last-case variable and block-name constant if required. */
31228. char X /* Temporary. */
31229.
31230. if DEBUG_BLOCKCOUNTS_REQD or DEBUG_BLOCKNAMES_REQD
31231. then
31232. if ~ BRANCH_TO_CASE1
31233. then
31234. /* Branch must be generated around the last-case variable and/or block-name constant. Put target suffix into X. */
31235. if CASE_FORMAT = 'GENERAL'
31236. then
31237. /* This case should not occur since DOCASE_GENERAL_SETUP generates the branch for GENERAL cases; we include the code here for completeness. */
31238. X := 'BEG'
31239. else
31240. X := 'C1'
31241. fi
31242.
31243. if DEBUG_BLOCKNAMES_REQD
31244. then
31245. generate (LABEL || B || BLOCK_LABEL_PREFIX || X)
31246. fi
31247.
31248. if DEBUG_BLOCKCOUNTS_REQD
31249. then
31250. generate ('" DC C" || BLOCK_NAME(CURRENT_NESTLEVEL) || ""')
31251. fi
31252. if DEBUG_BLOCKCOUNTS_REQD
31253. then
31254. generate (BLOCK_LABEL_PREFIX || "LSC DC X'00' LAST CASE NUMBER")
31255. fi
31256. fi
31257. fi
31258. /*
31259. */
31260. fi
31261. /*
"DOCASE" Macro -- 26 June 1973

PROC DOCASE_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_CASE  || CASE  || 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. */

END
```
"CASE" Macro -- 27 June 1973

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

C-41"
call TRACE.Printer ( ; "CASE")
/* Print macro name "CASE" in mnote if tracing on. */
call CASE_POSITION.CHECK
/* Verifies mother block is a DOCASE or attempts fixup with up to 2 BLENDs. Indicates whether un-fixup-able ERROR_OCCURRED. */
if ERROR_OCCURRED then
    mnote (8, "STRC3304 "CASE" TREATED AS "BLCCK" MACRO")
call BLOCK (USER_NAME;)
mexit
if ERRCR_OCCURRED then
    mnote (8, 'STRC3304 "CASE" TREATED AS "BLCCK" MACRO')
call BLOCK (USER_NAME;)
mexit
if REL # '' or MASK # '' then
    mnote (8, 'STRC3310 REL= OR MASK= NOT IN PARENTHESES—IGNORED')
call CASE_GET_DOCASE_INFO
/* Extract CASE_FORMAT, CASE_LABEL_REQD, CASE_COUNTER, MISC_FOUND, MAMA_BLOCK_PREFIX, INDEX_RANGE_ASSURED, INDEX_ADDR, and INDEX_LENGTH from mother DOCASE block. */
if USER_NAME then
    generate (USER_NAME II 'DS OH')
    /* Any USER_NAME on a CASE macro is just a comment since a branch to it will produce unpredictable results. If one was specified, get it out of the way now. */
fi
if SYSLIST(1) = 'MISC' then
call CASE_MISC_PROCESS
/* Completely process miscellaneous CASE block. */
else
    if CASE_FORMAT = 'GENERAL' or = 'SPARSE' or = 'CHARCOMP' then
        if GCASE_NEST_LEVEL < GCASE_NEST_LIMIT then
            call CASE_PROCESS_COMPARE_OPERANDS
            /* 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. */
            if CASE_FORMAT = 'GENERAL' then
                call CASE_PROCESS_VECTOR_OPERANDS
                /* Generate labels and save information about any operands which are to be handled via branch vector. */
            fi
        fi
    fi
else
    if must be SIMPLE or COMDIST type. */
    call CASE_GET_NAME
    /* Set LABEL if label required on first of code (usually is; only exception is when DOCASE is falling through to first CASE macro). Set NEXT_CASE with label to be used on next case. */
    if CASE_FORMAT = 'SIMPLE' then
        call CASE_BCT_GEN
        /* Generate BCT instruction for this case. */
    else
        call CASE_CONDTEST_GEN
        /* Generate conditional test specified on CASE macro. */
    fi
fi
call CASE_TRACE_COUNTER
/* Generate code to count this block, note last case number, and/or display block name if appropriate debugging requested. */
call CASE_UPDATE_INFO
if LABEL # '' then
    generate (LABEL II 'DS OH')
```c
33129.  // CASE
33130.  /* Verifies mother is DOCASE macro or attempts fixup by inserting up
33131.  to two BLENDs (if that will get us to a DOCASE mother).  Indicates if
33132.  no fixup possible in ERROR_OCCURRED. */
33133.  if BLOCK_TYPE(CURRENT_NESTLEVEL) # 'DOCASE'
33134.    execute (8, 'STRC3301 ASSUMING TWO BLEND OMITTED-INSERTED')
33135.    call BLEND ( ; )
33136.    ERROR_OCCURRED := false /* Note patch up. */
33137.  else
33138.    execute (8, 'STRC3302 ASSUMING BLEND OMITTED-INSERTED')
33139.    ERROR_OCCURRED := false /* Note patch up. */
33140.  fi
33141.  if CURRENT_NESTLEVEL > 2 and
33142.    BLOCK_TYPE(CURRENT_NESTLEVEL-2) = 'DOCASE'
33143.    execute (8, 'STRC3303 ASSUMING TWO BLEND OMITTED-INSERTED')
33144.    call BLEND ( ; )
33145.    ERROR_OCCURRED := false
33146.  fi
33147.  fi
33148.  fi
33149.  fi
33150.  fi
33151.  fi
33152.  fi
33153.  fi
33154.  fi
33155.  fi
33156.  fi
```

/*...*/
/* Generate compare-and-branch sequences for all "symbolic" operands
   (i.e., those which cannot be handled by the branch vectors: all non-
   self-defining terms, all self-defining operands which are not in
   the range 0-255 inclusive, and all "range" operands (m, R)
   where either m or R is either non-self-defining or outside the
   range 0-255) or for all operands if LOCASE was flagged as Sparse or
   Charcomp. */

I := 1 /* Start search with first operand. */
COMP_LABEL_NO := NEXT_COMP_LABEL_NO(GCASE NEST LEVEL)
/* Note the next compare label number. */
EQUAL_TEST_OUTSTANDING, RANGE_TEST_OUTSTANDING := false
while I ≤ N'SYSLIST
  do if (CASE_FORMAT = 'SPARSE' or = 'CHARCOMP' or
       [N'SYSLIST(I) ≤ 1 and (T'SYSLIST(I) ≠ 'N')
        or SYSLIST(I) > 1 and (T'SYSLIST(I) ≠ 'N')
        or SYSLIST(I,1) < 0 or > 255)
       or (N'SYSLIST(I) > 1 and
        SYSLIST(I,1) < 0 or > 255)
        or T'SYSLIST(I,2) ≠ 'N' or
        SYSLIST(I,2) < 0 or > 255))
  then if EQUAL_TEST_OUTSTANDING
  then LABEL := BLOCK_LABEL_PREFIX || 'BEG'
    generate (' BE ' || LABEL)
    /* After leaving this proc, someone will generate the BEG label
     at the beginning of the block. */
    EQUAL_TEST_OUTSTANDING := false
  else if RANGE_TEST_OUTSTANDING
  then LABEL := BLOCK_LABEL_PREFIX || 'BEG'
    generate (' BN ' || LABEL)
    /* Again, by leaving BEG label in LABEL, it will be generated
     when we branch on lower end of range. */
    RANGE_TEST_OUTSTANDING := false
    else if CASE_LABEL_REQD
    then COMP_LABEL := MAMA_BLOCK_PREFIX || 'C' || COMP_LABEL_NO
    /* Generate label name to be attached to first instruction. */
    fi
    fi
  fi
  while I ≤ N'SYSLIST
  do /* Advance to next operands of CASE. */
    od /* (Termination: I is incremented above and not modified by
called proc; N'SYSLIST is fixed; I must eventually exceed
    COMP_LABEL_NO. */
if EQUAL_TEST_OUTSTANDING
  then /* Generate branch to next symbolic case. */
    generate (' BNE ' || MAMA_BLOCK_PREFIX || 'C' ||
    COMP_LABEL_NO)
  else if RANGE_TEST_OUTSTANDING
  then generate (' BN ' || COMP_LABEL)
    /* Generate branch to next compare case. Label was left in COMP_LABEL
     when we branched on lower end of range. */
    fi
  fi
NEXT_COMP_LABEL_NO(GCASE_NEST_LEVEL) := COMP_LABEL_NO
/* Store case number of next symbolic case to be defined. */
"CASE" Macro -- 27 June 1973

33261. PROC CASE_GEN_COMPARE
33262.    /* Generate compare for the single compare operand at SYSLIST(I)--
33263.    either general case non-self-definer or any SPARSE or CHARCOMP
33264.    operand. DOCASE index is at INDEX_ADDR unless that's null, then in
33265.    GPR1. Length is in INDEX_LENGTH for CHARCOMP type. Any label to be
33266.    generated in in COMP_LABEL; any branch target at exit is to be
33267.    put into COMP_LABEL. Also on exit, EQUAL_TEST_OUTSTANDING or
33268.    RANGE_TEST_OUTSTANDING should be set to indicate which type of
33269.    operand was processed. Operands may be of the form $m$ or ($m$, $n$),
33270.    the latter implying the range from $m$ to $n$. $m$ and $n$ may be
33271.    self-defining terms or symbols Equated to absolute expressions for
33272.    GENERAL or SPARSE format; for CHARNAME, they may be absolute or
33273.    symbolic addresses of character strings or may be literals (with the
33274.    leading "c" and, for character literals, "C" possibly omitted). */
33275.    CHAR INSPT /* Temporary. */

C-46
if INDEX_ADDR = ""
then /* Index is in GPR */
generate (COMP_LABEL || ' LA 0, | SYSLIST(I,1))
else /* Index is at INDEX_ADDR */
if CASE_FORMAT = 'CHARCMP'
then
   INSERT := ""
   if SYSLIST(I,1)[1,1] < 4
   then
      if SYSLIST(I,1)[1,1] = ""
      then
         /* Case format: "" */
      INSERT := "="
      fi
   else
      if SYSLIST(I,1)[1,1] = ""
      then /* Literal without the "" (operand ends with "") */
      INSERT := "="
      fi
   fi
   generate (COMP_LABEL || ' CLC ' | INDEX_ADDR ' (' | INDEX_LENGTH | ' '), | SYSLIST(I,1))
else
   generate (COMP_LABEL || ' CLI ' | INDEX_ADDR ' ', | SYSLIST(I,1))
fi
if COMP_LABEL = ""
then
   COMP_LABEL := ""
   COMP_LABEL_NO := COMP_LABEL_NO + 1
fi
if SYSLIST(I) < 1
then /* Operand is not a range. */
   EQUAL_TEST_OUTSTANDING := true
else /* A range has been specified: (g,*) */
   if SYSLIST(I) > 2
   then
      mnote (8, 'STRC3312 ' | SYSLIST(I) | '0:INVALID--ONLY FIRST TWO SUBOPERANDS PROCESSED')
      fi
   RANGTESTOUTSTANDING := true
   fi
   range (8) | SYSLIST(I) | '1:INVALID--ONLY FIRST TWO SUBOPERANDS PROCESSED')
   fi
   generate (COMP_LABEL || ' CLC ' | INDEX_ADDR ' (' | INDEX_LENGTH | ' '), | SYSLIST(I,2))
if CASE_FORMAT = 'CHARCOMP'
then
   INSERT := ""
   if SYSLIST(I,2)[1,1] < 4
   then
      if SYSLIST(I,2)[1,1] = ""
      then /* Character string */
      INSERT := "="
      fi
   else
      if SYSLIST(I,2)[1,1] = ""
      then /* Literal */
      INSERT := "="
      fi
   fi
   generate (COMP_LABEL || ' CLC ' | INDEX_ADDR ' (' | INDEX_LENGTH | ' '), | SYSLIST(I,2))
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 \( g \) or \((g,m)\) where \( g \) and \( m \) are self-defining terms in the range 0-255 inclusive. This procedure assumes the CASE_FORMAT is general (not sparse). */

```c
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
if N'SYSLIST > 0 then /* One or more operands were specified. */
I := 1 /* Start with first operand. */
while I ≤ N'SYSLIST do
  if T'SYSLIST(I,2) = 'g'
    then /* g is a self-defining term. */
      OP := SYSLIST(I,1)
      if OP is in the range. */
        if T'SYSLIST(I,2) = '0' or '1'
          then /* g is self-defining or not present. */
            if T'SYSLIST(I) > 2
              then
                mnote (8, 'STRC3301 II SYSLIST(I) II INVALID--ONLY FIRST TWO SUBOPERANDS PROCESSED')
            fi
            if T'SYSLIST(I,2) = 'N'
              then
                LIMIT := SYSLIST(I,2)
                if LIMIT ≤ 255 add OP > 0
                  then /* It's in the right range also. */
                    if LIMIT < OP
                      then
                        mnote (8, 'STRC3305 II SYSLIST(I) II "INVALID--" II OP || "ASSUMED")
                    fi
                    LIMIT := OP
                  fi
                else
                  LIMIT := OP
                fi
                if LIMIT > MAX_CASE_VALUE(GCASE_NEST_LEVEL)
                  then /* We have found a new maximum case number. */
                    MAX_CASE_VALUE(GCASE_NEST_LEVEL) := LIMIT
                  fi
                if OP ≤ LIMIT
                  then CASE_OCCURS(BASE+OP) := true
                    SUBCASE(MAMA_BLCCK_PREFIX || 'G' || OP || ' DS DE')
                  fi
            fi
          fi
        fi
      if OP/4*4 = 0
        then MULTIPLEOF4 := false
      fi
    fi
  fi
  I := I + 1 /* Do do next operand. */
  if OP is incremented, LIMIT is fixed during loop; or must eventually exceed LIMIT. */
```
"CASE" Macro -- 27 June 1973

```assembly
33434. EXEM 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. in GUESS /* Issued 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_OCCURS(BASE+GUESS ) := LREH
33449. generate (MAMA_BLOCK_PREFIX || 'G' || GUESS || 'ES OH')
33450. if UNEXPECTED_OPERANDS_FOUND
33451. then
33452. NOTE ("STRC3306 EARLIER UNEXPECTED Operand IMPLIES THIS TO BE CASE ||
33453. GUESS")
33454. fi
33455. fi
33456. COLD
```
CASE Macro -- 27 June 1973

33459. BLOCK CASE_SET_NAMES
33460. /* Set LABEL if one will be required on this SIMPLE or CONDTESTEST 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_REQD
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

C-50
```
33473. PEGG CASE_BCT_GEN
      
33474. /* Generate BCT for this simple case. Verify operand, if any. */
33475.
33476.  GASAGE (LABEL || ' BCT 1,' || NEXT_CASE)
33477.  LABEL := "
33478.  if T'SYSLIST(1) # '0'
33479.    then /* An operand was specified. */
33480.      if T'SYSLIST(1) = 'W'
33481.        then /* Operand is a self-defining term. */
33482.          if SYSLIST(1) = CASE_COUNTER
33483.            then
33484.              STOP (9, 'STBC3305 OPERAND INVALID VALUE OF SIMPLE CASE ' ||
33485.              CASE_COUNTER)
33486.      fi
33487.    else /* Operand is not self-defining term. */
33488.      STOP (9, 'STBC3309 OPERAND MUST BE SELF-DEFINING TERM OR OMITTED ' ||
33489.      CASE_COUNTER)
33490.    fi
33491.    fi
33492. end
```
$\text{CASE}$ Macro -- 27 June 1973

33494. PROC CASE_CONDTESTGEN
33495. /* Generate conditional test indicated by operands. */
33496. int OP_COUNT /* Number of operands for instruction being passed
33497. to SIMPLE_CONDITIONAL. */
33498. ULTIMATE_BRANCH_LABEL := NEXT_CASE
33499. ULTIMATE_FALLTHRU_LABEL := BLOCK_LABEL_PREFIX || 'beg'
33500. ULTIMATE_FALLTHRU_CONDITION := true
33501. FALLTHRU_LABEL_USED := false
33502. FIRST_INDEX := 1
33503. LAST_INDEX := N'SYSLIST
33504. /* Process entire operand list as a single conditional expression. */
33505. UNIQUE_LABEL_ID := 'i'
33506. CALL CONDITIONAL_EXPRESSION_PROCESSOR (LABEL; SYSLIST)
33507. /* Generate code corresponding to the operands of the CASE macro
33508. (referred to collectively as SYSLIST). Only the SYSLIST can be passed
33509. directly as arguments; the following variables are effectively
33510. arguments but are passed in global variables:
33511. FIRST_INDEX,
33512. LAST_INDEX,
33513. ULTIMATE_BRANCH_LABEL,
33514. ULTIMATE_FALLTHRU_LABEL,
33515. ULTIMATE_FALLTHRU_CONDITION,
33516. UNIQUE_LABEL_ID,
33517. FALLTHRU_LABEL_USED.
33518. Process operands of the SYSLIST beginning with SYSLIST(FIRST_INDEX)
33519. through SYSLIST(LAST_INDEX) (for the CASE macro, this is the entire
33520. SYSLIST), generating the indicated test to pass control to the
33521. ULTIMATE_FALLTHRU_LABEL if the test succeeds, else to the
33522. ULTIMATE_BRANCH_LABEL. The UNIQUE_LABEL_ID is used to insure
33523. unique labels where needed. If a branch is made to the
33524. ULTIMATE_FALLTHRU_LABEL, then set FALLTHRU_LABEL_USED; else
33525. it is unaltered. */
33526. if FALLTHRU_LABEL_USED
33527. then
33528. LABEL := BLOCK_LABEL_PREFIX || 'beg'
33529. else
33530. LABEL := ""
33531. fi
33532. fi
33533. return
**CASE** Macro -- 27 June 1973

```plaintext
if ~CASE_LABEL_REQD
  // We are falling through into this block. */
  generate ('B' || MAMABLOCK_PREFIX || 'I' || 'C' || CASE_COUNTER)
  /* Generate branch to next case number (probably C). */
fi
if MISC_FOUND
  mnote (8, 'STRC3311 MULTIPLE MISC CASES IN DOCASE--THIS BLOCK ' || 'IS DEAD CODE')
else
  LABEL := MAMABLOCK_PREFIX || 'MSC'
  /* Make MSC label outstanding (generate on next instruction). */
  if INDEX_RANGE_ASSURED
    then
      mnote (8, 'STRC3308 "DOCASE ...ONLY" INVALID WITH MISC')
      INDEX_RANGE_ASSURED := false
    fi
  fi
```
"CASE" Macro -- 27 June 1973

33565. PROC CASE_TRACE_COUNTER
33566. */ Generate any debugging counters and/or labels requested. */
33567.
33568. if DEBUG_BLOCKCOUNTS_REQD or DEBUG_BLOCKNAMES_REQD
33569.    then
33570.       if DEBUG_BLOCKCOUNTS_REQD
33571.       then
33572.           /* Generate code to advance this case's counter. */
33573.           generate (LABEL || ' LH 1, ' || BLOCK_LABEL_PREFIX || 'CTR')
33574.           LABEL := ' || LA 1, 1('1')
33575.           generate (' || STH 1, ' || BLOCK_LABEL_PREFIX || 'CTR')
33576.           if SYSLIST(1) = 'MISC' or CASE_COUNTER > 255
33577.               then
33578.                 generate (' || MVI ' || BLOCK_LABELPREFIX || 'LSL, X'FF')
33579.               else
33580.                 HEX_IN := CASE_COUNTER
33581.                 call XHEX ( )
33582.                 generate (*, "STBC3313 CASE DEBUG ID=X'" || HEX_OUT || X'00")
33583.                 if DEBUG_BLOCKNAMES_REQD
33584.                     then
33585.                       generate (" || MVI ' || BLOCK_LABEL_PREFIX || 'LSL, X'0' || " CASE NUMBER FOR TRACING")
33586.                   fi
33587.                   fi
33588.                   fi
33589.                   fi
33590.                   fi
33591.               if DEBUG_BLOCKNAMES_REQD
33592.               then
33593.                   generate (" || BLOCK_NAME(CURRENT_NEST_LEVEL) || ")
33594.               fi
33595.               if DEBUG_BLOCKCOUNTS_REQD
33596.               then
33597.                   generate (' || DC C' || BLOCK_LABEL_PREFIX || 'CTR')
33598.               fi
33599.               fi
33600.               fi
33601.               fi
33602.               fi

C-54
CASE UPDATE INFO

33604. **DLOG CASE_UPDATE_INFO**
33605. /* Returns to another DOCASE level possibly updated information which
33606. was extracted by CASE_GET_DOCASE_INFO. */
33607. int **MO** /* Index level of DOCASE block. */
33609. **MON := CURRENT_NEST_LEVEL - 1**
33610. **INFORMATION(MON) := 0** || **MISC_FOUND** || **MULTIPLE_SOF4** ||
33611. **NO_EXPECTED_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. **CDEP**
"ESAC" Macro -- 3 July 1973

35001. BACKED ESAC { ; USER_NAME}  
35002. /* Generate end to match CASE block. Do standard block closing, then  
35003. generate branch to end of mother DOCASE block. */  
35004. call TRACE_PRINTER { ; 'ESAC'}  
35005. /* Print macro name "ESAC" in mnote if tracing on. */  
35006. if CURRENT_NEST_LEVEL <= NESTING_LIMIT  
35007. then  
35008. call VERIFY_END { ; 'CASE', USER_NAME}  
35009. /* Verify current block has the name specified by the USER_NAME  
35010. operand of the ESAC macro (if any) and that it is a CASE block.  
35011. Various errors receive messages and either intermediate blocks are  
35012. blended as a fix-up or ERROR_OCCURRED is set.  
35013. (Lemma: If CURRENT_NEST_LEVEL > 0 and  
35014. [USER_NAME = '' or = BLOCK_NAME(CURRENT_NEST_LEVEL)] and  
35015. BLOCK_TYPE(CURRENT_NEST_LEVEL) = 'CASE', then  
35016. ERROR_OCCURRED will be set false and CURRENT_NEST_LEVEL will not  
35017. be modified.) */  
35018. if ERROR_OCCURRED  
35019. then  
35020. mexit  
35021. fi  
35022. fi  
35023. call POP_OLD_BLOCK( ; )  
35024. /* Delete current block, generating END and XII labels as required, and  
35025. popping stack. (Lemma: POP_OLD_BLOCK decrements CURRENT_NEST_LEVEL  
35026. by exactly one.) */  
35027. if CURRENT_NEST_LEVEL <= NESTING_LIMIT  
35028. then  
35029. generate (' END ')  
35030. /* Generate branch to end of DOCASE. */  
35031. fi  
35032. fi  
35033. fi  
35034. /* (Lemma: If CURRENT_NEST_LEVEL > 0 and  
35035. [USER_NAME = '' or = BLOCK_NAME(CURRENT_NEST_LEVEL)] and  
35036. BLOCK_TYPE(CURRENT_NEST_LEVEL) = 'CASE' at entry to ESAC, then  
35037. CURRENT_NEST_LEVEL will be decremented by exactly one.) */  

C-56
ESACOD Macro -- 3 July 1973

37001. \texttt{BASE} ESACOD ( ; USER\_NAME)
37002. \texttt{/* Generates final part of DCASE processing: for SIMPLE, CONDTEST,}
37003. \texttt{or SPARSE type DCASE, the EQU for the MISC block (or END of DCASE)
37004. to the last generated branch target is required; for GENERAL type
37005. DCASE, the branch vector and the transfer to any symbolic
37006. compare or MISC block must be generated. Finally, the block is
37007. popped. */}
37008. \texttt{idt}
37009. CASE\_COUNTER, /* Holds number of last case generated. */
37010. T, /* Temporary. */
37011. COMP\_LABEL\_NO, /* Label number of outstanding compare case. */
37012. MAX\_SD\_VALUE, /* Maximum self-defined operand. */
37013. BASE /* Index within CASE\_OCCURS array for CASE 0. */
37014. \texttt{end}
37015. MISC\_FOUND, /* Indicates whether a MISC CASE was found. */
37016. MULTIPLE\_OF\_4, /* Indicates whether all branch-vector operands were
37017. multiples of 4. */
37018. INDEX\_RANGE\_ASSURED, /* true if we have been assured (by
37019. "DCASE *** ONLY") that no values other than those specified
37020. by CASE operands will occur. */
37021. ANY\_COMP\_CASES, /* Indicates whether any "compare" cases were
37022. generated (either CHARCOMP or symbolic general case operands). */
37023. ANY\_SELFDEF\_CASES, /* Indicates whether any "self-defining" cases (to
37024. be handled by branch vector) were generated. */
37025. RANGE\_TEST\_REQD /* Indicates that both branch vector and compare
37026. operands were present. */
37027. \texttt{cha}
37028. CASE\_FORMAT, /* Type of CASEs present: GENERAL, SPARSE,
37029. CHARCOMP, SIMILE, or CONDTEST. */
37030. INDEX\_ADDR, /* Address of DCASE index. */
37031. NOCASE, /* Label for branch vector processing used for unspecified
37032. cases. */
37033. LABEL /* Any outstanding label waiting to be generated. */
37034. \texttt{*/ (Ground rules: No ESACOD proc modifies CURRENT\_NEST\_LEVEL.
37035. This can be shown by referring to the cross-reference index.) */

C-57
"ESACOD" Macro -- 3 July 1973

37018. call TRACE_PRINTER ( ; 'ESACOD' )
37019. /* Print macro name "ESACOD" in mnote if tracing on. */
37020. if CURRENT_NEST_LEVEL = NESTING_LIMIT
37021. then
37022. call VERIFY_END ( ; 'DOCASE', USER_NAME)
37023. /* 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_OCCURED in set. */
37024. [Lemma: If CURRENT_NEST_LEVEL > 0 and USER_NAME = ' or = BLOCK_NAME(CURRENT_NEST_LEVEL) and BLOCK_TYPE = 'DOCASE', then ERROR_OCCURED will be set false and CURRENT_NEST_LEVEL will not be modified.] */
37025. if ERROR_OCCURED
37026. then
37027. exit
37028. fi
37029. call ESACOD_INFO_UNPACK
37030. /* Extracts CASE_FORMAT, CASE_COUNTER, INDEX_ADDR, MISC_FOUND, BLOCK_LABEL_PREFIX, INDEX_RANGE_ASSUGED, and MULTIPLESOF4 from stack. */
37031. if CASE_FORMAT = 'GENERAL'
37032. then
37033. call ESACOD_GENERAL_CASE_CHOICE
37034. /* Generate all code to complete processing of general case. */
37035. else
37036. if CASE_FORMAT = 'SPARSE' or = 'CHARCCMP'
37037. then
37038. T := NEXT_COMP_LABEL_NO(GCASE_NEST_LEVEL)
37039. /* We need to define last compare case target. */
37040. GCASE_NEST_LEVEL := GCASE_NEST_LEVEL - 1
37041. /* Pop GCASE stack. */
37042. else /* CONDTES or SIMPLE. */
37043. T := CASE_COUNTER + 1
37044. /* We need to define last conditional test target. */
37045. fi
37046. if MISC_FOUND
37047. then
37048. generate ( BLOCK_LABEL_PREFIX || 'C' || T || ' EQU ' || BLOCK_LABEL_PREFIX || 'MSC')
37049. else
37050. generate ( BLOCK_LABEL_PREFIX || 'C' || T || ' DS OH')
37051. fi
37052. fi
37053. call POP_OLD_BLOCK ( )
37054. /* (Lemma: POP_OLD_BLOCK decrements CURRENT_NEST_LEVEL by exactly one.) */
37055. end
37056. /* (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

Nacro -- 3 July 1973

37093. PROC ESACOD_INFO_UNPACK
37094. /* Extract the following information from the stack: */
37095. CASE_COUNTER := OPERAND1(CURRENT_NEST_LEVEL)
37096. INDEX_ADDR := OPERAND2(CURRENT_NEST_LEVEL)
37097. CASE_FORMAT := OPERAND3(CURRENT_NEST_LEVEL)
37098. MISC_FOUND := INFORMATION(CURRENT_NEST_LEVEL)[2,1]
37099. MULTIPLESOF4 := INFORMATION(CURRENT_NEST_LEVEL)[3,1]
37100. INDEX_RANGE_ASSURED := INFORMATION(CURRENT_NEST_LEVEL)[5,1]
37101. BLOCK_LABEL_PREFIX := '$' || BLOCK_NUMBER(CURRENT_NEST_LEVEL)
37102. EXIT
"ESACOD" Macro -- 3 July 1973

37105. ELOG ESACOD_GENERAL_CASE_CHOICE
37106. /* Generate all code to complete processing of general case.
37107. Includes the generation of a branch vector, if required. */
37109. call ESACOD_GENERAL_CASE_INFO
37110. /* Pops MAX_SD_VALUE, CCMP_LABEL_NO,
37111. and BASE (of CASE_OCCURS array) out of GCASE stack. */
37112. if ~ ERROR_OCCURRED
37113. then ANY_SELFDEF_CASES
37114. then
37115. call ESACOD_SELFDEF_GEN
37116. /* Handles branch vector-type implementaion for all cases which
37117. contain self-defining terms (of value < 256). Also generates
37118. linkage for any other terms and/or MISC case which were used
37119. with the self-definers. */
37121. else /* No self-definers were present. */
37122. if ANY_COMP_CASES
37123. 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. baists (8, 'SYNC:701: NO CASE CONTAINS NO VALID CASES')
37130. fi
37131. fi
37133. corp
37135. DEFINE ESACOD_GENERAL_SYM_ONLY
37136. /* Generate linkage to process symbolic operands and MISC in the
37137. absence of self-definers (self-defined terms of value < 256). */
37140. generate (BLOCK_LABEL_PREFIX || 'SIG' EQU 'I')
37141. if MISC_FOUND
37142. generate (BLOCK_LABEL_PREFIX || 'C' || COMP_LABEL_NO || 'EQU ' ||
37143. BLOCK_LABEL_PREFIX || 'MISC')
37146. else
37147. generate (BLOCK_LABEL_PREFIX || 'C' || CCM_LABEL_NO || 'DS DS')
37148. fi
37149. core
ESACOD Macro -- 3 July 1973

ESACOD_SELFDEF_GEN

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

LABEL := BLOCK_LABEL_PREFIX || 'BEG'
/* Note that BEG label must be generated on first instruction. */
docase ifany
of case ANY_COMPCASES
NOCASE := BLOCK_LABEL_PREFIX || 'Cl'
esac case MISC
NOCASE := BLOCK_LABEL_PREFIX || 'MSC'
esac case misc
NOCASE := BLOCK_LABEL_PREFIX || 'END'
esac
case misc

RANGE_TEST_REQD := ((INDEX_RANGE_ASSURED) or ANY_COMPCASES)
if RANGE_TEST_REQD then
call ESACOD_OUTOF_RANGE_CHECK
/* Generate check for index out of the range 0 through MAX_SD_VALUE. */
fi
call ESACOD_BRVCT_GEN
/* Generate branch vector and all final constants and equates
required. */
corr
ESACOD Macro -- 3 July 1973

37182.  **ESACOD** ESACOD_GENERAL_CASE_INFO
37183.       /* Pops following information out of GCASE stack. Indicates success
37184.       (or lack thereof) in ERROR_OCCURR. */
37185.       int I
37186.
37187.       I := GCASE_NEST_LEVEL
37188.       if I > GCASE_NEST_LIMIT
37189.       then
37190.         ERROR_OCCURED := true
37191.       else
37192.         MAX_SD_VALUE := MAX_CASE_VALUE(I)
37193.         ANY_SELFDEF_CASES := (MAX_SD_VALUE > 0)
37194.         COMP_LABEL_NO := NEXT_COMP_LABEL_NO(I)
37195.         ANY_COMP_CASES := (COMP_LABEL_NO > 1)
37196.         BASE := ((I-1) * 256) + 1
37197.       fi
37198.  core
37199.  core
37200.  core
ESACOD macro -- 3 July 1973

37202. ESEQ 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 GPB. */
37209. generate (LABEL || " LTR 1,1")
37210. IF CASE_OCCURS(BASE)
37211. THEN /* CASE 0 occurs. */
37212. generate (' BM ' || NOCASE)
37213. ELSE
37214. generate (' BNP ' || NOCASE)
37215. FI
37216. generate (' C 1, ' || BLOCK_LABEL_PREFIX || ' SIZ')
37217. generate (' BB ' || NOCASE)
37218. IF MULTIPLESOF4
37219. THEN
37220. generate (' LA 0,3')
37221. generate (' WS 0,1')
37222. generate (' BNZ ' || NOCASE)
37223. FI
37224. ELSE
37225. generate (LABEL || ' CLI ' || INDEX_ADDR || ', ' || MAX_SD_VALUE)
37226. generate (' BB ' || NOCASE)
37227. IF MULTIPLESOF4
37228. THEN
37229. generate (' TM ' || INDEX_ADDR || ",B'00000011'"")
37230. generate (' BEZ ' || NOCASE)
37231. FI
37232. FI
37233. LABEL := ""
37234. END
C-65

```c
共鸣 SIGN
37236. #SG SIGCOD_BAVCT_GEN
37237. /* Generate branch vector proper. */
37238. int I, INCH
37240. if INDEX_ADDR != ""  
37241. then /* Generate code to put byte index into CPRI. */
37242. generate (LABEL || ' S1,1')
37243. LABEL := ""
37244. generate (' IC 1,1 || INDEX_ADDR)
37245. fi
37246. if MULTIPLESOF4
37247. then  
37248. INC := 0
37249. else
37250. INC := 1
37251. generate (LABEL || ' S1A 1,2')
37252. LABEL := ""
37253. fi
37254. if CASE_OCCURS(BASE) || INDEX_ADDR != ""  
37255. then /* Zero case must be included in branch vector. */
37256. generate (LABEL || ' B **(1)')
37257. I := 0
37258. else
37259. generate (LABEL || ' B *(1)')
37260. I := INC
37261. /* Skip the zero case and start with case 1 (or case 4). */
37262. fi
37263. LABEL := ""
37264. while I < MAX_SD_VALUE
37265. do
37266. if CASE_OCCURS(BASE+I)
37267. then
37268. generate (' B ' || BLOCK_LABEL_PREFIX || 'G' || I)
37269. else
37270. generate (' B ' || NOSCASE)
37271. fi
37272. I := I + INC
37273. od /* (Termination: INC > 0, so I is incremented in loop;  
37274. MAX_SD_VALUE is fixed, therefore I must eventually exceed
37275. DAI_SD_VALUE.) */
37276. if RANGE_TEST_REQS and INDEX_ADDR = ""  
37277. then
37278. generate (BLOCK_LABEL_PREFIX || "SIZ DC F" || MAX_SD_VALUE ||
37279. "")
37280. fi
37281. if ANY_COMP_CASES
37282. then
37283. if NISC_FOUND
37284. then
37285. generate (BLOCK_LABEL_PREFIX || 'C' || COMP_LABEL_NUM || ' EQU' ||
37286. BLOCK_LABEL_PREFIX || 'RSC')
37287. else
37288. generate (BLOCK_LABEL_PREFIX || 'C' || COMP_LABEL_NUM || ' DS DH')
37289. fi
37290. fi
37291. corp
```
```
41001. macro BLOCK (USER_NAME;)
41002. /* Generate simple one-in-one-out block in structure with name
41003. specified. */
41004. char LABEL
41005. /* Contains any outstanding label waiting to be generated. */
41007. call TRACE_PRINTER ( ; 'BLOCK')
41008. /* Prints macro name "BLOCK" in mnote if tracing on. */
41009. call PUSH_NEW_BLOCK (USER_NAME; BLOCK_TYPE_VALUE='BLOCK')
41010. /* Define new block; add to stack. Initialize block specifications.
41011. Note block type and set up a unique BLOCK_LABEL_PREFIX for use in
41012. generating labels. */
41013. if ERROR_OCCURRED
41014. then
41015. mexit
41016. fi
41017. LABEL := USER_NAME
41018. call BLOCK_TRACE_COUNTERS
41019. /* If block counts were requested, generate counters and incrementing
41020. instructions. Any label waiting to be defined is returned in
41021. LABEL. */
41022. if LABEL # ''
41023. then
41024. generate (LABEL | | ' DS OH')
41025. /* Define label if one required and not yet defined. */
41026. fi
41027. fi
```
"BLOCK" Macro -- 15 June 1973

41029. **PROC BLOCK_TRACE_COUNTERS**
41030. /* Generate debugging information required---block name constant
41031. and/or block counters. */
41032. if DEBUG_BLOCKCOUNTERS_REQD or DEBUG_BLOCKNAMES_REQD
41033. then
41034. if DEBUG_BLOCKCOUNTERS_REQD
41035. then
41036. /* Generate block count incrementing instructions. */
41037. generate (LABEL I 'LB 1, ' || BLOCK_LABEL_PREFIX || 'BLC')
41038. LABEL := 10 /* Clear LABEL to show it has been generated. */
41039. generate ('LA 1,1(1)')
41040. generate ('STO 1, ' || BLOCK_LABEL_PREFIX || 'BLC')
41041. fi
41042. generate (LABEL || ' ' || BLOCK_LABEL_PREFIX || 'GO')
41043. /* Establish GO label as requiring definition. */
41044. if DEBUG_BLOCKNAMES_REQD
41045. then
41046. generate ("DC C'\"' || BLOCK_NAME(CURRENT_NEST_LEVEL) || '")
41047. fi
41048. if DEBUG_BLOCKCOUNTERS_REQD
41049. then
41050. generate ("' ' || BLOCK_NAME(CURRENT_NEST_LEVEL) || 'BLC DC H'O' BLOCK COUNT")
41051. fi
41052. fi
41053. fi
41054. fi
41055. fi
41056. fi
41057. qed
## BLEND Macro -- 15 June 1973

```plaintext
43001.  MACRO BLEND ( ; USER_NAME, RETURN=RETURN, LINK=LINK, RESTORE=RESTORE, RC=RC)
43002.  /* The BLEND macro acts as a generic name for IF, DO, ESAC, ESACOD, and
43003.  Proc and as a terminating macro for BLOCK. For any
43004.  of the former, the proper macro is invoked depending on the
43005.  block type being terminated. For BLOCK blocks, the block is
43006.  simply terminated. */
43007.  int I /* Temporary. */
43008.  call TRACE_PRINTER ( ; 'BLEND') /* Prints macro name "BLEND" in mnote if tracing on. */
43009.  if CURRENT_NEST_LEVEL > NESTING_LIMIT then
43010.    call POP_OLD_BLOCK ( ; )
43011.  else
43012.    if CURRENT_NEST_LEVEL = 0 then
43013.      note (6, 'STAC4302 NO BLOCKS ACTIVE--"BLEND" IGNORED')
43014.    else
43015.      I := CURRENT_NEST_LEVEL
43016.      if USER_NAME = '' then
43017.        while I > 0 and BLOCK_NAME(I) # USER_NAME
43018.          do /* (Termination: I is decremented—must eventually become
43019.              0) */
43020.            I := I - 1
43021.          od
43022.      if I = 0 then
43023.        note (6, 'STAC4302 NO BLOCK ACTIVE NAMED ' || USER_NAME ||
43024.            ' "BLEND" IGNORED')
43025.      mexit
43026.    fi
43027.  fi
43028.  do case BLOCK_TYPE(I) only
43029.    of
case 'IF':
43030.      call FI ( ; USER_NAME)
43031.    esac
case 'DO':
43032.      call OD ( ; USER_NAME)
43033.    esac
case 'CASE':
43034.      call ESAC ( ; USER_NAME)
43035.    esac
case 'DOCASE':
43036.      call ESACOD ( ; USER_NAME)
43037.    esac
case 'PROC':
43038.      call CORP ( ; USER_NAME, RETURN=RETURN, LINK=LINK, RESTORE=RESTORE, RC=RC)
43039.    esac
case 'BLOCK':
43040.      call POP_OLD_BLOCK ( ; )
43041.    esac
43042.  esac
43043.  esac
43044.  esac
43045.  esac
43046.  esac
43047.  esac
43048.  esac
43049.  esac
43050.  esac
43051.  esac
43052.  esac
43053.  esac
43054.  esac
43055.  esac
43056.  esac
43057.  fi
43058.  send /* (Lemma: If CURRENT_NEST_LEVEL > 0 and
43059.  [USER_NAME = '' or BLOCK_NAME(CURRENT_NEST_LEVEL)] at entry to
43060.  BLEND, then CURRENT_NEST_LEVEL will be decremented by exactly
43061.  one.) */
```
"FINAL" Macro -- 10 July 1973

53001. macro FINAL ( ; )
53002. /* Insure all blocks are closed. Then if SAVETRACE_ON_FIRST_PROC,
53003. define label $LASTSAV to be PREV_SAVETRACE_AREA and EQUate
53004. PREV_SAVETRACE_PTR to 0. */
53005.
53006. call TRACE_PRINTER ( ; "FINAL")
53007. /* Print macro name "FINAL" in note if tracing on. */
53008. while CURRENT_NEST_LEVEL > 0
53009. do
53010. if CURRENT_NEST_LEVEL > NESTING_LIMIT
53011. then
53012. note (8, 'STSC53C1 BLEND OF OUTSTANDING BLOCK ASSUMED')
53013. else
53014. note (8, 'STSC5301 BLEND OF ' || BLOCK_NAME(CURRENT_NEST_LEVEL) ||
53015. ' ASSUMED')
53016. fi
53017. call BLEND [ ; ]
53018. /* (Lemma: If CURRENT_NEST_LEVEL > 0 and no BLEND operands are
53019. specified, CURRENT_NEST_LEVEL will be decremented by exactly
53020. one.) */
53021. od /* (Termination: CURRENT_NEST_LEVEL decreases monotonically
53022. and therefore must eventually become 0.) */
53023. if SAVETRACE_ON_FIRST_PROC
53024. then
53025. if PREV_SAVETRACE_PTR = '$FIRSTSV'
53026. then /* No non-OS proc occurred; generate dummy area. */
53027. generate ("$LASTSAV = 'FIRSTSV'
53028. generate ("$LASTSAV EQU '$FIRSTSV'
53029. else
53030. generate ("$LASTSAV = 'FIRSTSV'
53031. generate (PREV_SAVETRACE_PTR || ' EQU 0')
53032. fi
53033. fi
53034. fi

*==========================================================================*
"EXIT" Macro -- 10 July 1973

55001. macro EXIT (USER_NAME; EXIT_TARGET)
55002.  /* Find exit point. Generate branch. */
55003.  call TRACE_PRINTER ( ; 'EXIT')
55004.  /* Print macro name "EXIT" in mnote if tracing on. */
55005.  if CURRENT_NEST_LEVEL > NESTING_LIMIT
55006.    then
55007.      exit
55008.    fi
55009.  call EXIT_FIND ( ; EXIT_TARGET)
55010.  /* Set ULTIMATE_BRANCH_LABEL to point to end of block whose name
55011.    is the argument and , or as needed as EXIT label; if no such block,
55012.    issue message and set ERROR_OCCURRED. */
55013.  if ~ ERROR_OCCURRED
55014.    then
55015.      generate (USER_NAME || B || ULTIMATE_BRANCH_LABEL)
55016.    fi
55017.  end
"PROC" Macro -- 5 July 1973

8101. MACRO PROC (USER_NAME:, LINKAGE=, ID=, BASE=, SAVEN=, DEBUG=, XIT=)
8102. /* Defines a procedure block. If LINKAGE=OS is specified, standard
8103. OS 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. FIRST_PROC, /* Indicates whether this is the first PROC macro coded
8110. in this assembly. */
8111. FIRST_VALUE_KNOWN, /* Indicates whether the first SAVE= operand was
8112. a self-defining term (or omitted) or if it was symbolic. */
8113. OS_LINKAGE, /* Indicates whether LINKAGE=(OS, -)
8114. was entered. */
8115. SPECIAL_PREFIX, /* Indicates whether the BLOCK_LABEL_PREFIX was
8116. changed to the special debugging form "$PE". */
8117. USING13, /* Indicates whether the base register is GPR13. */
8118. MULTIBASE, /* Indicates more than one base register was
8119. requested but accons for loading have not yet been generated. */
8120. WORKREG_USED /* Indicates whether the value in WORKREG was
8121. modified and its contents saved in register 0. */
8122. char
8123. COMMA2, MULT, /* Contain "," and "H" respectively if a range of
8124. registers is to be saved, or the null string if a single register
8125. to be saved. Used to generate "STM" or "STH" instructions. */
8126. FIRST, LAST, /* First and last register in range to be saved. */
8127. LABEL, /* Any outstanding label waiting to be generated. */
8128. LOCAL_POINTER, OS_POINTER, /* Instruction segments to generate
8129. store instruction for proper save area. */
8130. PREVIOUS_DEBUG_VECTOR, /* Hold value of debug switches on entry to
8131. PROC macro for restoring on exit from CORP. */
8132. PROC_ID_BYTE, /* Value of hex proc number (PROC_COUNTER in hex)
8133. used in various debugging instructions. */
8134. SAVE_LENGTH, /* Length of save area (in words), except length of
8135. register part only for local PROCs. */
8136. SAVETYPE, /* Type of save area generated: FULL (savetrace), OSSAVE,
8137. NORML, NORMALS, TRUNC, TRUNCERS, or NONE. */
8138. SAVEREG, /* Register (work or base) which is pointing at new
8139. save area before chaining. */
8140. WORKREG /* Register used for setting up linkage, etc. */
8141. int
8142. OFFSET, /* Offset (in words) to either FIRST (if FIRST_VALUE_KNOWN),
8143. or to GPRO within save area. */
8144. OFFSET_TO_GPRO, /* Offset in words to GPRO within save area. */
8145. SAF, SAL /* Register number to go into first register word of
8146. save area; this, for example, could be 14 even though FIRST
8147. is a symbolic register of unknown value at macro expansion time.
8148. SAL is similar but for last register. */
call TRACE_PRINTER ( ; "PROC")
call PUSH_NEW_BLOCK(USER_NAME; BLOCK_TYPE_VALUE='PROC')
/* Define new block; add to stack. Initialize block specifications.*/
Note block type and set up unique BLOCK_LABEL_PREFIX for use in generating unique labels. */
if ERROR_OCCURRED /* during PUSH_NEW_BLOCK (viz., stack overflow) */
then
mexit
fi
LABEL := USER_NAME
/* Generate PROC's name at first opportunity. */
call PROC_SCAN_OPTIONS
/* Validate LINKAGE and WORK= keywords; issue error messages and set OS_LINKAGE and WORKARE. Process completely DEBUG and EXIT keywords.
Change BLOCK_LABEL_PREFIX if necessary to special PROC form (indicating change in SPECIAL_PREFIX) and set value of FIRST_PROC. */
call PROC_HEADER
/* Generate "CSECT" and "USING *,15" if required. Handle in-line ID (a la IBM SAVE macro). */
call PROC_REG_SAVE
/* Set SAVE_TYPE and SAVE_LENGTH to indicate type of save area required. Save contents of general purpose registers, if required. */
call PROC_ENSUREBASE
/* Set up base register where required and issue USING. Set USING13 if base register to be loaded into 13 was put temporarily into WORKREG. If multiple base registers, set MULTIBASE */
call PROC_GEN_SAVEAREA
/* Generate proper save area depending on the variables SAVE_TYPE and SAVE_LENGTH set by PROC_REG_SAVE and depending on the SAVE macro operands */
call PROC_MULTIBASE_GEN
/* Generate definition of adconds for multiple base registers. */
call PROC_DEBUG_STUFF
/* Generate trace and count code for debugging, if requested. */
if WORK = 'SOME' and WORKREG USED
then
generate (LABEL | | LS | | WORKREG | | '0', 0)
fi
label := ''
if LABEL = ''
then
generate (LABEL | | DS | | OS, 0')
fi
if LINKAGE(2) = 'CSECT'
then
generate (' DROP 15')
endif for USING generated by PROC_HEADER. */
call PROC_INFO_SAVE
/* Save any information necessary to generate CORP macro. */
}

*C-72*
PROC SCAN OPTIONS

PROC_macro coded in this assembly. */

OS LINKAGE := (LINKAGE(1) ≠ 'OS' AND ≠ '

if LINKAGE(2) ≠ 'CSECT' AND ≠ '"

then
  mnote (6, 'STC8102 SECOND LINKAGE OPERAND IGNRED')

WORKREG := WORK,

If WORKREG = 'NONE' OR = '

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

  if OS_LINKAGE
    then
      WORKREG := '2';
      else
        WORKREG := '1';
        fi
  fi

EXIT_SEVERITY := EXIT

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

NOT_FIRST_PROC := true

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

call PROC_DEBUG_SET

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

SPECIAL_PREFIX := (DEBUG_SAVETRACE_REQD OR DEBUG_PROCTRACE_REQD OR DEBUG_PROCCOUNTS_REQD) AND PROC_COUNTER < 254

if SPECIAL_PREFIX
  then /* We want to label this proc with a hex proc number. */
    PROC_COUNTER := PROC_COUNTER + 1
    heX_IN := PROC_COUNTER
    call HEx ( ""
    PROC_ID_BYTE := heX_OUT
    BLOCKLABELPREFIX := "$P0" || PROC_ID_BYTE
    */ Change labels on PROC to ease understanding of debug information. */
    BLOCK_PREFIX := "#" || BLOCK_NAME(CURRENT_NEST_LEVEL) || ", " || DEBUG_ID="" || PROC_ID_BYTE || ""
    else
      PROC_ID_BYTE := '00'
      fi
  fi

/* Validate LINKAGE= and WORK= keywords; issue error messages and set OS LINKAGE = WORKREG (the latter receiving either the register specified by the WORK= operand or some default). Process completely necessary from the normal "$P0" to the special PROC prefix */

Set FIRST_PROC to indicate whether this is the first PROC macro coded in this assembly. */

if LINKAGE(1) ≠ 'OS' AND ≠ '"

then
  mnote (6, 'STC8101 LINKAGE="" || LINKAGE(1) || "" INVALID--"OS" ASSUMED')
  fi

if LINKAGE(2) ≠ 'CSECT' AND ≠ '"

then
  mnote (6, 'STC8102 SECOND LINKAGE OPERAND IGNRED')

WORKREG := WORK,

If WORKREG = 'NONE' OR = '

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

  if OS_LINKAGE
    then
      WORKREG := '2';
      else
        WORKREG := '1';
        fi
  fi

EXIT_SEVERITY := EXIT

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

NOT_FIRST_PROC := true

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

call PROC_DEBUG_SET

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

SPECIAL_PREFIX := (DEBUG_SAVETRACE_REQD OR DEBUG_PROCTRACE_REQD OR DEBUG_PROCCOUNTS_REQD) AND PROC_COUNTER < 254

if SPECIAL_PREFIX
  then /* We want to label this proc with a hex proc number. */
    PROC_COUNTER := PROC_COUNTER + 1
    heX_IN := PROC_COUNTER
    call HEx ( ""
    PROC_ID_BYTE := heX_OUT
    BLOCKLABELPREFIX := "$P0" || PROC_ID_BYTE
    */ Change labels on PROC to ease understanding of debug information. */
    BLOCK_PREFIX := "#" || BLOCK_NAME(CURRENT_NEST_LEVEL) || ", " || DEBUG_ID="" || PROC_ID_BYTE || ""
    else
      PROC_ID_BYTE := '00'
      fi
  fi
PROC Macro -- 5 July 1973

81167. PROC PROC_DEBUG_SET
81168. /* Save the previous DEBUG specifications so they can be restored at
81169. COMP time. Scan the DEBUG= suboperands setting the debug flags
81170. indicated. If GLOBAL is specified send null restore value to
81171. suppress it. */
81172. bit 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. SAVETRACE_CHECK /* 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. SAVETRACE_VALUE := false /* Save current value of debug switches. */
81181. previous_debug_vector :=
81182. DEBUG_BLOCKNAMES_REQD
81183. DEBUG_PROCNAMES_REQD
81184. DEBUG_LISTBLOCKS_REQD
81185. DEBUG_BLOCKCOUNTS_REQD
81186. DEBUG_PROCCOUNTS_REQD
81187. DEBUG_CORPVALUES_REQD
81188. DEBUG_SAVETRACE_REQD
81167. DEBUG_BLOCKCOUNTS_REQD
81168. DEBUG_PROCCOUNTS_REQD
81169. DEBUG_PROCTRACE_REQD
81170. DEBUG_CORPVALUES_REQD
81171. DEBUG_SAVETRACE_REQD
81182. DEBUG Blocknames_REQD := true
81183. DEBUG_PROCNAMES_REQD := true
81184. DEBUG_LISTBLOCKS_REQD := true
81185. DEBUG_BLOCKCOUNTS_REQD := true
81186. DEBUG_PROCCOUNTS_REQD := true
81187. DEBUG_CORPVALUES_REQD := true
81188. DEBUG_SAVETRACE_REQD := true
81167. DEBUG_BLOCKCOUNTS_REQD
81168. DEBUG_PROCCOUNTS_REQD
81169. DEBUG_PROCTRACE_REQD
81170. DEBUG_CORPVALUES_REQD
81171. DEBUG_SAVETRACE_REQD
81172. /* Scan all operands. */
81173. while I ≤ DEBUG
81174. do
81175. case DEBUG(I)
81176. of
81177. ("GLOBAL", "GBL")
81178. GLOBAL := true
81179. SAVETRACE_VALUE := false
81180. case ("BLOCKNAMES", "BN")
81181. DEBUG_BLOCKNAMES_REQD := true
81182. /* BLOCKNAMES causes the name of each block to be generated as an
81183. in-line character constant at the start of each block (of any type,
81184. not just BLOCK macros) for ease of locating code in dumps. */
81185. SAVETRACE_VALUE := false
81186. case ("NOBLOCKNAMES", "NBN")
81187. DEBUG_BLOCKNAMES_REQD := false
81188. esac
81189. case ("PROCNAMES", "PN")
81190. DEBUG_PROCNAMES_REQD := true
81191. /* PROCNAMES causes the name of each PROC to be generated as an
81192. in-line character constant at the start of the PROC for ease of
81193. locating code in dumps. */
81194. SAVETRACE_VALUE := false
81195. case ("NOPROCNAMES", "NPN")
81196. DEBUG_PROCNAMES_REQD := false
81197. esac
81198. case ("LISTBLOCKS", "LB")
81199. DEBUG_LISTBLOCKS_REQD := true
81200. /* LISTBLOCKS causes the name, number, and depth of each block to be
81201. generated in an mnote at the start and end of the block. */
81202. SAVETRACE_VALUE := false
81203. case ("NOLISTBLOCKS", "NLB")
81204. DEBUG_LISTBLOCKS_REQD := false
81205. esac
81206. case ("BLOCKCOUNTS", "BC")
81207. DEBUG_BLOCKCOUNTS_REQD := true
81208. /* BLOCKCOUNTS causes counters to be kept on the number of executions
81209. of all blocks. */
81210. SAVETRACE_VALUE := false
81211. case ("NOBLOCKCOUNTS", "NBC")
81212. DEBUG_BLOCKCOUNTS_REQD := false
81213. esac
81214. case ("PROCCOUNTS", "PC")
81215. DEBUG_PROCCOUNTS_REQD := true
81216. /* PROCCOUNTS causes counters to be kept on the number of executions
81217. of all PROC blocks. */
81218. SAVETRACE_VALUE := false
81219. case ("NOPROCCOUNTS", "NPC")
81220. DEBUG_PROCCOUNTS_REQD := false
81221. esac
81222. esac
81223. esac
81224. esac
81225. DEBUG_LISTBLOCKS_REQD := false
81226. esac
81227. case ("BLOCKCOUNTS", "BC")
81228. DEBUG_BLOCKCOUNTS_REQD := true
81229. /* BLOCKCOUNTS causes counters to be kept on the number of executions
81230. of all blocks. */
81231. SAVETRACE_VALUE := false
81232. case ("NOBLOCKCOUNTS", "NBC")
81233. DEBUG_BLOCKCOUNTS_REQD := false
81234. esac
81235. case ("PROCCOUNTS", "PC")
81236. DEBUG_PROCCOUNTS_REQD := true
81237. /* PROCCOUNTS causes counters to be kept on the number of executions
81238. of all PROC blocks. */
81239. SAVETRACE_VALUE := false
81240. case ("NOPROCCOUNTS", "NPC")
81241. DEBUG_PROCCOUNTS_REQD := false
81242. esac

C-74
CASE ('PROCTRACE', 'PT')
  DEBUG_PROCTRACE_REQD := true
  /* PROCTRACE causes a trace vector to be generated and instructions to
   move the hex PROC number into the vector to show the order of PROC
   calls. */
CASE ('NOPROCTRACE', 'NPT')
  DEBUG_PROCTRACE_REQD := false
CASE ('CORPVALUES', 'CV')
  DEBUG_CORPVALUES_REQD := true
  /* CORPVALUES causes the value of the registers at CORP time (before
   restoring those saved at PROC entry) to be stored into an area for
   reference. */
CASE ('NOCORPVALUES', 'NCV')
  DEBUG_CORPVALUES_REQD := false
CASE ('SAVETRACE', 'ST')
  /* SAVETRACE causes all save areas for non-OS register saving to be
   linked into a static chain from the OS save area to provide
   formatting in the save area trace portion of ABEND/SNAP dumps. */
  SAVETRACE_VALUE := true
  SAVETRACE_CHECK := true
  /* Note that savetrace has been specified. */
CASE ('NOSAVETRACE', 'NST')
  SAVETRACE_VALUE := false
  SAVETRACE_CHECK := true
CASE ('ALL', 'NONE')
  DEBUG_BLOCKNAMES_REQD, DEBUG_PROCNAMES_REQD, DEBUG_LISTBLOCKS_REQD,
  DEBUG_BLOCKCOUNTS_REQD, DEBUG_PROCCOUNTS_REQD, DEBUG_PROCTRACE_REQD,
  DEBUG_CORPVALUES_REQD, SAVETRACE_VALUE := (DEBUG(I) = 'ALL')
  /* Set (or reset) all main debug switches. */
  SAVETRACE_CHECK := true
CASE ('MACRONAMES', 'MN')
  DEBUG_MACRONAMES_REQD := true
  /* MACRONAMES causes the name of each structured macro (including inner
   macros) to be printed in an mnote whenever invoked. */
CASE ('NOMACRONAMES', 'NMN')
  DEBUG_MACRONAMES_REQD := false
CASE ('DEBUGMACROS', 'DM')
  DEBUG_DEBUGMACROS_REQD := true
  /* DEBUGMACROS causes various intermediate values within the macros
   to be printed in notes for use in debugging the macros. */
CASE ('NODEBUGMACROS', 'NDM')
  DEBUG_DEBUGMACROS_REQD := false
CASE misc
  mnote (8, 'STRC8104 DEBUG=' (I DEBUG(I) 'INVALID-IGNORED')
esacod
I := I + 1 /* Go on to next suboperand. */
ed /* (Termination: I is incremented, N'DEBUG is fixed in loop;
I must eventually exceed N'DEBUG. */
81310. if SAVETRACE_CHECK
81311.   then
81312.     if SAVETRACE_VALUE
81313.       if FIRST_PROC
81314.         then
81315.           if OS_LINKAGE
81316.             then
81317.               SAVETRACE_ON_FIRST_PROC := TRUE
81318.               DEBUG_SAVETRACE_REQD := TRUE
81319.               note (4, 'STBC8103 WARNING-SAVETRACE REQUIRES "FINAL" MACRO')
81320.             else
81321.               note (8, 'STBC8106 SAVETRACE REQUIRES FIRST PROC TO BE LINKAGE=OS')
81322.           else
81323.             /* Not first PROC. */
81324.           if SAVETRACE_CN_FIRST_PROC
81325.             then /* SAVETRACE is being resumed. */
81326.               DEBUG_SAVETRACE_REQD := TRUE
81327.             else
81328.               note (8, 'STBC8105 SAVETRACE MUST BE SPECIFIED ON FIRST PROC')
81329.           else
81330.             fi
81331.           else
81332.             DEBUG_SAVETRACE_REQD := FALSE
81333.             fi
81334.           fi
81335.       fi
81336.     else /* Not first PROC. */
81337.       if GLOBAL
81338.         then
81339.           PENDING_DEBUG_VECTOR := ""
81340.         fi
81341.     fi
81342. fi
81343. corp
PROC Macro

/* Generated a "CSECT" and "USING *,15" if LINKAGE=(***,CSECT)
specified. If LINKAGE=(OS,*** specified, generate inline ID
similar to ID SAVES macro; for non-OS linkage, do the same if
USAGE(*,PRIVATE) or ID specified, but omit ID-length-count field.
Return any label generated as branch target in LABEL. */

char SECT,/* CSECT name (or "$PRIVATE") for default TID constant.
*/

TARGET /* Temporary. */

LENSTd := ((LENGTH/2)*2) + 1 /* Round up to odd number. */
generate (' DC ALt (' || LENGTH || ')','CL' || LENGTH ||
QUOTE || ID || QUOTE)

else
LEa3TD := ((K'ID + 1)/2)*2 /* Round up to even number. */
generate (' DC CL' || LENGTH || QUOTE || ID || QUOTE)

fi

fi

fi

if OS_LINKAGE
if LINKAGE=(**,

81363. PROC HEADER
81364. /* Generates a "CSECT" and "USING *,15" if LINKAGE=(***,CSECT)
81365. specified. If LINKAGE=(OS,*** specified, generate inline ID
81366. similar to ID SAVES macro; for non-OS linkage, do the same if
81367. USAGE(*,PRIVATE) or ID specified, but omit ID-length-count field.
81368. Return any label generated as branch target in LABEL. */
81369. sect, /* CSECT name (or "$PRIVATE") for default TID constant.
81370. */
81371. target /* Temporary. */
81372. lenSTd /* Length of character identifier. */
81373. if LINKAGE(2) = 'CSECT'
81374. then
81375. gen7erate (LABEL || ' CSECT')
81376. label := ''
81377. target := ''
81378. else
81379. target := 'PRIVATE' /* Default name if none specified. */
81380. fi
81381. if OS_LINKAGE or ID = 'NONE' or
81382. osprograms_mode
81383. then
81384. target := block_label_prefix || 'AA'
81385. generate (LABEL || 3 'TARGET)
81386. label := target
81387. else
81388. target := '
81389. if ID = '' or ID = 'NONE'
81390. then /* An ID was specified on PROC macro. */
81391. target := user_name
81392. fi
81393. if OS_LINKAGE
81394. lenSTd := 8
81395. else
81396. secT := user_name
81397. length := 8
81398. fi
81399. if OS_LINKAGE
81400. then
81401. generate (' DC ALt (' || LENGTH || ')','CL' || LENGTH ||
81402. QUOTE || ID || QUOTE)
81403. else
81404. length := ((K'ID + 1)/2)*2 /* Round up to even number. */
81405. generate (' DC CL' || LENGTH || QUOTE || ID || QUOTE)
81406. fi
81407. fi
81408. else
81409. target := '
81410. generate (' DC CL' || LENGTH || QUOTE || ID || QUOTE)
81411. fi
81412. fi
81413. fi
81414. code

C-77
PROC  MACRO  5 July 1973

01446. #PROC_PROC_RES_SAVS
01447. /* If SAVE_TRACE and doing SAVE_TRACE generate store of all registers.
01448. If SAVE_TRACE and no SAVE_TRACE, do nothing. For other than
01449. SAVE_TRACE, extract from SAVE_TRACE register list to be saved.
01450. Save'use, type and size of save area and put into SAVE_TYPE and
01451. SAVE_LENGTH. Generate instruction to store registers. */
01452. #IF 1 /* Temporary. */

01453. FIRST, LAST := 
01454. OFFSET, SAF, SAL, OFFSET_TO_GPRO := 0
01455. /* Initialize save request information variables. */
01456. if SAVE = 'NOED' then
01457. if DEBUG_WRITE_TRACE := 0
01458. then
01459. SAVETYPE := 'FULL'
01460. SAVE_LENGTH := 15
01461. LABEL := 'STM 14,12,' || BLOCK_LABEL_PREFIX || 'SV+12')
01462. LABEL := **
01463. else
01464. call PROC_GET_SAVE_INFO
01465. /* Collect following save request information. Put character string
01466. name of first register to be saved in FIRST, last in LAST, 'N' into
01467. MULT, and ' ' into COMM2; if only a single register to be saved,
01468. MULT, and COMM2 get null strings. Register number which could
01469. go into first word of register part of save area goes into SAF, last
01470. into SAL. (These, for example, could be 14 and 12 while FIRST and
01471. LAST are symbolic register designations of unknown value at macro
01472. expansion time.) Set FIRST_VALUE_KNOWN if FIRST is not symbolic. */
01473. call PROC_GET_SAVE_TYPE
01474. /* Put type of save area to be generated into SAVE_TYPE. Set
01475. SAVE_LENGTH with length (in words) of save area (except only length
01476. of register part for non-OS_LINKAGE areas). Offset in save area
01477. (in words) of register 0 is put into OFFSET_TO_GPRO. Offset of
01478. offset FIRST (if FIRST_VALUE_KNOWN) or else of register 0 is
01479. put into OFFSET. also set LOCAL_POINTER to ' ' and OS_POINTER to
01480. '13)' if OS_LINKAGE, else LOCAL_POINTER to
01481. (BLOCK_LABEL_PREFIX || 'SV+') and OS_POINTER to ' '. Thus
01482. LOCAL_POINTER || OFFSET || 'A' || OS_POINTER refers to the given
01483. offset in the proper area. */
01484. if SAVE_TYPE = 'FULL' then
01485. asemble (LABEL || ' STM 14,12,' || BLOCK_LABEL_PREFIX || 'SV+12')
01486. /* Save all registers on FULL (savetrace-required) save area. */
01487. else
01488. if FIRST_VALUE_KNOWN then
01489. i := OFFSET + 4 /* calculate offset in bytes. */
01490. assemble (LABEL || ' ST' || MULT || ' ' || FIRST || ' ' ||
01491. LAST || COMM2 || LOCAL_POINTER || ' ' || OS_POINTER)
01492. else
01493. assemble (LABEL || ' ST' || MULT || ' ' || FIRST || ' ' ||
01494. LAST || COMM2 || LOCAL_POINTER || ' ' || FIRST || ' ' ||
01495. OFFSET ||
01496. fi
01497. fi
01498. LABEL := **
01499. fi
01500. fi
PROC_SET_SAVE_INFO

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

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

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

/* Set SAVE_TYPE with type of save area to be generated: NONE, OSSAVE, TRUNC, TRUNCADR, NORM, NORMHDR, or FULL. Set SAVE_LENGTH with length (in words) of save area (except only length of register part for non-OS LINKAGE areas). Offset in save area (in words) of either FIRST (if FIRST_VALUEKNOWN) or else of register 0 is put into OFFSET; the latter always is stored into OFFSET_TO_GPRO for CORP's reference. Also set LOCAL_POINTER to '1' and OS_POINTER to '(13) if OS_LINKAGE, else LOCAL_POINTER to (BLOCK_LABEL_PREFIX || 'SV') and OS_POINTER to '1'. */

int I /* Temporary. */

if OS_LINKAGE
    then
        OS_POINTER := '(13)'
        LOCAL_POINTER := '1'
        if SAVE(3) = 'NONE'
            then
                SAVE_TYPE := 'NONE'
            else
                SAVE_TYPE := 'TRUNCADR'
        end
        OFFSET_TO_GPRO, OFFSET := 5 /* Put offset to reg 0 in both. */
        if SAVE(4) = ''
            then
                SAV_LENGTH := '18' /* Standard OS save area is 18 words. */
            else
                SAV_LENGTH := SAVE(4) /* Length specified. */
        fi
    else
        /* Not OS LINKAGE. */
        OS_POINTER := '-'
        LOCAL_POINTER := BLOCK_LABEL_PREFIX || 'SV'
        if DEBUG_PROCCOUNTS_REQD
            then
                SAVE_TYPE := 'FULL' /* Full 18 word pseudo-OS save area. */
                SAV_LENGTH := '15'
            else
                i := SAL - ((SAL+2)/16*16) - SAF + ((SAP+2)/16*16) + 1
                SAVE_LENGTH := i /* Convert calculated length to character string. */
                if SAF = 14
                    then
                        /* SAVE(1) was omitted, specified as 14, or symbolic. */
                end
                if DEBUG_PROCCOUNTS_REQD
                    then
                        SAVE_TYPE := 'NORMHDR' /* First register word is 14. */
                        OFFSET_TO_GPRO, OFFSET := 3
                    else
                        SAVE_TYPE := 'NORM'
                        OFFSET_TO_GPRO, OFFSET := 2
                    end
                else
                    /* Save area is to start after 14: a truncated area. */
                    if DEBUG_PROCCOUNTS_REQD
                        then
                            SAVE_TYPE := 'TRUNCHDR'
                            OFFSET := 1 /* To SAVE(1). */
                        else
                            SAVE_TYPE := 'TRUNC'
                            OFFSET := 0 /* To SAVE(1). */
                        fi
                    if SAF > 13
                        then
                            /* First VALUEKNOWN */
                            /* SAVE(4) was not symbolic */
                            if FIRST_VALUEKNOWN
                                then
                                    if SAF > 13
                                        then
                                            OFFSET_TO_GPRO := OFFSET_TO_GPRO + 16
                                        else
                                            fi
                                        fi
                                    else
                                        /* Adjust OFFSET from giving offset to GPRO to give offset to SAP. */
                                        if SAF > 13
                                            then
                                                OFFSET := OFFSET_TO_GPRO + SAF
                                            else
                                                OFFSET := OFFSET_TO_GPRO
                                            fi
                                        else
                                            /* Offset to GPRO. */
                                        fi
                                    fi
                                else
                                    /* Set offset to GPRO to give offset to SAP. */
                                    if SAF > 13
                                        then
                                            OFFSET := OFFSET_TO_GPRO + SAF
                                        else
                                            OFFSET := OFFSET_TO_GPRO
                                        fi
                                    else
                                        /* Offset to GPRO. */
                                    fi
                                fi
                            else
                                /* Set offset to GPRO to give offset to SAP. */
                                if SAF > 13
                                    then
                                        OFFSET := OFFSET_TO_GPRO + SAF
                                    else
                                        OFFSET := OFFSET_TO_GPRO
                                    fi
                                else
                                    /* Offset to GPRO. */
                                fi
                            fi
                        else
                            /* Set offset to GPRO to give offset to SAP. */
                            if SAF > 13
                                then
                                    OFFSET := OFFSET_TO_GPRO + SAF
                                else
                                    OFFSET := OFFSET_TO_GPRO
                                fi
                            else
                                /* Offset to GPRO. */
                            fi
                        fi
                    else
                        /* Offset to GPRO. */
                    fi
                fi
            else
                /* Offset to GPRO. */
            fi
        fi
    fi

end
PROC PROC_ESTABLISH_BASE

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

bit INLIN_5AVSA
/* Indicates whether an inline save area is to be generated. */

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

int I, J /* Temporary. */
PROC Macro -- 5 July 1973

81621. if BASE # 'NONE' and (OS_LINKAGE or BASE # '')
81622. then /* Generate a base register. */
81623. BAS3REG := BASE(1)
81624. INLIN_SAVEAREA := (SAVE # 'NONE' and SAVE(3) = '13') and -(INLINE_SAVEAREA and OS_LINKAGE)
81625. if BASEREG # '13' and (INLINE_SAVEAREA and OS_LINKAGE)
81626. then
81627. C8109 REGISTER 13 INVALID-IGNORED'
81628. BASEREG := '12'
81629. fi
81630. if BASEREG = ''
81631. then /* Generate a base register specified. */
81632. if INLINE_SAVEAREA and OS_LINKAGE
81633. then
81634. BASEREG := WORKREG
81635. /* We will load the base value first into the work register, then copy the value to register 13 after we finish all linkage. */
81636. JLI3 :=true
81637. J := I + 1
81638. fi
81639. if OS_LINKAGE
81640. then
81641. GENERATE (LABEL II 'LR 0', II WORKREG)
81642. LABEL := 81643. if OSK = 'NONE' and WORKREG_USED
81644. then
81645. BASEREG := true
81646. else
81647. BASEREG := '12'
81648. fi
81649. fi
81650. while I <= N'BASE
81651. do
81652. if BASE(I) = '13'
81653. then
81654. LABEL := (3, 'STRC9109 REGISTER 13 INVALID-IGNORED')
81655. LABEL := BASE(I) # '13'
81656. LABEL := BASEREG # '13'
81657. LABEL := BASE(I) # '13'
81658. fi
81659. fi
81660. I := I + 1
81661. J := J + 4096
81662. end /* (Termination: I is incremented in loop, N'BASE is fixed; I must eventually exceed N'BASE.*/
81663. if INLINE_SAVEAREA
81664. then
81665. if OS_LINKAGE
81666. then
81667. GENERATE ('81668. /* Advance to fullword boundary; outstanding label can wait for next instruction. */
81669. fi
81670. TARGET := BLOCK_LABEL_PREFIX II 'MBLOCK EQU.'
81671. GENERATE (LABEL II 'BAL' II BASEREG II ', ' II TARGET)
81672. LABEL := TARGET
81673. BASE := '0'
81674. GENERATE (LABEL II 'BALR' II BASEREG II ',0')
81675. LABEL := '
81676. fi
81677. if J >= 75191
81678. then
81679. GENERATE ('USING *, II BASEREG)
81680. fi
81681. if N'BASE > 1
81682. then
81683. GENERATE (BLOCK_LABEL_PREFIX II 'MBLOCK EQU.'
81684. MULTIBASE := 'BMP
81685. J := 4096
81686. I := 2
81687. while I <= N'BASE
81688. do
81689. GENERATE ('USING ** II J II ', II BASE(I))
81690. fi
81691. endif
81692. fi
81693. endif
81694. fi
81695. fi
81696. C8109
PROC GEN SAVEAREA

/* Generate appropriate save area according to SAVE_TYPE, SAVE_LENGTH, and the SAVE suboperands. */

if SAVE_TYPE = 'OSSAVE'
then
    call PROC_32N_OSSAVE_AREA
    /* Generate OS save area and chain it up following OS linkage conventions. Also link up static chain of local save areas if this is the first proc and SAVETRACE requested. */
else
    if (SAVE # 'NONE' and SAVE_TYPE # 'NONE') or
    .EBJG_SAVETRACE_REQD
    then
        call PROC_32N_LOCAL_SAVEAREA
        /* Generate local PROC save area according to SAVE_TYPE and SAVE_LENGTH and, if SAVETRACE requested, provide static save area chaining. */
    fi
fi

C-83
PROC GEN OSSAVE AREA

/* Generate OS save area and chain it up following OS linkage
   conventions. Also link up static chain of local save areas
   if DEBUG_SAVETRACE_REQUIRED and FIRST_PROC */

if DEBUG_SAVETRACE_REQUIRED and FIRST_PROC
then
  /* Static chain of local save areas must be linked to OS save areas. */
  generate (LABEL 'ST ' II FIRSTSV+'')
  LABEL := '

  generate (' RVC 8(4,13),='(FIRSTSV)+'
  generate (' L := 13,'(LASTSAV)+')
  PREV_SAVETRACE_PTR := 'FIRSTSV'
  PREV_SAVETRACE_AREA := '0'

fi

if DEBUG_PROCTRACE_REQUIRED and FIRST_PROC and SAVE(3) *
then
  /* Static pointer to PROC trace vector in word 1 of OS save area. */
  if WORK = 'JONE' and WORKREG_USED
then
    generate (' LR O,WORKREG')

fi

if DEBUG_PROCTRACE_REQUIRED and FIRST_PROC and SAVE(3) *
then
  /* Static pointer to PROC trace vector in word 1 of OS save area. */
  if WORK = 'JONE' and WORKREG_USED
then
    generate (' LR O,WORKREG')

fi

end.
PROC_DEFINE

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

char X /* Temporary. */

if SAVE(3) = 'JlNAM'
then
    generate (LABEL II ' LA 0,' II SAVE_LENGTH II ')*4')
    LABEL := 

    if WORK = 'NONE'
    then
        generate (' LR 0,' II WORKREG)
        generate (') LR 0,' II WORKREG)
    else
        generate (') LR 0,' II WORKREG)
    fi

    if LAST # '12'
    then
        BLOCK_LABEL_PREFIX := 'ISV'
    else
        BLOCK_LABEL_PREFIX := 'CC'
    fi

    if FIRST # '14'
    then
        SAVEREG := BASEREG
    fi

    if DBJROKETRACE_REQD and FIRST_PROC
    then
        generate (X II ' DC A (TRACE),(' II SAVE_LENGTH II ')*4')
        /* Generate inline save area with first word pointing to trace vector. */
    else
        generate (X II ' DC (' II SAVE_LENGTH II ')*4')

        if WORK = 'NONE' and WORKREG_USED
        then
            WORKREG_USED := true
        else
            WORKREG_USED := false
        fi

        if DBJROKETRACE_REQD and FIRST_PROC
        then
            generate (LABEL II ' LA 0,' II WORKREG)
            generate (LABEL II ' LA 0,' II WORKREG)
        else
            generate (LABEL II ' LA 0,' II WORKREG)
            generate (LABEL II ' LA 0,' II WORKREG)
        fi

    fi

else /* Jaer-supplied out-of-line save area. */

    if WORK = 'NONE' and WORKREG_USED
    then
        generate (LABEL II ' LA 0,' II WORKREG)
    else
        generate (LABEL II ' LA 0,' II WORKREG)
    fi

    fi

else /* Area has been generated and address is in SAVEREG register. */


PROC Macro -- 5 July 1973

81833. PROC LOCAL_SAVEAREA
81834.      /* Generate local PROC save area according to SAVE_TYPE and SAVE_LENGTH
81835.      and, if SAVETRACE requested, provide static save area chaining. */
81836. char FWD_PTR /* Alias used for next save area. */
81838.  if LABEL = ' ' then
81840.      LABEL := BLOCK_LABEL_PREFIX || 'DD'
81841.     generate (' B ' || LABEL)
81842.  fi
81843.  generate (BLOCK_LABEL_PREFIX || 'SV DS 0F')
81845.  if SAVE_TYPE = 'FULL' or SAVE_TYPE[5,3] = 'HDR'
81846.      then /* Word one should contain PROC count and ID byte. */
81847.      generate ('
81848.      ' FLAG (FR=ENTERED, FR=FINISHED), ID, COUNT')
81849.  fi
81850.  if SAVE_TYPE = 'FULL'
81851.    then
81852.      FWD_PTR := BLOCK_LABEL_PREFIX || 'NXT'
81853.      generate (' DC A' || PREV_SAVETRACE_PTR || ',')
81854.      /* Save label used as forward pointer. */
81855.      PREV_SAVETRACE_PTR := FWD_PTR
81856.      /* Save label used as forward pointer. */
81859.      /* Save save of this save area. */
81860.  fi
81861.  generate ('
81862.  call PROC_MDLFRAMES2_SW
81863.  C86C
<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>8165.</td>
<td><code>.PROC DEBUG_STAFF</code></td>
</tr>
<tr>
<td>8166.</td>
<td><code>/* Generate trace and count code for debugging. */</code></td>
</tr>
<tr>
<td>8167.</td>
<td><code>bit PCT_DEBUGGED_WITH_VECTOR</code> <code>/* Indicates whether -PCT labeled halfword</code></td>
</tr>
<tr>
<td>8168.</td>
<td><code>which holds PROC counter was generated following the trace vector. */</code></td>
</tr>
<tr>
<td>8169.</td>
<td><code>COUNT_SPOT</code> <code>/* Suffix of label for PROC counter. */</code></td>
</tr>
<tr>
<td>8171.</td>
<td><code>if DEBUG_PROCTRACE_REQD</code></td>
</tr>
<tr>
<td>8172.</td>
<td><code>then</code></td>
</tr>
<tr>
<td>8173.</td>
<td><code>if TRACE_GENRED</code></td>
</tr>
<tr>
<td>8174.</td>
<td><code>then</code> <code>/* Previously generated TRACE vector must be updated. */</code></td>
</tr>
<tr>
<td>8175.</td>
<td><code>if WORK = 'NONE' and WORK_REG_USED</code></td>
</tr>
<tr>
<td>8176.</td>
<td><code>then</code></td>
</tr>
<tr>
<td>8177.</td>
<td>`generate (LABEL</td>
</tr>
<tr>
<td>8178.</td>
<td><code>LABEL :='</code></td>
</tr>
<tr>
<td>8179.</td>
<td><code>if</code></td>
</tr>
<tr>
<td>8180.</td>
<td>`generate (LABEL</td>
</tr>
<tr>
<td>8181.</td>
<td>`generate (WORK_REG</td>
</tr>
<tr>
<td>8182.</td>
<td>`generate (' PCT_GENNED</td>
</tr>
<tr>
<td>8183.</td>
<td><code>WORK_REG_USED := true</code></td>
</tr>
<tr>
<td>8184.</td>
<td><code>else</code> <code>/* Trace vector must be generated. */</code></td>
</tr>
<tr>
<td>8185.</td>
<td><code>if TRACE_VEC_GENRED</code></td>
</tr>
<tr>
<td>8186.</td>
<td><code>then</code> <code>/* Branch around trace vector. */</code></td>
</tr>
<tr>
<td>8187.</td>
<td>`LABEL := BLOCK_LABEL_PREFIX</td>
</tr>
<tr>
<td>8188.</td>
<td>`TRANSLATE (0'</td>
</tr>
<tr>
<td>8189.</td>
<td>`generate (DC 256</td>
</tr>
<tr>
<td>8190.</td>
<td>`generate (LABEL</td>
</tr>
<tr>
<td>8191.</td>
<td><code>generate (MVC $TRACE+256,STTRACE+1')</code></td>
</tr>
<tr>
<td>8192.</td>
<td>`generate (0'</td>
</tr>
<tr>
<td>8193.</td>
<td><code>PC_GENNED_WITH_VECTOR := true</code></td>
</tr>
<tr>
<td>8194.</td>
<td><code>if (JE33G_PROCCOUNTS_REQD</code></td>
</tr>
<tr>
<td>8195.</td>
<td><code>or DEBUG_BLOCKCOUNTS_REQD)</code></td>
</tr>
<tr>
<td>8196.</td>
<td><code>and FIRST_PROC</code></td>
</tr>
<tr>
<td>8197.</td>
<td><code>then</code> <code>/* PROC counter must be generated since word one of first save area</code></td>
</tr>
<tr>
<td>8198.</td>
<td><code>points to proc trace vector, so we can't keep count there. */</code></td>
</tr>
<tr>
<td>8199.</td>
<td><code>PCT_SAVED_WITH_VECTOR := true</code></td>
</tr>
<tr>
<td>8200.</td>
<td><code>fi</code></td>
</tr>
<tr>
<td>8201.</td>
<td><code>fi</code></td>
</tr>
<tr>
<td>8202.</td>
<td><code>fi</code></td>
</tr>
<tr>
<td>8203.</td>
<td><code>fi</code></td>
</tr>
<tr>
<td>8204.</td>
<td><code>fi</code></td>
</tr>
<tr>
<td>8205.</td>
<td><code>fi</code></td>
</tr>
<tr>
<td>8206.</td>
<td><code>fi</code></td>
</tr>
<tr>
<td>8207.</td>
<td><code>if DEBUG_PROCCOUNTS_REQD or DEBUG_BLOCKCOUNTS_REQD</code></td>
</tr>
<tr>
<td>8208.</td>
<td><code>then</code> <code>/* We must update the count. */</code></td>
</tr>
<tr>
<td>8209.</td>
<td><code>if OS_LINKAGE</code></td>
</tr>
<tr>
<td>8210.</td>
<td><code>then</code> `/* Count will be in BLOCK_LABEL_PREFIX</td>
</tr>
<tr>
<td>8211.</td>
<td><code>if PCT_DEBUGGED_WITH_VECTOR</code></td>
</tr>
<tr>
<td>8212.</td>
<td><code>then</code> <code>/* Define PCT. */</code></td>
</tr>
<tr>
<td>8213.</td>
<td><code>LABEL := ''</code></td>
</tr>
<tr>
<td>8214.</td>
<td><code>fi</code></td>
</tr>
<tr>
<td>8215.</td>
<td><code>fi</code></td>
</tr>
<tr>
<td>8216.</td>
<td><code>fi</code></td>
</tr>
<tr>
<td>8217.</td>
<td><code>fi</code></td>
</tr>
<tr>
<td>8218.</td>
<td><code>fi</code></td>
</tr>
<tr>
<td>8219.</td>
<td><code>fi</code></td>
</tr>
<tr>
<td>8220.</td>
<td><code>fi</code></td>
</tr>
<tr>
<td>8221.</td>
<td><code>fi</code></td>
</tr>
<tr>
<td>8222.</td>
<td><code>fi</code></td>
</tr>
<tr>
<td>8223.</td>
<td><code>fi</code></td>
</tr>
<tr>
<td>8224.</td>
<td><code>fi</code></td>
</tr>
<tr>
<td>8225.</td>
<td><code>fi</code></td>
</tr>
<tr>
<td>8226.</td>
<td><code>fi</code></td>
</tr>
<tr>
<td>8227.</td>
<td><code>fi</code></td>
</tr>
<tr>
<td>8228.</td>
<td><code>fi</code></td>
</tr>
<tr>
<td>8229.</td>
<td><code>fi</code></td>
</tr>
<tr>
<td>8230.</td>
<td><code>fi</code></td>
</tr>
<tr>
<td>8231.</td>
<td><code>fi</code></td>
</tr>
<tr>
<td>8232.</td>
<td><code>fi</code></td>
</tr>
<tr>
<td>8233.</td>
<td><code>fi</code></td>
</tr>
<tr>
<td>8234.</td>
<td><code>fi</code></td>
</tr>
<tr>
<td>8235.</td>
<td><code>fi</code></td>
</tr>
<tr>
<td>8236.</td>
<td><code>fi</code></td>
</tr>
<tr>
<td>8237.</td>
<td><code>fi</code></td>
</tr>
<tr>
<td>8238.</td>
<td><code>fi</code></td>
</tr>
<tr>
<td>8239.</td>
<td><code>fi</code></td>
</tr>
<tr>
<td>8240.</td>
<td><code>fi</code></td>
</tr>
<tr>
<td>8241.</td>
<td><code>fi</code></td>
</tr>
<tr>
<td>8242.</td>
<td><code>fi</code></td>
</tr>
<tr>
<td>8243.</td>
<td><code>fi</code></td>
</tr>
<tr>
<td>8244.</td>
<td><code>fi</code></td>
</tr>
</tbody>
</table>

C-87
PROC PROC_INFO_SAVE

/* Save all information needed at CORP time. */
char I /* OFFSET_TO_CORP, biased by 50 and converted to character format. */

OPERAND1(CURRENT_NEST_LEVEL) := FIRST
OPERAND2(CURRENT_NEST_LEVEL) := LAST
OPERAND3(CURRENT_NEST_LEVEL) := SAVE_LENGTH
OPERAND4(CURRENT_NEST_LEVEL) := PREVIOUS_DEBUG_VECTOR
I := OFFSET_TO_CORP + 50

/* Bias value by 50 and convert to two-digit character string. */
INFORMATION(CURRENT_NEST_LEVEL) :=
   I || GS_LINEAGE || (SAVE(3) = 'DYNAN') || FIRST_VALUE_KNOWN ||
   PROC_ID_BYTE || SPECIAL_PREFIX

/*
```
81962. PROC PROC_MULTI_BAS3_G
81963. /* This proc generates any multiple base register adcons if
81964. needed and not yet generated and notes that such adcons have
81965. been generated. */
81966. int I, J /* Preparation. */
81967. 
81968. if MULTIBAS3
81969. then /* Multibase adcons required but not yet generated. */
81970. if LABEL = "" then /* Scramble around adcons. */
81971. LABEL := BLOCK_LABELPREFIX || 'FF'
81972. generate ("B" || LABEL)
81973. I := 2
81974. J := 4096
81975. X := BLOCK_LABELPREFIX || 'MBR'
81976. while I < 3'B3F do
81977. generate (I || X || DC('I' || BLOCK_LABELPREFIX || 'MBR'))
81978. X := ""
81979. I := I + 1
81980. J := J + 4096
81981. od /* Termination: I is incremented during loop, N'BASE is
81982. fixed; I must eventually exceed N'BASE. */
81983. MULTIBASE := false
81984. 
81985. for
81986. 
81987. 
81988. 
81989. 
81990. 
```

C-89
CORP macro -- 6 July 1373

3001. BASED CORP (USER_NAME; PROC_NAME, RETURN=, LINK=14, RESTORE=, RC=)
3002. /* Defines the end of a procedure block. The register or registers
3003. indicated by RESTORE are restored with the exception of those listed
3004. in RETURN. If RESTORE is omitted, all saved registers are restored
3005. (except those in the RETURN list). The return code is set from the
3006. RC operand and return is made to the address specified by the
3007. LINK operand, unless LINK=NONF is specified. */
3009. /* [Ground rule: No CORP proc modifies CURRENT_NEST_LEVEL.
3010. This can be shown via the cross-reference listing.] */
3012. int FIRST_SAVER2L2_REG, /* Register number which may be placed into the
3013. first word of the save area. This may be, for example, 14 when
3014. the first register saved is some symbolic of unknown value. */
3015. OFFSET_TO_GPPO / Offset in save area (in words) to the storage
3016. place for GPPO. This may be positive or negative. */
3018. bit ANY_REGS_SAVED, /* Indicates whether any registers were saved in
3019. this proc. */
3021. DYNAMIC_SAVERAREA, /* Indicates whether SAVE=(***,DYNAM,***)
3022. was coded on PROC. */
3023. RCK_AREA_B633, /* Indicates whether RCK save area is needed. */
3024. FIRST_VALUE_B643, /* Indicates whether FIRST is other than a
3025. symbolic. */
3026. FIRST_REG_ALLE2_KNOW2, /* Indicates whether first register to be
3027. saved (in RSP1) is other than symbolic. */
3028. OS_LINKAGE /* Indicates whether LINKAGE=(OS,*** was coded
3029. on PROC. */
3030. 
3031. LABLE / Any outstanding label waiting to be generated. */
3032. GPPO_OFFSET_SPERSON, /* OFFSET TO GPPO is character form. */
3034. FIRST_REG_SAVED2, LAST_REG_SAVED2,
3035. FIRST_REG_SAVED3, LAST_REG_SAVED3,
3037. DYNAMIC_SAVEMEM, /* Indicates whether SAVE=(***,DYNAM,***)
3039. was coded on PROC. */
3040. LOCAL_POINT2ERN, OS_POINTER, RESTORE_AREA,
3042. RC_RE, /* Register holding return code before restoring of
3044. registers. */
3046. LOCAL_POINTERN, /* Length of save area. */
3048. LOCAL_POINTERN, /* First and last register to be restored. */
3050. PROC_ID_BY3, / One-byte hex number used as identifier of current
3052. PROC in traces and the like. */
3054. PREVIOUSDEBUG VECTOR /* Value of debug switches (packed) before
3056. encountering this PROC or [ if DEBUG=(***,GLOBAL) specified
3058. null. */
3060. 
3061. call TRACE_PRINTER ( ; 'CORP')
3063. /* Print macro name "CORP" in mnote if tracing on. */
3065. if CURRENT_NEST_LEVEL > NESTING_LIMIT
3067. then
3069. call POP_OPL_BLOCK ( ; )
3071. fi
3073. 
3075. call VERIFY_END ( ; 'PROC', PROC_NAME)
3077. /* Verifies current block has the same specified in the PROC_NAME
3079. operand on the CORP macro (if any) and that it is in a PROC block.
3081. Various errors receive messages and either intermediate blocks are
3083. skipped as a fixup or ERROR_OCCURRED is set.
3085. (Lena: if CURRENT_NEST_LEVEL > 0 and
3087. [PROC_NAME = ''] or = BLOCK_NAME(CURRENT_NEST_LEVEL) ) and
3089. BLOCK_TYPE(CURRENT_NEST_LEVEL) = 'PROC', then
3091. ERROR_OCCURRED will be set false and CURRENT_NEST_LEVEL will not
3093. be modified. */
3095. if ERROR_OCCURRED
3097. then
3099. label
3010. 
3012. label := USER_NAME
3014. /* Generate label at first opportunity. */

C-90
C-91
"CORP" MACRO  --  6 July 1973

83148. PROC CORP_GET_PROC_INFO
83149. /* Get info saved by PROC macro. */
83150. char X */ Temporary. */
83151. FIRST_REG_SAVED := OPERAND1(CURRENT_NEST_LEVEL)
83152. LAST_REG_SAVED := OPERAND2(CURRENT_NEST_LEVEL)
83153. X := INFORMATION(CURRENT_NEST_LEVEL)
83154. OFFSET_TO_GPRO := X[1,2] - 50 /* Stored biased by 50. */
83155. if OFFSET_TO_GPRO < 0
83156. then
83157. GPRO_OFFSET_STRING := '-' || OFFSET_TO_GPRO
83158. /* In string conversion, absolute value is taken; restore sign. */
83159. else
83160. GPRO_OFFSET_STRING := '+' || OFFSET_TO_GPRO
83161. fi
83162. OS_LINKAGE := X[3,1]
83163. DYNAMIC_SAVAREA := X[4,1]
83164. FIRST_VALUE_KNOWN := X[5,1]
83165. PREVIOUS_DEBUG_VECTOR := OPERAND4(CURRENT_NEST_LEVEL)
83166. ANY_REGS_SAVED := (FIRST_REG_SAVED ' ')
83167. PROC_ID_BYTE := X[6,2]
83168. if X[8,1]
83169. then /* Special PROC prefix. */
83170. BLOCK_LABEL_PREFIX := '$P' || PROC_ID_BYTE
83171. else /* Standard prefix. */
83172. BLOCK_LABEL_PREFIX := '*' || BLOCK_NUMBER(CURRENT_NEST_LEVEL)
83173. fi
83174. if FIRST_VALUE_KNOWN
83175. then
83176. FIRST_SAVAREA_REG := FIRST_REG_SAVED /* Convert to integer. */
83177. else
83178. FIRST_SAVAREA_REG := 14
83179. fi
83180. /* FIRST_SAVAREA_REG is similar to the variable SAP of PROC. */
83181. C-92
/* Set RES1 and RES2 to RESTORE operands if present, else to FIRST and LAST saved. Set RESTORE_AREA to "SV". Also set MUL and COMMA to proper values. */

/* Assume: */

FEST1 := FIRST_REG_SAVED
REST2 := LAST_REG_SAVED
MUL := "M"
COMMA := ","
FIRST_REST_VALUE_KNOWN := FIRST_VALUE_KNOWN

/* Now find out. */

if ANY_REGS_SAVED
  then
    if RESTORE = "A"
      then
        mnote(3, 'SRCB301 NO REGISTERS SAVED--RESTORE IGNORED')
      fi
    else

      if RESTORE = "A"
        then
          REST1 := RESTORE(1)
        fi
      then
      fi

    if REST2 = "A"
      then
      thm.
      fi
    else

      if RESTORE = "A"
        then
          MUL, COMMA := ""
      fi
    fi

  RESTORE_AREA := "SV"

END
"CORP" MACRO -- 6 JULY 1973

3219.     #define CORP_DEBUGGING_SAVERO
3220.     /* Given:  - US_LINK:  if DEBUG_CORPVALUES_REQ, store copy of
3221.     register values into CORP savearea.  If CORPVALUES and RETURN= (or
3222.     BCS=) specified, copy SY save area to BCK save area, set
3223.     RESTORE_AREA to "BCK", and set BCK_AREA_REQ.  */
3224.     
3225.     char CORP_BCK_OFFSET,  /* Character string to be inserted to insure
3226.     registers are stored at proper offset from CORP/BCK label.  */
3227.     SV_OFFSET;   /* Character string to be inserted to reference proper
3228.     offset from SY save area label.  */
3229.     
3230.     if DEBUG_CORPVALUES_REQ
3231.       then
3232.         if DEBUG_SAVEMACH_REQ
3233.           then
3234.             SV_OFFSET := ' + 12
3235.             CORP_BCK_OFFSET := ' + 12
3236.           else
3237.             CORP_BCK_OFFSET := ''
3238.           if DEBUG_BUCKCOMPS_REQ
3239.             then
3240.                 SV_OFFSET := ' + 4
3241.             else
3242.                 SV_OFFSET := ''
3243.           fi
3244.       fi
3245. 
3246.     assemble ("label || ' |m |_12, || BLOCK_LABEL_PREFIX ||
3247.     'CORP || CORP_BCK_OFFSET")
3248.     LABEL := '
3249.     if DEBUG_MACH_REQ
3250.         then
3251.             BCK Area >>: $213
3252.           if DEBUG_MACH_OFF / (' ' || BLOCK_LABEL_PREFIX || 'BCK'
3253.             CORP_BCK_OFFSET || (' ' || SAVE_LENGTH || '4'),
3254.             BCK_LABEL_PREFIX || 'SY' || SV_OFFSET)
3255.           fi
3256.         fi
3257.     else
3258.     fi
3259.     
3260.     *end...
CORP Macro -- 6 July 1973

83259. PROC CORP_RESTORE_ASG13
83260. /* If current save area is dynamic, save pointer to it in GP#1. In
83261. any case, load GP#13 to point to previous save area. Given:
83262. OS_LINKAGE is true. */
83263. if DYNAMIC_SAVEAREA
83264. then
83265. generate (LABEL || ' LR 1,13')
83266. LABEL := "
83267. fi
83268. if SAVETRACE (FIRST_PROC and PROC_ID_BYTE = '01"
83269. then
83270. generate (LABEL || ' L 13,FIRSTSV+4")
83271. else
83272. generate (LABEL || ' L 13,OSSTATE")
83273. fi
83274. LABEL := "
83275. fi
83276. C-95
PROC CORP,SET,RETURN,CODE

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

RC_REG := '15'  /* Indicate no return code. */

if RC = '15'
then
  if OS_LINKAGE
  then
    GENERATE (LABEL || ' SR 15,15')
    /* Clear 15 for normal OS return. */
    LABEL := '11'
    RC_REG := '15'
  fi
else
  if RC[1,1] = '('
  then
    /* Register was specified. */
    RC_REG := RC(1)
  else
    /* Value was specified. */
    if RC = 'NONE'
    then
      if RC = '0'
      then
        GENERATE (LABEL || ' SR 15,15')
      fi
      fi
      fi
else
  fi
fi
fi
fi
fi
fi
fi
fi
fi
 fi
fi
fi

"CORP" Macro -- 6 July 1973
/* For the register containing the return code and all those listed in RETJN=. store each register into the appropriate word of the save area from which the ultimate LM instruction will occur (setting OS_POINTER and LOCAL_POINTER to indicate this save area). However, if no registers are to be restored, then this proc is a nop. */

int OFFSET, I

if ANY_RES_SAVE then
    if OS_LIKE then
        LOCAL_POINTER := "[13]"
    else
        LOCAL_POINTER := ""
    end
    LOCAL_POINTER := BLOCK_LABEL_PREFIX || RESTORE_AREA || "4"

    if RC_REG = "" then
        for every restore, do
            if RETURN(I) = 'N' /* self-defining term */ then
                if FIRST_SAVEAREA_REG < 14 then
                    LOCAL_POINTER := "" || RETURN(I) || ' ' || RC_REG
                else
                    LOCAL_POINTER := "" || RETURN(I) || ' ' || OS_POINTER
                end
            else
                LOCAL_POINTER := "" || RETURN(I) || ' ' || ' ' || RC_REG || "4"
            end
        end
        I := I + 1
        while I < N\'RETURN do
            if RETURN(I) = 'N' /* self-defining term */ then
                OFFSET := (OFFSET_TO_SPRO - 1) * 4
                generate (LABEL || 'ST' || RETURN(I) || ' ' || RC_REG || ' ' || OS_POINTER)
            else
                if FIRST_SAVEAREA_REG < 14 then
                    generate (LABEL || 'ST' || RETURN(I) || ' ' || RC_REG || ' ' || OS_POINTER)
                else
                    generate (LABEL || 'ST' || RETURN(I) || ' ' || RC_REG || ' ' || OS_Pointer)
                end
            end
        end
    else
        for every restore, do
            if RETURN(I) = 'N' /* self-defining term */ then
                if FIRST_SAVEAREA_REG < 14 then
                    generate (LABEL || 'ST' || RETURN(I) || ' ' || RC_REG || ' ' || OS_POINTER)
                else
                    generate (LABEL || 'ST' || RETURN(I) || ' ' || RC_REG || ' ' || OS_POINTER)
                end
            else
                if FIRST_SAVEAREA_REG < 14 then
                    generate (LABEL || 'ST' || RETURN(I) || ' ' || RC_REG || ' ' || OS_Pointer)
                else
                    generate (LABEL || 'ST' || RETURN(I) || ' ' || RC_REG || ' ' || OS_Pointer)
                end
            end
        end
        I := I + 1
        while I < N\'RETURN do
            if RETURN(I) = 'N' /* self-defining term */ then
                if FIRST_SAVEAREA_REG < 14 then
                    generate (LABEL || 'ST' || RETURN(I) || ' ' || RC_REG || ' ' || OS_Pointer)
                else
                    generate (LABEL || 'ST' || RETURN(I) || ' ' || RC_REG || ' ' || OS_Pointer)
                end
            else
                if FIRST_SAVEAREA_REG < 14 then
                    generate (LABEL || 'ST' || RETURN(I) || ' ' || RC_REG || ' ' || OS_Pointer)
                else
                    generate (LABEL || 'ST' || RETURN(I) || ' ' || RC_REG || ' ' || OS_Pointer)
                end
            end
        end
    end
end

C-97
"CORP" Macro -- 6 July 1973

83381. macro CORP_FREE_DYN_SAVESAVE
83382.  /* Issue FREEMAIN for dynamic save area. */
83383.  generate (LABEL || ' LA ' || SAVE_LENGTH || '***')
83384.  LABEL := ''
83385.  generate (' FREEMAIN L,LV=(0),A=(1) ')
83386.  end

*...*...*...*...*...*...*...*...*...*...*...*...*...*...*...*...*...*...*...*...*...*...*...*...*...*...*...*...*...*...
PROC CORP_RESTORE_REGS

/* Restore registers REST1 through REST2 from proper save area if
   saved. */

int OFFSET, I /* Temporaries. */

if ANY_REGS_SAVED
    then
    if FIRST_REST_VALUE_KNOW
        I := REST1 /* Convert to integer. */
        OFFSET := (OFFSET_TO_GPR0 * I - ((I + 2)/16*16)) * 4
        GPR_OFFSET (LABEL || L || MULI || REST1 || REST2 || COMMA2 || LOCAL_POINTER || OFFSET || OS_POINTER)
        else
            if FIRST_SAVEAREA_REGS < 1
                GPR_OFFSET (LABEL || L || MULI || REST1 || REST2 || COMMA2 || LOCAL_POINTER || OS_POINTER)
                GPR_OFFSET_STRING || "*4" || OS_POINTER)
            else
                GPR_OFFSET (LABEL || L || MULI || REST1 || REST2 || COMMA2 || LOCAL_POINTER || OS_POINTER)
                "*({(REST1 || REST2 || COMMA2 || LOCAL_POINTER || OS_POINTER || "{(REST1 || REST2 || COMMA2 || LOCAL_POINTER || OS_POINTER || "{(REST1 || REST2 || COMMA2 || LOCAL_POINTER || OS_POINTER || "{(REST1 || REST2 || COMMA2 || LOCAL_POINTER || OS_POINTER || "{(REST1 || REST2 || COMMA2 || LOCAL_POINTER || OS_POINTER || "{(REST1 || REST2 || COMMA2 || LOCAL_POINTER || OS_POINTER || "{(REST1 || REST2 || COMMA2 || LOCAL_POINTER || OS_POINTER || "{(REST1 || REST2 || COMMA2 || LOCAL_POINTER || OS_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_POINTER || OS_POINTERS) || "{(REST1 || REST2 || COMMA2 || LOCAL_POINTER || OS_POINTERS) || "{(REST1 || REST2 || COMMA2 || LOCAL_POINTER || OS_POINTERS) || "{(REST1 || REST2 || COMMA2 || LOCAL_POINTER || OS_POINTERS) || "{(REST1 || REST2 || COMMA2 || LOCAL_POINTER || OS_POINTERS) || "{(REST1 || REST2 || COMMA2 || LOCAL_POINTER || OS_POINTERS) || "{(REST1 || REST2 || COMMA2 || LOCAL_POINTER || OS_POINTERS) || "{(REST1 || REST2 || COMMA2 || LOCAL_POINTER || OS_POINTERS) || "{(REST1 || REST2 || COMMA2 || LOCAL_POINTER || OS_POINTERS) || "{(REST1 || REST2 || COMMA2 || LOCAL_POINTER || OS_POINTERS) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "{(REST1 || REST2 || COMMA2 || LOCAL_Pointer) || "$C" 

C-99
"CORP" macro -- 6 July 1973

93418. PROC CORP_RESTORE_DEBUG_ENVIRONMENT
93419. /* Restore local flags which were in process 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_STACK_REQD := PREVIOUS_DEBUG_VECTOR[1,1]
93426. DEBUG_PROGRAMS_REQD := PREVIOUS_DEBUG_VECTOR[2,1]
93427. DEBUG_LISTS_REQD := PREVIOUS_DEBUG_VECTOR[3,1]
93428. DEBUG_BLOCK_COUNTS_REQD := PREVIOUS_DEBUG_VECTOR[4,1]
93429. DEBUG_PROC_COUNTS_REQD := PREVIOUS_DEBUG_VECTOR[5,1]
93430. DEBUG_PROCEDURE_REQD := PREVIOUS_DEBUG_VECTOR[6,1]
93431. DEBUG_STACK_VALUES_REQD := PREVIOUS_DEBUG_VECTOR[7,1]
93432. DEBUG_SAVE_STACK_REQD := PREVIOUS_DEBUG_VECTOR[8,1]
93433. fi
93434. end

C-100
03466. "CORP" Macro -- 6 July 1973

03467. PROC CORP, NCRP _8K_AREAS

03468. /* If rejuiced, generate CRP and BCK save areas. */

03469. char LAST_A'Re, /* Label of CRP or BCK area, whichever is generated

03470. b Last. */

03471. FWD_PTR, /* Label generated as forward pointer in last area. */

03472. TARGET /* Temporary. */

03473. if OS_LINKAGE and DEBUS_CJRPVALUES_REQD

03474. then /* We need a CRP save area. */

03475. if LINK = '033Z'

03476. then /* We need to generate branch around save areas. */

03477. TARGET := BLOCK_LABEL_PREFIX || 'FIN'

03478. GENERATE (LABEL || 'B' || TARGET)

03479. LAST_AREA := BLOCK_LABEL_PREFIX || 'CRP'

03480. GENERATE (LAST_AREA || 'DS OF')

03481. if DEBig_SAVE_TRACE_REQD

03482. then GENERATE ("DC X'FC" || PROC_ID_BYTE || "0000")

03483. if CK_AREA_REQD

03484. then GENERATE ("DC X'FB" || PROC_ID_BYTE || "0000")

03485. if BCK_AREA_REQD

03486. then GENERATE ("DC X'FD" || PROC_ID_BYTE || "0000")

03487. if BCK_AREA_REQD

03488. then /* Generate the BCK save area. */

03489. LAST_A'Re := BLOCK_LABEL_PREFIX || 'BCK'

03490. LABEL := TARGET

03491. FWD_PTR := BLOCK_LABEL_PREFIX || 'BCK'

03492. GENERATE (PREV_SAVE_TRACE_PTR || 'BCK' || BLOCK_LABEL_PREFIX || 'CRP')

03493. GENERATE ('DC A'|| PREV_SAVE_TRACE_AREA || 'FWD_PTR || ')")

03494. fi

03495. if BCK_AREA_REQD

03496. last GENERATE ("DC X'FD" || PROC_ID_BYTE || "0000")

03497. fi

03498. fi

03499. fi

03500. PREV_SAVE_TRACE_PTR := FWD_PTR

03501. PREV_SAVE_TRACE_AREA := LAST_AREA

03502. fi

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

83489. P4Oe COME whose EXIT_LABEL
83490. /* If an EXIT label is required, put it into LABEL (generating
83491. any label already there). Issue mute regarding EXIT references. */
83492. if EXIT_LABEL_132J(CURRENT_NEST_LEVEL)
83493. then
83494. if LABEL # ""
83495. then
83496. LABEL := BLACK LABEL_PREFIX ++ "IT"
83497. if EXIT_SEVERITY = ""
83498. then
83499. EXIT_SEVERITY := '4'
83500. endif
83501. endif
83502. endwhile
83503. /* Label will have been generated by POP OLD BLOCK time. */
83504. 22L2 C-102
"EXIT_FIND" Macro -- 10 July 1973

91001. 
91002. "EXIT_FIND" ( ; REQD_NAME)
91003. /* Set ULTIMATE_BRANCH_LABEL to exit label for block whose name
91004. is the argument; if no such block, issue message and set
91005. ERROR_OCCURRED. On valid block, that block is marked as needing
91006. an XIr Label. */
91007. int I /* Temporary */
91008. glll TRACE_PRINTER ( ; 'EXITFIND')
91009. /* Print macro name "EXITFIND" in mnote if tracing on. */
91010. ERROR_OCCURRED := false /* Assume all will go well. */
91011. I := CURRENT_NEST_LEVEL - 1 /* Start search at surrounding block. */
91012. if REQD_NAME = ' ' and '*' then /* We must search for the right block. */
91013. while I > 0 and REQD_NAME # BLOCK_NAME(I) and
91014. BLOCK_TYPE(I) = 'PROC'
91015. do I := I - 1
91016. if I < 0 or
91017. (REQD_NAME # BLOCK_NAME(I) and '' and '*')
91018. then /* Not found in search. */
91019. REQD_NAME = BLOCK_NAME(I)
91020. if I < 0 or
91021. (BLOCK_TYPE(I) = 'DO') and
91022. ERROR_OCCURRED := true
91023. else
91024. if BLOCK_TYPE(I) = 'PROC' and INFORMATION(I)[6,1] or
91025. INFORMATION(I)[7,1]
91026. then /* Just use special PROC prefix form. */
91027. ULTIMATE_BRANCH_LABEL := '$P' || INFORMATION(I)[6,2] || 'EXIT'
91028. else
91029. ULTIMATE_BRANCH_LABEL := '$P' || BLOCK_NUMBER(I) || 'EXIT'
91030. fi
91031. else
91032. fi
91033. fi
91034. else
91035. fi
91036. if I > 0.
91037. then
91038. ERROR_OCCURRED := true
91039. if I < 0 or
91040. (BLOCK_TYPE(I) = 'DO') and
91041. INFORMATION(I)[6,1] or
91042. INFORMATION(I)[7,1]
91043. then /* STC9103 EXIT TO IMMEDIATELY SURROUNDING BLOCK INVALID */
91044. ERROR_OCCURRED := true
91045. else
91046. BLOCK_TYPE(I) = 'PROC' and INFORMATION(I)[8,1]
91047. then /* STC9104 EXIT TO DO BLOCK INVALID WITHIN ATEND OR ONEXIT */
91048. ERROR_OCCURRED := true
91049. else
91050. if BLOCK_TYPE(I) = 'PROC' and INFORMATION(I)[8,1]
91051. then /* STC9105 EXIT TO DO BLOCK INVALID WITHIN ATEND OR ONEXIT */
91052. ERROR_OCCURRED := true
91053. else
91054. fi
91055. fi
91056. fi
91057. fi
91058. fi

/*=*/
"POP_OLD_BLOCK" Macro — 10 July 1973

2002. /* Remove the current block from the stack. Also generate END and XIT labels if required. */
2003.  
2004. call TRACE_PRINTER ( ; "POP")
2005. /* Eject macro name "POP" in mnote if tracing on. */
2006.  
2007. if CURRENT_NEST_LEVEL > NESTING_LIMIT
2008.   then
2009.       if END_LABEL_REQD(CURRENT_NEST_LEVEL)
2010.         then
2011.           generate ('$' || BLOCK_NUMBER(CURRENT_NEST_LEVEL) || 'END DS OH')
2012.       fi
2013.       if EXIT_LABEL_REQD(CURRENT_NEST_LEVEL) or OLD_EXIT =''
2014.         then
2015.           if EXIT_SEVERITY = *
2016.             then
2017.               EXIT_SEVERITY := 'O'
2018.             fi
2019.           fi
2020.           "STC0201 ONE OR MORE EXIT'S REFERENCE THIS POINT"
2021.       fi
2022.   fi
2023.   if OLD_EXIT =''
2024.     then
2025.       generate (OLD_EXIT || 'DS OH')
2026.     fi
2027.   fi
2028. fi
2029. if DEBUG_LISTLOCKS_REQD
2030.   then
2031.     "NOTE (*, 'STC0301 END OF BLOCK' || BLOCK_NUMBER(CURRENT_NEST_LEVEL)
2032.     || '(' || BLOCK_NAME(CURRENT_NEST_LEVEL) || ') AT DEPTH' ||
2033.     CURRENT_NEST_LEVEL)
2034.     "**********************************************************************"
2035.   fi
2036. fi
2037. CURRENT_NEST_LEVEL := CURRENT_NEST_LEVEL - 1
2038.  
2039. /* (Lemma: Execution of POP_OLD_BLOCK always decrements CURRENT_NEST_LEVEL by exactly one.) */
2040.  
2041.  

**C-104**
PUSH NEW BLOCK

Macro -- 10 July 1973

93001. macro PUSH NEW BLOCK (BLOCK NAME VALUE, BLOCK TYPE VALUE, OPERAND1 VALUE, OPERAND2 VALUE, OPERAND3 VALUE, INFORMATION VALUE, END LABEL VALUE = false)
93002. 
93003. 
93004. 
93005. 
93006. 
93007. 
93008. 
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. */
93012. 
93013. call TRACE PRINTER (, 'PUSH')
93014. 
93015. cCASE NEST LIMIT := 100
93016. /* Insure maximum 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. */
93019. 
93020. 
93021. ERROR OCCURRED := false
93022. CURRENT NEST LEVEL := CURRENT NEST LEVEL + 1
93023. if CURRENT NEST LEVEL > NESTING LIMIT
93024. then
93025. note (1, 'STAC9901 BLOCK NESTING LIMIT OF ' || NESTING LIMIT ||
93026. ' EXCEEDED--MACROS MUST BE MODIFIED')
93027. ERROR OCCURRED := true
93028. else
93029. LAST BLOCK NUMBER := LAST BLOCK NUMBER + 1
93030. /* Set block number for this block. */
93031. BLOCK NUMBER (CURRENT NEST LEVEL) := LAST BLOCK NUMBER
93032. if NEST_LABEL EQD (CURRENT NEST LEVEL) := false
93033. then
93034. OPERAND1 (CURRENT NEST LEVEL) := OPERAND1 VALUE
93035. OPERAND2 (CURRENT NEST LEVEL) := OPERAND2 VALUE
93036. OPERAND3 (CURRENT NEST LEVEL) := OPERAND3 VALUE
93037. INFORMATION (CURRENT NEST LEVEL) := INFORMATION VALUE
93038. END_LABEL EQD (CURRENT NEST LEVEL) := END_LABEL VALUE
93039. BLOCK NAME (CURRENT NEST LEVEL) := BLOCK NAME VALUE
93040. if BLOCK NAME VALUE = ''
93041. then
93042. BLOCK NAME (CURRENT NEST LEVEL) := 'BLK' || LAST BLOCK NUMBER
93043. fi
93044. BLOCK_LABEL PREFIX := '' || LAST BLOCK NUMBER
93045. if DEBUG LIST BLOCKS EQD
93046. then
93047. BLOCK TYPE (CURRENT NEST LEVEL-1) = 'DCASE' and
93048. BLOCK TYPES VALUE = 'CASE'
93049. then
93050. note (3, 'STAC902 NON-CASE BLOCK IMMEDIATELY SURROUNDED BY DCASE INVALID')
93051. fi
93052. if DEBUG LIST BLOCKS EQD
93053. then
93054. note (3, '********************************************************************************')
93055. note (3, 'STAC902 START OF BLOCK ' || LAST BLOCK NUMBER || ' (' ||
93056. BLOCK NAME (CURRENT NEST LEVEL) || ') AT DEPTH ' || CURRENT NEST LEVEL)
93057. fi
93058. fi
93059. fi
93060. fi
/*SIMPLE_CONDITIONAL Macro -- 10 July 1973

94001. MACRO SIMPLE_CONDITIONAL (LABEL;
94002.     OP_CODE,
94003.     OPER1,
94004.     OPER2,
94005.     OPER3,
94006.     OPER4,
94007.     BRANCH_LABEL,
94008.     FALLTHRU_CONDITION,
94009.     OP_CODE?)
94010.  /* Separate 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.  BCTA3 /* Label to go on bc instruction. */
94015.  call TRACE_PRINTR ( "SIMPCOND")
94016.  /* Print macro name "SIMPCOND" in note if tracing on. */
94017.  call SIMPCOND_OPE_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.  do case OP_CODE?
94022.  CASE 1 /* Mask or relation only. */
94023.  BC_TAG := LABEL
94024.  CASE 2
94025.  mnote (3, "STANDARD INSUFFICIENT OPERANDS FOR TEST " || OP_CODE || ")
94026.  CASE 3
94027.  generate (LABEL || " " || OP_CODE || " " || OPER1)
94028.  CASE 4
94029.  generate (LABEL || " " || OP_CODE || " " || OPER1 || "", " || OPER2)
94030.  CASE 5
94031.  generate (LABEL || " " || OP_CODE || " " || OPER1 || "", " || OPERB)
94032.  CASE misc
94033.  mnote (3, "STANDARD SUPEFLUOUS OPERANDS FOR TEST " || OP_CODE || ")
94034.  endif
94035.  if LOCAL_REL = "."
94036.  if FALLTHRU_CONDITION /* is true: */
94037.  then /* Invert relation. */
94038.  if LOCALicemail[1,1] = "R"
94039.  then
94040.  LOCAL_REL := LOCAL_REL[2,7]
94041.  else
94042.  LOCAL_REL := "R" || LOCAL_REL
94043.  endif
94044.  endif
94045.  endif
94046.  endif
94047.  endif
94048.  if LOCAL_REL = "."
94049.  if FALLTHRU_CONDITION /* is true: */
94050.  then /* Invert mask. */
94051.  if LOCAL_REL[1,1] = "R"
94052.  then
94053.  LOCAL_REL := LOCAL_REL[2,7]
94054.  else
94055.  LOCAL_REL := "R" || LOCAL_REL
94056.  endif
94057.  endif
94058.  endif
94059.  //generate (BC_TAG || " II LOCAL_REL || " || BRANCH_LABEL)
94060.  endif
94061.  endif /* Invert mask. */
94062.  endif /* Invert mask. */
94063.  endif /* Invert mask. */
94064.  endif /* Invert mask. */
94065.  endif /* Invert mask. */
/* Extract LOCAL_ASK or LOCAL_REL from OPER's. If LOCAL_REL is a
   external value (GT, GE, EQ, LT, or LE), replace it with the
   proper value (H, NL, E, L, or NH). */

if OP_COUNT = 0
  then (9, 'STRING NO CONDITION SPECIFIED—"MASK=0" ASSUMED')
  LOCAL_ASK := '0'
else
  if SYSLIST(OP_COUNT)[1,5] = 'MASK=
     then LOCAL_ASK := SYSLIST(OP_COUNT)[6,8]
  else LOCAL_ASK := SYSLIST(OP_COUNT)[1,4] - 'REL='
  LOCAL_REL := SYSLIST(OP_COUNT)[5,8]
else
  LOCAL_REL := SYSLIST(OP_COUNT)
fi
do case LOCAL_REL of
  case 'GT' LOCAL_REL := 'H'
  case 'GE' LOCAL_REL := 'NL'
  case 'EQ' LOCAL_REL := 'E'
  case 'LT' LOCAL_REL := 'L'
  case 'LE' LOCAL_REL := 'NH'
  fi
fi
fi
"TRACE.PRINTER" Macro -- 12 July 1973

95001. macro TRACE.PRINTER( ; MACRO_NAME)
95002. /* Prints macro name if tracing on. */
95003. if DEBUG_MACRO_NAMES_REQD
95004. then
95005. inote (*, 'SZRC9500', MACRO_NAME)
95006. fi
95007. end
macro VERIFY_END ( ; REQD_TYPE, REQD_NAME)

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

/* Temporary. */
"VERIFY-END" Macro -- 10 July 1973

ERROR_OCCURRED := false /* Assumed. */

if REQD_NAME = "" then
  if CURRENT_NEST_LEVEL ≤ 0 then
  then
    note (6, 'STRC9607 NO BLOCKS ACTIVE--MACRO IGNORED')
    ERROR_OCCURRED := true
  else
    block (6, 'STRC9601 ONE BLEND ASSUMED TO GET TO "" | REQD_TYPE |"
    call BLEND (;
    else
      block (6, 'STRC9602 TWO BLENDS ASSUMED TO GET TO "" | REQD_TYPE |"
      call BLEND (;
      else
        block (6, 'STRC9603 CURRENT BLOCK IS NOT "" | REQD_TYPE |"
        ERROR_OCCURRED := true
        fi
      fi
    fi
  fi
else /* A block name was specified. */
  I := CURRENT_NEST_LEVEL
  while I > 0 and REQD_NAME ≠ BLOCK_NAME(I) do
    I := I - 1
    od /* (Termination: I is decremented and would eventually become ≤ 0 even if other test never occurs. */
  if I ≤ 0 then
    note (6, 'STRC9604 NO ACTIVE BLOCK NAMED "" | REQD_NAME |"
    ERROR_OCCURRED := true
  else
    block_type (CURRENT_NEST_LEVEL) ≠ REQD_TYPE
    then
      block (6, 'STRC9605 BLOCK "" | REQD_NAME |" IS NOT A "" | REQD_TYPE |"
      ERROR_OCCURRED := true
    else
      while CURRENT_NEST_LEVEL > I do
        block named found, but of wrong type. */
        note (6, 'STRC9606 END OF BLOCK "" | REQD_NAME |"
        ERROR_OCCURRED := true
      od
      block (6, 'strc9607 END OF BLOCK "" | REQD_NAME |"
      call BLEND (;
      /* (Lemma: If CURRENT_NEST_LEVEL > 0 and no BLEND operands specified, BLEND will decrement CURRENT_NEST_LEVEL by exactly one.) */
      od /* (Termination: On all iterations, I is fixed and CURRENT_NEST_LEVEL > I > 0. BLEND decrements CURRENT_NEST_LEVEL eventually become ≤ (actually =) I.) */
  fi
fi
mend

/* (Lemma: If CURRENT_NEST_LEVEL > 0 and
 [REQD_NAME = "" or = BLOCK_TYPE(CURRENT_NEST_LEVEL)] and
 REQ_TYPE = BLOCK_TYPE(CURRENT_NEST_LEVEL), then
 ERROR_OCCURRED will always be set false and CURRENT_NEST_LEVEL will be unmodified. Proof: If the hypothesized conditions are true, the module calls TRACE_PRINTER and sets ERROR_OCCURRED to false as its only actions. TRACE_PRINTER modifies no globals.] */

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

# CONDITIONAL_EXPRESSION_PROCESSOR (FIRST_ID; )
# Process the positional operands (the SYSLIST) as passed directly
# from calling macro beginning with SYSLIST(FIRST_INDEX) through
# SYSLIST(LAST_INDEX) generating the indicated tests to pass control
# to ULTIMATE_FALLTHRU_LABEL when the ULTIMATE_FALLTHRU_CONDITION is
# found to match the logical value tested and branches to the
# ULTIMATE_BRANCH_LABEL otherwise; the UNIQUE_LABEL_ID is used to
# insure unique labels; if a branch is made to the fall-through label,
# FALLTHRU_LABEL_USED is set, else unaltered. Only the "SYSLIST"
# operand is passed as an actual macro operand. The other variables
# mentioned are globals. */

int COND_COUNT, /* Counts the simple conditionals within the
      conditional expression. */
      DEPTH, /* Angle bracket nesting depth of simple conditional being
          processed. */
      INDEX, /* Operand index within the SYSLIST of the simple conditional
          being processed. */
      OP_COUNT, /* Number of operands in the simple conditional being
          processed. */
      NEXT_INDEX, /* Index of the AND or OR which follows the current
          simple conditional. */
      LA_DEPTH, /* Angle bracket depth during operand look-ahead. */
      I, /* Operand index of operand being examined during
          look-ahead. */

bit AND_OR_OUTSTANDING, /* Indicates whether an AND or OR follows
               the current simple conditional. */
      LOCAL_FALLTHRU_CONDITION, /* Logical value of the simple conditional
          being processed which is to lead to control falling through the
          test. */
      LOCAL_LABEL_REQD(20), /* Indicates whether the corresponding
               simple conditionals require a label due to branching logic. */

char LABEL, /* Outstanding label waiting to be generated. */
      LOCAL_BRANCH_LABEL /* Label for branch target if current simple
          conditional has the opposite truth value from that stored in
          LOCAL_FALLTHRU_CONDITION. */
call TRACE_PRINTER ( : 'CEP')

LABEL := FIRST_INDEX

/* Set calling label as outstanding. */

COND_COUNT := 0

DEPTH := 0

INDEX := FIRST_INDEX

while INDEX ≤ LAST_INDEX do

   call CEP_FIND_NEXT_CONDITION

   /* Step INDEX up to next simple conditional incrementing DEPTH for any
     UNCH found and setting ERROR_OCCURED on any syntax error. Increments
      COND_COUNT for the condition found. */

   if ERROR_OCCURED

      then

         BREAK

   fi

   ANO_OR_OUTSTANDING := false

call CEP_LOOKAHEAD

/* Find LOCAL_BRANCH LABEL or generate one. If LOCAL_BRANCH_LABEL
   is the ULTIMATE_FALLTHRU_LABEL, set FALLTHRU_LABEL_USED. Set
   LOCAL_FALLTHRU_CONDITION. Also set NEXT_INDEX and NEXT_DEPTH with
   the INDEX/DEPTH of the ANO OR following this conditional.
   (The value of NEXT_INDEX returned is greater than the value
   of INDEX entered.) */

   if ERROR_OCCURED

      then

         BREAK

   fi

   if LOCAL_LABSL_SEQ(COND_COUNT)

      then

         LABEL := BLOCK_LABEL_PREPIL || UNIQUE_LABEL_ID || COND_COUNT

   fi

   OP_COUNT := M'SYSLIST(INDEX)

   CALL SIMPLE_CONDITIONAL(LABEL)

   SYSLIST(INDEX,1),

   SYSLIST(INDEX,2),

   SYSLIST(INDEX,3),

   SYSLIST(INDEX,4),

   SYSLIST(INDEX,5),

   LOCAL_FALLTHRU_LABEL,

   LOCAL_FALLTHRU_CONDITION,

   OP_COUNT

   LABEL := *T

   INDEX := NEXT_INDEX

   DEPTH := NEXT_DEPTH

   goto /* Termination: INDEX is incremented (by CEP_LOOKAHEAD's
   return of NEXT_INDEX) and LAST_INDEX is fixed in loop; INDEX must
   eventually exceed LAST_INDEX. */

   if AND_OR_OUTSTANDING

      then

         MOVE (0, *STRC9701 INSUFFICIENT OPERANDS*)

         if

            if DEPTH ≠ 0

               then

                  MOVE (0, *STRC9702 INSUFFICIENT BRACKETS*)

               fi

            fi

         fi

   end
57100. \texttt{PROC CEP_FIND_NEXT_CONDITION}
57101. /* Step up to next simple condition, incrementing DEPTH for any "<" 
57102. or ">" found. Start with \texttt{SYSLIST(INDEX)} and advance INDEX till 
57103. found or \texttt{LAST_INDEX}. Set \texttt{ERROR_OCCURRED} on any syntax error, 
57104. giving message. Increment \texttt{COND_COUNT} by 1. */
57106. \texttt{ERROR_OCCURRED} := \texttt{false}
57107. \texttt{while INDEX} \leq \texttt{LAST_INDEX} and (\texttt{SYSLIST(INDEX)} = "<" or = ">")
57108. \texttt{do} 
57109. \texttt{DEPTH} := \texttt{DEPTH} + 1
57110. \texttt{INDEX} := \texttt{INDEX} + 1
57111. \texttt{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. \texttt{if SYSLIST(INDEX)[1,1] \# "}
57115. \texttt{then}
57116. \texttt{ERROR_OCCURRED := TRUE}
57117. \texttt{fi}
57118. \texttt{COND_COUNT} := \texttt{COND_COUNT} + 1
57119. \texttt{cor}
57120. \texttt{end}

\texttt{57119. if \texttt{ERROR_OCCURRED} := \texttt{TRUE}}
PROC CEP_LOOKAHEAD

Search operands beyond current simple conditional. If AND/OR found, set AND OR OUTSTANDING. Find spot this test is to branch to and put label found or generated for that spot into LOCAL BRANCH_LABEL. If case an 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 ≤ LAST_INDEX and (SYSLIST(I) = '>'; OR = '/') do
    LA_DEPTH := LA_DEPTH - 1
else
    LA_DEPTH := depth
    /* [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
            /* Search ahead for branch target. Set AND OR OUTSTANDING, LOCAL_FALLTHRU_CONDITION, and LOCAL BRANCH_POINT. */
        else
            ERROR_OCCURRED := true
            /* [Lemma: The value of NEXT_INDEX returned is greater than the value of INDEX entered.] */

C-114
CONDITIONAL_EXPRESSION_PROCESSOR macro -- 9 July 1973

97171. BREQ 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. LOCKFOB /* Operation ("AND" or "OR") opposite of SYSLIST(I). */
97181. MAX_DEPTH, /* The maximum depth at which the LOOKFOR'ed
97182. TARGET /* Simple conditional number which is the target for
97183. the branch. */
AND_OBS_OUTSTANDING := false

if SYSLIST(I) = 'AND'
then
LOCAL_FALLTHRU_CONDITION := true
MAX_DEPTH := LA_DEPTH
else /* Operation is OR. */
LOCAL_FALLTHRU_CONDITION := false
if LA_DEPTH = 0
then
I := LAST_INDEX + 1
fi /* Advance I to force skip of unnecessary search. */
LOCKPOS := 'AND'
MAX_DEPTH := LA_DEPTH - 1
fi

if SYSLIST(I) = 'AND' and OR
then
LOCAL_FALLTHRU_CONDITION := true

else
if SYSLIST(I) = '>' or '/'
then
if LA_DEPTH > 0
then
LA_DEPTH := LA_DEPTH - 1
else
if SYSLIST(I)[1,1] = '('
then
TARGET := TARGET + 1
else
-- SYSLIST(I) # 'AND' and # 'OR'
then
fi

else
fi

od /* [Termination: I is either incremented or set to LAST_INDEX + 1; if other tests never occurred.] */

if I > LAST_INDEX
then
if (LOCKFOR = 'OR' /* Operand was AND. */ and
~ ULTIMATE_FALLTHRU_CONDITION) or
(LOCKFOR = 'AND' /* Operand was OR. */ and
ULTIMATE_FALLTHRU_CONDITION /* is true */)
then
LOCAL_BRANCH_LABEL := ULTIMATE_FALLTHRU_LABEL
Fallthru_LABEL_USED := false
else
LOCAL_BRANCH_LABEL := ULTIMATE_BRANCH_LABEL
fi
else
LOCAL_BRANCH_LABEL := BLOCK_LABEL_PREFIX || UNIQUE_LABEL_ID || TARGET
LOCAL_LABEL_REQ(TARGET) := false
/* Note that we are relying on the automatic initialization of the
LOCAL_LABEL_REQ array to false at start of every invocation of
CONDITIONAL_EXPRESSION_PROCESSOR. */

end

* * *
maco TERMINATE_DO_LOOP ( ; )

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

bit

LB_LABEL_REQ /* Indicates whether label required on looping
  branch instruction. */

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

$ X := INFOBRMATION(CURRENT_NEST_LEVEL)

BRANCH8 := X[1,1]
BRANCH10 := X[2,1]
BRANCH11 := X[3,1]

LB_OPCODE := OPEBAND3(CURRENT_NEST_LEVEL)
 BLOCK_LABEL_PREFIX := '$'

if LB_LABEL_REQ
  then
    LABEL := BLOCK_LABEL_PREFIX || 'LPB'
  fi

/* Generate all looping instructions. */
docase BRANCHES of
  case ('W', 'U')
    generate ('B' II BLOCK_LABEL_PREFIX II BRANCH8 II '1')
  esac
  case (IB')
    generate ('B' II BLOCK_LABEL_PREFIX II 'BEG')
  esac
  case ('LI') /* Nothing to generate; fall through to looping branch. */
  esac

if LB_OPCODE = 'BCT'
  then
    generate (LABEL || 'BCT' II OPER1 II ',', II BLOCK_LABEL_PREFIX || X)
  else
    generate (LABEL || ' ' II LB_OPCODE II ',', II OPER2 || ',', II OPER1 || ',', II BLOCK_LABEL_PREFIX || X)
  fi

if LB_OPCODE = 'B'
  then
    X := 'BEG'
    X := X || '1'
  else
    X := BRANCH10 || '1'
  fi

if LB_OPCODE = 'ACT'
  then
    generate (LABEL || ' ACT ' || OPER1 || ',', || BLOCK_LABEL_PREFIX || X)
  else
    generate (LABEL || ' ' || LB_OPCODE || ' ', || OPER1 || ',', || OPER2 || ',', || BLOCK_LABEL_PREFIX || X)
  fi

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

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

59001. \textbf{MACRO} XHEX ( )
59002. \texttt{// Converts the integer in HEX_IN to a two-character hex string in}
59003. \texttt{HEX_OUT. */}
59004. \texttt{end I, J /* Temporaries. */}
59006. \texttt{call TRACE_PRINT ( ; 'XHEX')} 59007. \texttt{// Print macro name "XHEX" in case tracing on. */}
59008. \texttt{if HEX (1) \neq 0*}
59009. \texttt{then /* Hex array must be initialized. */}
59100. HEX (1) := *0 HEX (2) := *1 HEX (3) := *2 HEX (4) := *3
59104. end I
59105. I := HEX_IN/16
59106. J := HEX_IN - I*16 + 1
59107. HEX_OUT := HE X (I+1) || HE X (J)
59108. endd
AND_OU.TXT
ANY_COMP_CASE
ANY_ADD_CASE
ANY_SUB_CASE
ANY_SELDIFF_CASE
AZEND
AZEND_GEN
CASE
CASE_TAG
ECK_AREA_REQD
ELERD
BLOCK
BLOCK_LABEL_PREFIX
BLOCK_NAME
BLOCK_NAME_VALUE
BLOCK_NUMBER
BLOCK_TRACE_COUNTERS
BLOCK_TYPE
BLOCK_TYPE_VALUE
BRANCH_LABEL
BRANCH_TO_CASE1
BRANCH1
BRANCH2
BRANCH3
E10
E11
E8
CASE
CASE_ASSUMED_VECTOR_CASE
CASE Resort_CASE
CASE_COMPTEST_GEN
CASE_COUNTER
CASE_FORMAT
CASE_GEN_COMPARE
CASE_GET_DOCASE_INFO
CASE_LABEL_REQD
CASE_MISC_PROCESS
CASE_OCCURS
CASE_ECSIG_CASE
CASE_PROCESS_COMPARE_OPERANDS
CASE_PROCESS_VECTOR_OPERANDS
CASE_SET_NAMES
CASE_TRACE_COUNTER

defined *modified *parameter

C-119
CROSS-REFERENCE LISTING

LO_LABEL_BLOCK 21114. 21756. 21302.
DO_LOOPING_BRANCH_AND_FIRST_OF 21147. 21302.
DO_LOOPING_BRANCH_PROCESS 21302.
DO_SCAN_OPERANDS 21090. 21127.
DO_SET_FORMAT 21173. 21146.
DO_TRACE_COUNTERS 21119. 21940.
DO_UNTIL_POSTPROCESS 21621. 21738.
DO_UNTIL_PREPROCESS 21556. 21875.
DOCASE 21001.
DOCASE_DEBUG_STUFF 31071. 31126.
DOCASE_EXTRACT_OPERANDS 31034. 31102.
DOCASE_GENERAL_SETUP 31065. 31107.
DOCASE_INDEX_TO_REG 31059. 31149.
DOCASE_INFO_SAVE 31073. 31161.
DYNAMIC_SAVEAREA n30201. 83090. +83164. 83264.
ELSE *11001. 11024. 11123. u13001.
ELSE_BLOCK_NO u13008. *13042. 13044.
END_LABEL_VALUE 11025. 11057. 11094. *93008. 93039.
EQUAL_TEST_OUTSTANDING a33022. *33199. 33205. +33211. 33245. +33312.
+33338. *33414. 33515. 37052. 37112. +371911.
*37190. 91011. 91057. 83067. +91010. +91025.
*91047. +91082. -91027. +96010. +96016. +96053. +96068.
96059. 97054. +97066. +97150. +97118. +97133. +97164.
ESAC n35001. 43043.
ESACDD u37001. 43046.
ESACD_DBVCT_GEN 37177. 37226.
ESACD/general_case_choice 37062. 37105.
ESACD_GENERAL_CASE_INFO 37109. 37186.
ESACD_GENERAL_SYMBOLONLY 37125. 37135.
ESACD_INFO_UNPACK 37056. 37090.
ESACD_CUT_OF_RANGE_CHECK 37173. 37202.
ESACD_SELDEF_GEN 37116. 37151.
EXIT_FINDD 11130. 55010. +57001.
EXIT_LABEL u11016. *11138. 11221. 11236. 11241.
EXIT_SEVERITY n 160. +81136. 83499. +83501. 83503. 92015. +92017.
EXIT_TARGET 92019.
EXIT_TARGETS 94006. 55010. 55040. 90606.
EXIT_FEATURED_CONDITION 94006. 94046.
EXIT_FEATURED_LABEL_USED a 256. 11113. *11155. 11196. 21627. +21704. +21749.
FILL a33014. 35002. *97247. 11072. 11506. 43027.
FILL_LABEL_EXPR a23001. 23005. +23053. 23067. u25018. +25047. +25061.
FINAL 25072. 27007. 27012. 27039.
FIXED n35001. 81027. +81424. 81465. 81469. 81470. +81493.
FIRST u81500. 81504. 81781. 81051.
FIRST_ID a23009. 23164. +23165. 23167. +23171. +21711. +21719. 97001.
FIRST_INDEX a 247. +11173. +21696. +21683. +33503. 97047.
FIRST_PROC a81010. +81318. 81384. 81725. 81746. 81069. 81966.
FIRST_PROC_SAVED a83033. +83152. 83168. 83170. 83191.
FIRST_PROCESS_VALUEKNOWN a83026. +83195. 83208. 83395.
FIRST_PROCESS_VALUEKNOWN a83013. +83178. 83180. 83168. 83349. 83363. 3302.
FIRST_VELOCITY_VALUEKNOWN a81012. +81482. +81487. +81506. 81511. 81599. 81958.
FWD_FIR 81836. +81851. 81855. 81856. +83441. +83459. +83461.
GCASE_INDEXLEVEL a 233. +31201. 31201. 31202. 31204. 31206. 31208.
GCASE_INDEXLEVEL n 233. +31202. 33247. 33250. 33275. 33367. 33400. 33402.
GCASE_INDEXLEVEL a 236. +31222. 33247. 33250. 33705. 33706. 33706. 31787. 31718.
GEXEC_OFFSET_STRING a83013. +83158. +83161. 83366. 83370. 83406. 83410.
GEXEC_OFFSET_STRING a33400. +33442. +33444. +33446. +33449. +33450.
EXIT undefined *Modified *Parameter
<table>
<thead>
<tr>
<th>ID</th>
<th><em>ID</em></th>
<th>81001</th>
<th>81362</th>
<th>81369</th>
<th>81403</th>
<th>81406</th>
<th>81408</th>
<th>81410</th>
</tr>
</thead>
<tbody>
<tr>
<td>a11001</td>
<td>a11001</td>
<td>11044</td>
<td>a11076</td>
<td>11064</td>
<td>a11026</td>
<td>11057</td>
<td>a11163</td>
<td>11090</td>
</tr>
<tr>
<td>IF</td>
<td>IF_ASMotch_BANCH</td>
<td>37320</td>
<td>37326</td>
<td>37320</td>
<td>37320</td>
<td>37322</td>
<td>37322</td>
<td>37322</td>
</tr>
<tr>
<td>INDEX</td>
<td>INDEX_Addr</td>
<td>31031</td>
<td>31115</td>
<td>31115</td>
<td>31115</td>
<td>31115</td>
<td>31115</td>
<td>31115</td>
</tr>
<tr>
<td>INDEX_LENGTH</td>
<td>31018</td>
<td>31123</td>
<td>31123</td>
<td>31123</td>
<td>31123</td>
<td>31123</td>
<td>31123</td>
<td></td>
</tr>
<tr>
<td>INDEX_RANGE_FLAG</td>
<td>33457</td>
<td>33457</td>
<td>33457</td>
<td>33457</td>
<td>33457</td>
<td>33457</td>
<td>33457</td>
<td></td>
</tr>
<tr>
<td>INDEX_REG</td>
<td>INDEX_TYPE</td>
<td>31016</td>
<td>31128</td>
<td>31128</td>
<td>31128</td>
<td>31128</td>
<td>31128</td>
<td>31128</td>
</tr>
<tr>
<td>INFO</td>
<td>INFO</td>
<td>23017</td>
<td>23034</td>
<td>23034</td>
<td>23034</td>
<td>23034</td>
<td>23034</td>
<td>23034</td>
</tr>
<tr>
<td>INFORMATION</td>
<td>INFORMATION_VALUE</td>
<td>221</td>
<td>221</td>
<td>221</td>
<td>221</td>
<td>221</td>
<td>221</td>
<td>221</td>
</tr>
<tr>
<td>INLINING_SAVEAREA</td>
<td>INLINING_SAVEAREA</td>
<td>81614</td>
<td>81614</td>
<td>81614</td>
<td>81614</td>
<td>81614</td>
<td>81614</td>
<td>81614</td>
</tr>
<tr>
<td>INSERT</td>
<td>INSERT</td>
<td>33276</td>
<td>33276</td>
<td>33276</td>
<td>33276</td>
<td>33276</td>
<td>33276</td>
<td>33276</td>
</tr>
<tr>
<td>J</td>
<td>J</td>
<td>31192</td>
<td>31208</td>
<td>31208</td>
<td>31208</td>
<td>31208</td>
<td>31208</td>
<td>31208</td>
</tr>
<tr>
<td>LA_DEPTH</td>
<td>LABEL</td>
<td>11010</td>
<td>11010</td>
<td>11010</td>
<td>11010</td>
<td>11010</td>
<td>11010</td>
<td>11010</td>
</tr>
<tr>
<td><em>DEFINED</em></td>
<td><em>Modified</em></td>
<td><em>Parameter</em></td>
<td><em>Parameter</em></td>
<td><em>Parameter</em></td>
<td><em>Parameter</em></td>
<td><em>Parameter</em></td>
<td><em>Parameter</em></td>
<td><em>Parameter</em></td>
</tr>
</tbody>
</table>
PROC_TACIOL_SAVE_TYPE
PROC_DEFINFO_NQ_OSSAVE
PROC_EEstablish_BASE
PROC_GEN_LOCAL_SAVEAREA
PROC_GEN_OSSAVE_AREA
PROC_GEN_SAVEAREA
PROC_BEGAS
PROC_ID_BYTE
PROC_INFO_SAVE
PROC_MULTIUSEGEN
PROC_NAME
PROC_REG/save
PROC_SCAN_OPTIONS
PROC_SET_SAVE_INFO
PUSH_NEW_BLOCK
QUOTE
RANGE
RANGE_TEST_OUTSTANDING
RANGE_TEST_QUERY
RC
REG
REG_NAME
REG_TYPE
REGISTER
RESTORE_AREA
RESTORE
SAF
SAF
SAL
SAVE
SAVE.LENGTH
SAVE_TYPE
SAVE_REG
SAVEBACK_CHECK
SAVEDEBACKORFIRSTPROC
SAVEBACKVALUE
SECT
SIMPCMD_GET_MASK ORREL
SIMPLE_CONDITIONAL
SPECIAL_PREFIX
SPECIAL_OFFSET
SPLIT
TARGET
Z

*Defined
*Modified
*Parameter

C-125
APPENDIX D

DIAGNOSTIC MESSAGES

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

<table>
<thead>
<tr>
<th>Severity</th>
<th>Message-Number</th>
<th>Message-Text</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>STRC1101</td>
<td>EXIT= IGNORED WITH ASYNCH</td>
<td>Both the EXIT= and ASYNCH operands were specified, but are mutually exclusive; the EXIT= operand has been ignored.</td>
</tr>
<tr>
<td>8</td>
<td>STRC1102</td>
<td>REL= OR MASK= NOT IN PARENTHESES—IGNORED</td>
<td>The REL= and MASK= operands must be part of a simple conditional and thus must be inside parentheses. The keyword has been ignored.</td>
</tr>
<tr>
<td>8</td>
<td>STRC1103</td>
<td>EXIT= IGNORED WITH ELSE=</td>
<td>The EXIT= and ELSE= operands were both specified but are mutually exclusive. The EXIT= operand has been ignored.</td>
</tr>
<tr>
<td>8</td>
<td>STRC1301</td>
<td>ELSE=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>Explanation</td>
</tr>
<tr>
<td>----------</td>
<td>----------------</td>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>8</td>
<td>STRC1501</td>
<td>ELSE BLOCK elsename NOT FOUND</td>
<td>The operand ELSE=elsename was coded on the IF macro, but the FI has occurred before the ELSE occurred.</td>
</tr>
<tr>
<td>8</td>
<td>STRC2101</td>
<td>OPERANDS AFTER SECOND &quot;WHILE&quot; IGNORED</td>
<td>The keyword &quot;WHILE&quot; appears more than once in the DO's operands.</td>
</tr>
<tr>
<td>8</td>
<td>STRC2102</td>
<td>OPERANDS AFTER SECOND &quot;UNTIL&quot; IGNORED</td>
<td>The keyword &quot;UNTIL&quot; appears more than once in the DO's operands.</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>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>
</tr>
<tr>
<td>4</td>
<td>STRC2104</td>
<td>WARNING—LOOPING BRANCH MAY NOT BE EXECUTED ON EVERY ITERATION</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>
</tr>
<tr>
<td>8</td>
<td>STRC2105</td>
<td>TWO LOOPING BRANCHES INVALID IN &quot;DO&quot;—&quot;WHILE&quot; IGNORED</td>
<td>Both the WHILE and UNTIL looping groups contain looping branches (BCTs, BXHs, or BXLE,); the WHILE looping group has been 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>STRC2106</td>
<td>INVALID NUMBER OF OPERANDS FOR <em>opcode</em>&lt;br&gt;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>*** INVALID AFTER LOOPING BRANCH—&quot;AND&quot; INSERTED&lt;br&gt;The operand following the looping branch must be &quot;AND&quot; or &quot;OR&quot;; *** 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&lt;br&gt;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&lt;br&gt;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&lt;br&gt;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&lt;br&gt;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 <em>opcode</em>&lt;br&gt;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>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</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>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</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>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</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>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</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>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</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>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,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>it is assumed to be the address of the DOCASE index. If the range</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>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;,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>or omitted. xxx was found.</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>STRC3103</td>
<td>xxx INVALID THIRD OPERAND—IGNORED</td>
<td></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>
<td></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>
<td></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>
<td></td>
</tr>
<tr>
<td>8</td>
<td>STRC3301</td>
<td>&quot;CASE&quot; NOT IMMEDIATE DAUGHTER OF &quot;DOCASE&quot;</td>
<td></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>
<td></td>
</tr>
<tr>
<td>8</td>
<td>STRC3302</td>
<td>ASSUMING &quot;BLEND&quot; OMITTED—INSERTED</td>
<td></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>
<td></td>
</tr>
<tr>
<td>8</td>
<td>STRC3303</td>
<td>ASSUMING TWO &quot;BLENDS&quot; OMITTED—INSERTED</td>
<td></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>
<td></td>
</tr>
<tr>
<td>8</td>
<td>STRC3304</td>
<td>&quot;CASE&quot; TREATED AS &quot;BLOCK&quot; MACRO</td>
<td></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>
<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>Explanation</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. |

D-6
<table>
<thead>
<tr>
<th>Severity</th>
<th>Message-Number</th>
<th>Message-Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>STRC3311</td>
<td>MULTIPLE MISC CASES IN THIS DOCASE—THIS BLOCK IS DEAD CODE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>More than one &quot;CASE MISC&quot; has occurred in the same DOCASE. Only the first is executable.</td>
</tr>
<tr>
<td>8</td>
<td>STRC3312</td>
<td>xxx INVALID—ONLY FIRST TWO SUBOPERANDS PROCESSED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>STRC3313</td>
<td>CASE DEBUG ID=X'hh'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Debug block counts are being kept for this CASE macro. When executed, this block will store X'hh' into the last-case variable in the immediately surrounding DOCASE block.</td>
</tr>
<tr>
<td>8</td>
<td>STRC3701</td>
<td>DOCASE CONTAINS NO VALID CASES</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No valid CASE macros were found as immediate sub-blocks of the DOCASE.</td>
</tr>
<tr>
<td>8</td>
<td>STRC4301</td>
<td>NO BLOCKS ACTIVE—&quot;BLEND&quot; IGNORED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A BLEND macro was coded but no blocks were active (the current nest level was zero). The macro has been ignored.</td>
</tr>
<tr>
<td>8</td>
<td>STRC4302</td>
<td>NO BLOCK ACTIVE NAMED xxx— &quot;BLEND&quot; IGNORED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A BLEND macro was issued for block xxx, but no block by that name is active.</td>
</tr>
<tr>
<td>8</td>
<td>STRC5301</td>
<td>BLEND OF biname ASSUMED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The block biname has not yet been terminated; a BLEND is being issued for it by the FINAL macro.</td>
</tr>
</tbody>
</table>
Severity | Message-Number | Message-Text                                      | Explanation |
----------|----------------|--------------------------------------------------|-------------|
8         | STRC8101       | LINKAGE=xxx INVALID—"OS" ASSUMED                | The first suboperand of the LINKAGE= keyword is invalid; "OS" has been substituted. |
8         | STRC8102       | SECOND LINKAGE OPERAND IGNORED                  | The second suboperand of the LINKAGE= keyword must be either "CSECT" or omitted. It is invalid and has been ignored. |
4         | STRC8103       | WARNING—SAVETRACE REQUIRES "FINAL" MACRO        | The SAVETRACE debug option has been specified on the first PROC; warning is printed to indicate need for FINAL macro. |
8         | STRC8104       | DEBUG=xxx INVALID—IGNORED                       | An invalid debug specification is present; the first eight characters of the invalid operand are listed as xxx. That option is ignored. |
8         | STRC8105       | SAVETRACE MUST BE SPECIFIED ON FIRST PROC        | The SAVETRACE debug option must be enabled on the first PROC. The operand has been ignored. |
8         | STRC8106       | SAVETRACE REQUIRES FIRST PROC TO BE LINKAGE=OS  | The SAVETRACE debug option is valid only if the first PROC includes the LINKAGE=OS specification. The operand has been ignored. |
4         | STRC8107       | REG 1 MUST BE AMONG THOSE SAVED                 | Register 1 was destroyed during the GETMAIN for a dynamic save area and registers 14 through 12 were not specified (or defaulted) as being saved and WORK=NONE was specified. No further check is made to assure register one was among those saved,
<table>
<thead>
<tr>
<th>Severity</th>
<th>Message-Number</th>
<th>Message-Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>STRC8107</td>
<td>but the restore has been issued. If register 1 was not among those saved, its value will be undefined. (cont.)</td>
</tr>
<tr>
<td></td>
<td>STRC8108</td>
<td>PROC proc-name, DEBUG ID=X'hh'</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>
</tr>
<tr>
<td>8</td>
<td>STRC8109</td>
<td>REGISTER 13 IS INVALID—IGNORED</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>
</tr>
<tr>
<td>8</td>
<td>STRC8301</td>
<td>NO REGISTERS SAVED—RESTORE IGNORED</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>
</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>
</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>
</tr>
<tr>
<td>0</td>
<td>STRC8303</td>
<td>ONE OR MORE EXIT'S REFERENCE THIS POINT</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>
</tr>
<tr>
<td>Severity</td>
<td>Message-Number</td>
<td>Message-Text</td>
</tr>
<tr>
<td>----------</td>
<td>----------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>8</td>
<td>STRC9103</td>
<td>EXIT TO IMMEDIATELY SURROUNDING BLOCK INVALID</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The block name specified as the EXIT target is the block immediately surrounding the EXIT.</td>
</tr>
<tr>
<td>8</td>
<td>STRC9104</td>
<td>EXIT TO DO BLOCK INVALID WITHIN ATEND OR ONEXIT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>An EXIT macro specifies (or implies) a DO block as its target, but the EXIT macro is nested within the ATEND or ONEXIT segment of the DO. The macro is ignored.</td>
</tr>
<tr>
<td>0</td>
<td>STRC9201</td>
<td>ONE OR MORE EXIT'S REFERENCE THIS POINT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This block is the target of one or more EXIT macros (or EXIT= operands of IF macros). The severity code of this message may be modified by specifying the EXIT= operand of a PROC macro.</td>
</tr>
<tr>
<td>12</td>
<td>STRC9301</td>
<td>BLOCK NESTING LIMIT OF limit EXCEEDED—MACROS MUST BE MODIFIED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The current static nesting level has exceeded the stack limit in the macros of limit. Either the program or macros must be modified.</td>
</tr>
<tr>
<td>8</td>
<td>STRC9302</td>
<td>NON-CASE BLOCK IMMEDIATELY SURROUNDED BY DOCASE INVALID</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The block being defined is not a CASE block but is immediately surrounded by a DOCASE block. The result is undefined.</td>
</tr>
<tr>
<td>8</td>
<td>STRC9401</td>
<td>INSUFFICIENT OPERANDS FOR TEST &quot;opcode&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The parenthesized list which is supposed to be a simple conditional contains two items, the operation code opcode and a bc-spec.</td>
</tr>
<tr>
<td>Severity</td>
<td>Message-Number</td>
<td>Explanation</td>
</tr>
<tr>
<td>----------</td>
<td>----------------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| 8        | STRC9402       | SUPERFLUOUS OPERANDS FOR TEST "opcode"  
The parenthesized list which is supposed to be a simple conditional contains more than 5 items. |
| 8        | STRC9403       | NO CONDITION SPECIFIED—"MASK=0" ASSUMED  
A void simple conditional has been specified. |
| 8        | STRC9601       | ONE BLEND ASSUMED TO GET TO "type" BLOCK  
In a block terminating macro (such as IF), the current block was not of the corresponding type (such as IF), but the surrounding block is of the proper type. One BLEND has been inserted. |
| 8        | STRC9602       | TWO BLENDS ASSUMED TO GET TO "type" BLOCK  
In a block terminating macro (such as IF), the current block was not of the corresponding type (such as IF), but the second surrounding block is of the proper type. Two BLENDS have been inserted. |
| 8        | STRC9603       | CURRENT BLOCK IS NOT "type" BLOCK—MACRO IGNORED  
In a block terminating macro (such as IF), the current block was not of the corresponding type (such as IF). |
| 8        | STRC9604       | NO ACTIVE BLOCK NAMED "blname"  
The request to terminate block blname has been ignored because no block named blname is in the nest. |
<table>
<thead>
<tr>
<th>Severity</th>
<th>Message-Number</th>
<th>Message-Text</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>8</strong></td>
<td>STRC9605</td>
<td>BLOCK &quot;blname&quot; IS NOT A type BLOCK—MACRO IGNORED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The request to terminate a block named blname of type type has been ignored because the indicated block is of a different type.</td>
</tr>
<tr>
<td><strong>8</strong></td>
<td>STRC9606</td>
<td>END OF BLOCK &quot;blname1&quot; IMPLIES END OF BLOCK &quot;blname2&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Request to terminate block blname1 must first terminate block blname2 which is nested inside block blname1.</td>
</tr>
<tr>
<td><strong>8</strong></td>
<td>STRC9607</td>
<td>NO BLOCKS ACTIVE—MACRO IGNORED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The request to terminate a block has been ignored since no blocks are outstanding.</td>
</tr>
<tr>
<td><strong>8</strong></td>
<td>STRC9701</td>
<td>INSUFFICIENT OPERANDS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The conditional expression ends with a logical operation (AND or OR) outstanding.</td>
</tr>
<tr>
<td><strong>8</strong></td>
<td>STRC9702</td>
<td>INSUFFICIENT BRACKETS</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 present.</td>
</tr>
<tr>
<td><strong>8</strong></td>
<td>STRC9703</td>
<td>SYNTAX ERROR—LOOKING FOR &quot;AND&quot; OR &quot;OR&quot;, FOUND xxx</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The operand xxx was found where a logical operator was expected.</td>
</tr>
<tr>
<td><strong>8</strong></td>
<td>STRC9704</td>
<td>SYNTAX ERROR—xxx INVALID</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Invalid operand xxx in conditional expression.</td>
</tr>
<tr>
<td><strong>8</strong></td>
<td>STRC9705</td>
<td>SUPERFLUOUS BRACKET IGNORED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>More right brackets (&quot;&lt;&quot; or &quot;/&quot;) have been found in the conditional expression than left brackets (&quot;&gt;&quot; or &quot;+&quot;).</td>
</tr>
<tr>
<td>Severity</td>
<td>Message-Number</td>
<td>Message-Text</td>
</tr>
<tr>
<td>----------</td>
<td>----------------</td>
<td>---------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>STRC9706</td>
<td>SYNTAX ERROR—LOOKING FOR SIMPLE CONDITIONAL, FOUND &quot;xxx&quot;</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>The operand xxx was found in a conditional expression where a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>simple conditional sublist was expected.</td>
</tr>
<tr>
<td>*</td>
<td>STRC9902</td>
<td>START OF BLOCK nn (blname) AT DEPTH level</td>
</tr>
<tr>
<td></td>
<td></td>
<td>In response to the debug option LISTBLOCKS, the message</td>
</tr>
<tr>
<td></td>
<td></td>
<td>indicates the start of the block whose sequential number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>is nn. The block name is blname; if no name was specified</td>
</tr>
<tr>
<td></td>
<td></td>
<td>on the block initiation macro, blname is an internal name</td>
</tr>
<tr>
<td></td>
<td></td>
<td>of the form BLKnn. The current static nest level is level.</td>
</tr>
<tr>
<td>*</td>
<td>STRC9903</td>
<td>END OF BLOCK nn (blname) AT DEPTH level</td>
</tr>
<tr>
<td></td>
<td></td>
<td>In response to the debug option LISTBLOCKS, the message</td>
</tr>
<tr>
<td></td>
<td></td>
<td>indicates the end of the block whose sequential number is</td>
</tr>
<tr>
<td></td>
<td></td>
<td>nn. The block name is blname; if no name was specified</td>
</tr>
<tr>
<td></td>
<td></td>
<td>on the block initiation macro, blname is an internal name</td>
</tr>
<tr>
<td></td>
<td></td>
<td>of the form BLKnn. The current static nest level is level.</td>
</tr>
</tbody>
</table>
APPENDIX E

INSTALLING THE STRCMACS

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

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

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

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

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

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

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

The fourth data set is the assembly language source of a program called RUFDRAFT. It is provided for those installations which do not have a TN (upper and lower case) print chain available. RUFDRAFT will translate the SIMPL-M listings of data set three for printing on HN, PN, or QN print trains. For instructions on using RUFDRAFT, see the comments in the beginning of the source.
When requesting a copy of the STRCMACS, it would be appreciated if you would enclose a tape—our supply of DTR's is limited.

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

\[ \mathcal{E} - 3 \]
<table>
<thead>
<tr>
<th>Term</th>
<th>Page 1</th>
<th>Page 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESACOD</td>
<td>23, A-20</td>
<td></td>
</tr>
<tr>
<td>evoking procs</td>
<td>36-38</td>
<td></td>
</tr>
<tr>
<td>EXIT</td>
<td>27, A-21</td>
<td></td>
</tr>
<tr>
<td>exit severity</td>
<td>38</td>
<td></td>
</tr>
<tr>
<td>FI</td>
<td>18, A-22</td>
<td></td>
</tr>
<tr>
<td>FINAL</td>
<td>38, A-23</td>
<td></td>
</tr>
<tr>
<td>FOREVER</td>
<td>28, A-12</td>
<td></td>
</tr>
<tr>
<td>FORTRAN</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>GEDANKEN</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>goto controversy</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>goto-less programming</td>
<td>1, 2-10</td>
<td></td>
</tr>
<tr>
<td>IF</td>
<td>18, 28, A-24</td>
<td></td>
</tr>
<tr>
<td>IFANY</td>
<td>27, A-15</td>
<td></td>
</tr>
<tr>
<td>IFEND</td>
<td>19, A-26</td>
<td></td>
</tr>
<tr>
<td>immediately surrounding</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>in-line identifier</td>
<td>34, A-31</td>
<td></td>
</tr>
<tr>
<td>infinite loop</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>installing STRCMACS</td>
<td>E-1</td>
<td></td>
</tr>
<tr>
<td>ISWIM</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>iteration</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>labels</td>
<td>42</td>
<td></td>
</tr>
<tr>
<td>link register</td>
<td>36, A-11</td>
<td></td>
</tr>
<tr>
<td>linkage of procs</td>
<td>36-38</td>
<td></td>
</tr>
<tr>
<td>LISP</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>literals</td>
<td>26, 42</td>
<td></td>
</tr>
<tr>
<td>loop branches</td>
<td>21, A-13</td>
<td></td>
</tr>
<tr>
<td>looping group</td>
<td>A-13</td>
<td></td>
</tr>
<tr>
<td>messages</td>
<td>D-1</td>
<td></td>
</tr>
<tr>
<td>MISC</td>
<td>23, A-7</td>
<td></td>
</tr>
<tr>
<td>miscellaneous case</td>
<td>23, A-7</td>
<td></td>
</tr>
<tr>
<td>modifying structured programs</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>modularity</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>modules</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>multiple decisions</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>nest level</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>nesting blocks</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>OD</td>
<td>21, A-27</td>
<td></td>
</tr>
<tr>
<td>ONEND</td>
<td>30, A-28</td>
<td></td>
</tr>
<tr>
<td>ONEXIT</td>
<td>29, A-29</td>
<td></td>
</tr>
<tr>
<td>ONLY</td>
<td>27, A-15</td>
<td></td>
</tr>
<tr>
<td>operand format for DO</td>
<td>B-7</td>
<td></td>
</tr>
<tr>
<td>OREGANO</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>PL/I</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>PROC</td>
<td>31, A-30</td>
<td></td>
</tr>
<tr>
<td>PROCEND</td>
<td>39, A-35</td>
<td></td>
</tr>
<tr>
<td>properly nested</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>proving program</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>correctness</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>quasi-goto-less code</td>
<td>3, 8</td>
<td></td>
</tr>
<tr>
<td>Term</td>
<td>Page</td>
<td>Term</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-------------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>range option</td>
<td>27</td>
<td>SIMPL-M (con't)</td>
</tr>
<tr>
<td>recursive code</td>
<td>34, 42</td>
<td>generate</td>
</tr>
<tr>
<td>reentrant code</td>
<td>34, 42</td>
<td>implied relations</td>
</tr>
<tr>
<td>register save areas</td>
<td>32, A-31</td>
<td>initialization</td>
</tr>
<tr>
<td>user supplied</td>
<td>33</td>
<td>K'</td>
</tr>
<tr>
<td>dynamic</td>
<td>33</td>
<td>macros</td>
</tr>
<tr>
<td>length of</td>
<td>33</td>
<td>N'</td>
</tr>
<tr>
<td>CORPVALUES</td>
<td>41</td>
<td>procs</td>
</tr>
<tr>
<td>BACK</td>
<td>41</td>
<td>SYSLIST</td>
</tr>
<tr>
<td>restoring registers</td>
<td>32, A-10</td>
<td>T'</td>
</tr>
<tr>
<td>return address register</td>
<td>36, A-10</td>
<td>type conversion</td>
</tr>
<tr>
<td>return code</td>
<td>35, A-10</td>
<td>SIMPL-X</td>
</tr>
<tr>
<td>returning values</td>
<td>32, A-10</td>
<td>simple conditionals</td>
</tr>
<tr>
<td>RUF DRAFT</td>
<td>E-1</td>
<td>stepwise refinement</td>
</tr>
<tr>
<td>save areas</td>
<td>32, A-31</td>
<td>STRCMACS</td>
</tr>
<tr>
<td>user supplied</td>
<td>33</td>
<td>super-instructions</td>
</tr>
<tr>
<td>dynamic</td>
<td>33</td>
<td>surrounding</td>
</tr>
<tr>
<td>length of</td>
<td>33</td>
<td>top-down programming</td>
</tr>
<tr>
<td>CORPVALUES</td>
<td>41</td>
<td>see stepwise refinement</td>
</tr>
<tr>
<td>BACK</td>
<td>41</td>
<td>trace vector</td>
</tr>
<tr>
<td>SIMPL-M arrays</td>
<td>8, B-1</td>
<td>UNTIL</td>
</tr>
<tr>
<td>assignments</td>
<td>B-3</td>
<td>USING</td>
</tr>
<tr>
<td>comments</td>
<td>B-4</td>
<td>WHILE</td>
</tr>
<tr>
<td>data types</td>
<td>B-2</td>
<td>work register</td>
</tr>
</tbody>
</table>