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

C. WRANDLE BARTH

JANUARY 1974

GODDARD SPACE FLIGHT CENTER
GREENBELT, MARYLAND
NOTICE

THIS DOCUMENT HAS BEEN REPRODUCED FROM THE BEST COPY FURNISHED US BY THE SPONSORING AGENCY. ALTHOUGH IT IS RECOGNIZED THAT CERTAIN PORTIONS ARE ILLEGIBLE, IT IS BEING RELEASED IN THE INTEREST OF MAKING AVAILABLE AS MUCH INFORMATION AS POSSIBLE.
STRCMACS: AN EXTENSIVE SET OF MACROS
FOR STRUCTURED PROGRAMMING IN OS/360
ASSEMBLY LANGUAGE

C. Wrandle Barth

January 1974

GODDARD SPACE FLIGHT CENTER
Greenbelt, Maryland
## CONTENTS

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

### APPENDIX A — MACRO INSTRUCTION DESCRIPTIONS

<table>
<thead>
<tr>
<th>Macro Instruction</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conditional expressions</td>
<td>A-1</td>
</tr>
<tr>
<td>ATEND</td>
<td>A-3</td>
</tr>
<tr>
<td>ATEXIT</td>
<td>A-4</td>
</tr>
<tr>
<td>BLEND</td>
<td>A-5</td>
</tr>
<tr>
<td>BLOCK</td>
<td>A-6</td>
</tr>
<tr>
<td>CASE</td>
<td>A-7</td>
</tr>
<tr>
<td>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>DOEND</td>
<td>A-17</td>
</tr>
<tr>
<td>ELSE</td>
<td>A-18</td>
</tr>
<tr>
<td>ESAC</td>
<td>A-19</td>
</tr>
<tr>
<td>ESACOD</td>
<td>A-20</td>
</tr>
<tr>
<td>EXIT</td>
<td>A-21</td>
</tr>
<tr>
<td>FI</td>
<td>A-22</td>
</tr>
<tr>
<td>FINAL</td>
<td>A-23</td>
</tr>
</tbody>
</table>
CONTENTS (continued)

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

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

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

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

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

APPENDIX E — INSTALLING THE STRCMACS ................ E-1
STRCMACS: AN EXTENSIVE SET OF MACROS FOR STRUCTURED PROGRAMMING IN OS/360 ASSEMBLY LANGUAGE

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

We will discuss the two techniques which have been most often referred to as "structured programming." One is that of programming with high-level control structures (such as the \texttt{if} and \texttt{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:

```plaintext
if I < 5
    then
        J := 5;
    else
        J := I;
```
he would use:

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

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

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

---

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

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

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

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

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

IF (q) GO TO 20
IF (r) GO TO 10
GO TO 60
10  GO TO 60
20 IF (s) GO TO 80
GO TO 30
40 IF (u) GO TO 70
GO TO 50
50 IF (.NOT. (v)) GO TO 60
GO TO 110
90  GO TO 100
100 IF (.NOT. (x)) GO TO 110
GO TO 100
110 IF (.NOT. (w)) GO TO 120
GO TO 60
120 IF (.NOT. (v)) GO TO 60
GO TO 60

goto-less programming

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

RETURN

5
in goto-less form. Alan Perlis has referred to similar situations as the 'Turing Tarpit' in which everything is possible, but nothing is easy."¹

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

The main arguments for goto-less programming are:

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

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

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

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

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

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

```
OUTER: begin;

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

```
 begin;

 OUTER: begin;

 INNER: begin;

 if I ≠ 0 then
   begin;
     α
   end;
 end;

 if I ≠ 0 then
   begin;
     β
   end;

 γ
```

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

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

Such a method was used in the programming of the structured macros themselves. Since 360 macro assembly language contains no statement grouping capability nor any looping construct, the actual programming was done in an imaginary abstract programming language called SIMPL-M. This not only simplified the writing of the macros, but it also is the "source" language for documentation and certification purposes. The listing of the macros in Appendix
C is in SIMPL-M. This source was then hand-translated into macro assembly language in a straight-forward manner. Any changes or extensions are always made in both to assure the "source" is kept current.

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

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

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

An excellent discussion of these points is provided in "A Case for the goto" by Martin E. Hopkins. 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.

Hopkins, ibid., pp. 787-790.
Three classes of programming languages may be distinguished. In the first, only goto-less programming may be done since no goto is provided. This group includes such languages as (pure) LISP, ISWIM, BLISS, 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.

---

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," Wirth's "Program Development by Stepwise Refinement," and Hoare's "Notes on Data Structuring." We will give here only a basic overview of the topic and refer the reader to the above papers for more details and examples.

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

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

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

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

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

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

reliable if a number of levels are used. At the outermost level, the "what" to be accomplished is the "what" of A. But instead of moving directly to the "how" of B, we go to a "how" B₁ for some abstract super-machine with arbitrarily complex instructions. Most of the instructions of B₁ are not intelligible to our real computer. But the number of instructions are few (maybe 50 or so), so we can feel that, if there were a machine which could understand B₁, it would surely accomplish the task A. We can now take the instructions of B₁ (call them the A₂₁) and for whichever are not understandable to our real computer repeat the problem-solving process producing a program B₂₁—the "how" for each A₂₁ in more primitive terms. This process is continued until eventually all instructions are in terms intelligible to our computer.

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

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

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

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

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

EFFICIENCY OF STRUCTURED CODE

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

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

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

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

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

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

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

\[
\begin{align*}
\text{TRY} & \quad \text{IF} & \quad (\text{LTR, } 3, 3, Z) \\
& \quad L & \quad 1, \text{WORD} \\
& \quad LA & \quad 1, 1(1) \\
& \quad ST & \quad 1, \text{WORD} \\
\text{FI} & \quad \text{TRY}
\end{align*}
\]

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

\[
(\text{LTR, } 3, 3, Z) \\
(\text{LTR, } 3, 3, \text{REL}=Z) \\
(\text{LTR, } 3, 3, \text{MASK}=8) \quad \text{[The mask of a BC instruction.]}
\]

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

\[
\begin{align*}
\text{LTR} & \quad 3, 3 \\
\text{BNZ} & \quad \text{end-of-block} \quad \text{or} \quad \text{BC} & \quad 7, \text{end-of-block}
\end{align*}
\]

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

\[
\begin{align*}
\text{IF} & \quad (\text{T S, SPOT, MASK}=8) \\
\text{IF} & \quad (\text{CR, } 3, 4, \text{E}) \\
\text{IF} & \quad (\text{CLM, } 3, \text{X'C', BYTE, REL}=\text{E})
\end{align*}
\]
The following relations may be used:

- **H or GT (High)**
- **N H or LE (Not High)**
- **L or LT (Low)**
- **N L or GE (Not Low)**
- **E or EQ (Equal)**
- **NE (Not Equal)**
- **O (Ones or Overflow)**
- **NO (Not Ones or Not Overflow)**
- **P (Plus)**
- **NP (Not Plus)**
- **M (Minus)**
- **NM (Not Minus)**
- **Z (Zero or Zeros)**
- **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.,*
\((\text{CR}, 7, 5, E), \text{OR}, (\text{SR}, 3, 1, Z)\) is invalid. The symbols "+" and "/" may be used instead of "<" and ">" for those installations whose print chains will not print the latter. If brackets are omitted, the OR is treated as having higher precedence than the AND. (If the brackets were omitted in MORECHK above, the operation would be performed as "((\text{CR}, 7, 5, E), \text{OR}, <, (\text{SR}, 3, 1, Z)), \text{AND}, (\text{LTR}, 1, 1, \text{MASK}=\text{SYMMASK}), >"). Instructions which do more than just set the condition code (such as the SR above) may be used within conditional expressions. It should be realized, however, that such operations may not always be executed. In the MORECHK block above, register 1 will not be subtracted from register 3 if registers 7 and 5 are equal.

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

\begin{verbatim}
LIMIT IF (C,7,=F'100', H),ELSE=TRY0
L 7,=F'100'
TRY0 ELSE BLEND=LIMIT
IF (LTR,7,7,M)
SR 7,7
FI
FI TRY0
\end{verbatim}

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:

\begin{verbatim}
GET (IN,CARDAREA)
IF ASYNCH
INPUTEND OI FLAG, EOF
FI
.
.
IN DCB ..., EODAD=INPUTEND
\end{verbatim}
The asynchronous IF generates an unconditional branch around the block. Note that if a label occurs on the IF macro, it will be defined on the branch instruction. As a consequence, the label "INPUTEND" is specified on the first instruction inside the block rather than on the IF macro itself.

**Iteration**

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

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

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

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

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

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

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

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

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

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

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

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

The above loop will execute only four times.

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

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

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

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

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

Multiple decisions

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

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

Example:

```
UPDATE DOCASE REQWORD
ADD
CASE

ESAC
ADD
REPL
CASE

ESAC
CHANGE
CASE

ESAC
CASE

ESAC
ESACOD
```

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

One of the CASE macros (usually the last of the list) may have the operand "MISC" to indicate that it is to be executed only if no other block is appropriate...
(that is, if the index is less than one or greater than \( n \), in the form we have discussed so far). This miscellaneous block is not counted in locating the \( i \)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'C'
    ...
    ...
    ESAC
D  CASE FIVE,(FOURTEEN,SIXTEEN)
    ...
    ...
    ESAC
E  CASE
    ...
    ...
    ESAC
ESAC
ESACOD
```
FIVE EQU 5
FOURTEEN EQU X'E'
SIXTEEN EQU FIVE+11

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

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

\[
\text{DOCASE I} \quad \text{or Fullword} \\
\text{DOCASE (I,W)} \\
\text{DOCASE (I,H) Halfword} \\
\text{DOCASE (I,B) One byte} \\
\text{DOCASE (3)} \quad \text{or Register index} \\
\text{DOCASE (R3)}
\]

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

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

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

- Code "DOCASE I, SIMPLE" when each CASE block is for a single index value and those value are the numbers 1, 2, 3, \ldots, 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.

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

27
In this code segment, a delimiter other than the comma will cause abnormal termination of the control card scan loop after printing a message. Since the EXIT macro has no operand, the exit is to the end of the block containing the block containing the EXIT (the surrounding block whose nest level is one less than the current nest level; in our example, the block SCAN). Any surrounding block's name may be specified as the EXIT's operand to cause transfer to the end of that block. In our example, adding the operand DOINFILE to the EXIT would skip the rest of the input when the error occurred.

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

```
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)
   IF (CLC,ARG(8),0(2),EQ)
      A 1,8(2)
      ST 1,8(2)
      EXIT UPDATE
   FI
   L 2,12(2)
OD
MVC 0(8,2),ARG
ST 1,8(2)
BLEND UPDATE
```

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

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

The looping segment of the block is the IF and load instructions which follow the DO. If the loop terminated normally, (that is, because of the DO macro's
test), the ATEND code segment will be executed; if the loop terminates abnormally (due to some EXIT being executed for the DO), the ONEXIT segment will be executed. The flow chart below shows the relationship of the various blocks.

---

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

*Defining modules*

To aid stepwise refinement, it is desirable to have a simple method for defining modules which entails a minimum of execution overhead and provides a maximum of module independence. Such modules normally are called procs (for procedure), involve about a page of code, and invoke other procs via a calling sequence.
The macros PROC and CORP are provided to delineate such modules. There are two types of PROCs: the normal type involves minimal overhead (normally just the saving and restoring of registers) and is used for the majority of modules created during the stepwise refinements; the other involves standard OS linkage conventions and is usually used for the main proc of the CSECT or other places where the evoking routine is expecting OS linkage.

The simplest non-OS proc is coded as:

```
X PROC
.
.
CORP
```

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

```
BAL 14,X
```

instruction.

The basic form of the OS-linkage proc is:

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

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

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

```
SAVE=3
```
or

```
SAVE=(15,7)
```
to cause limited saving and a correspondingly smaller save area. These registers (as all registers specified for the STRCMACS) may be specified symbolically. For example:

```
PROC SAVE=(R5, LAST)

R5 EQU 5
LAST EQU 9
```

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

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

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

```
CORP RESTORE=reg
```

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

```
X PROC

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

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

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=(, , \text{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:

\[ \text{RENTPROC PROC LINKAGE=OS, SAVE}=(, , \text{DYNAM, WORKSIZE}) \]
\[ \text{USING WORKSECT, 13} \]

33
WORKSECT DSECT
DS 18F New save area.
.
.
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

\[
\text{X PROC LINKAGE=(OS,CSECT)}
\]

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

\[
\text{PROC WORK=5}
\]

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:

\[
\text{CORP RC=value}
\]
If the value to be returned is contained in a register, use:

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

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

The last instruction normally generated by a CORP is a

```
BR  14
```

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

```
CORP  LINK=linkreg
```

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

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

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

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

MACRO
BMAC
B PROC

CMAC

CORP B, LINK=NONE
MEND

MACRO
CMAC
PROC

CORP LINK=NONE
MEND

A PROC LINKAGE=(OS,CSECT)

BMAC

CORP

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

**Special services**

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

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

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

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

**STRCMACS debugging aids**

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

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

```
PROC DEBUG=(list of options)
```

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

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

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

We will now discuss each of the options in turn.

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

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

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

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

- X'FF' or X'FE': The PROC's main save area: The byte is initialized to X'FF'; it is set to X'FE' each time the proc is entered and is reset to X'FF' each time the proc is "finished" (each time it returns).
- X'FC': The CORPVALUES save area, for those procs in which the CORPVALUES option is turned on.
X'FB': The BACK save area for those procs in which the CORPVALUES option is turned on and in which one or more registers are returned.

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

Word one of the first OS save area contains the address of the trace vector (if PROCTRACK 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'

PROC LINKAGE=(73, CSERT), DEBUG=(ALL, NolistBlocks, Global)

:  

LA $5,2
DO UNTIL, (RCT, 3)
   BAL 14, SUBY
   BAL 14, SUBZ

*  :

CORP EJECT

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

CORP SUBX EJECT

SUBY PROC DEBUG=ndc, rovaljies
*  :
   *
   L 3, YID
   CALL NEXTPROG

*  :

CORP RETURN=3 EJECT

SUBZ PROC
*  :
   *
   L 3, ZID
   LR 6, 3
   IF (C, 5, =F'1', EQ)
   BAL 14, SUBY
   FI

*  :

CORP RETURN=6 EJECT FINAL

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

43
EXAMPLE OF DEBUG FACILITIES

LOC  OBJECT CODE  ADDR1  ADDR2  STAT  SOURCE STATEMENT

*** NOTE ***

2  Ourprog  proc  linkage=(oscsect),debug=(ALL,NOISTBLOCKS,GLOBAL)

4,  STICK03  WARNING—SAVETRACE  REQUIRES  "FINAL"  MACRO

4,  STICK03  proc  ourprog,  debug  id=x'01'

47F0  FOOC  0000C

7+  $P01AA  DC  a1(7),cl7'OURPROG'

131+ ourprog  cssect

47F0  FOOC  0000C

159+  using  *,15

245+  b  $PO1AA

287+  ourprog  proc  linkage=(oscsect),debug=(ALL,NOLISTBLOCKS,GLOBAL)  00004000

** NOTE  ***  3

4,  STRC8103  WARNING—SAVETRACE  REQUIRES  "FINAL"  MACRO

4,  STRC8108  proc  ourprog,  debug  id=x'01'

44  32BEG  Lh  1,$2DOL

48  32BEG  Lh  1,$2DOL

52  c'trace'

56  trace(256),trace+1  nvc  strace+256,X'01'

60  mvi  strace+256,X'01'

64  lh  2,sp01pct

68  la  2,1(2)

72  sth  1,s2dol

76  lh  1,$2DOL

80  la  1,1(1)

84  sth  1,s2dol

88  b  $2GO

92  mvi  strace+256,X'01'

96  lh  1,$2DTR

100  la  1,1(1)

104  sth  1,s2dtr

108  b  $2GO

112  c'trace',0h'0'

116  current  loop  count

120  overall  loop  count

124  $2DOL

128  $2DTR

132  current  loop  count

136  overall  loop  count

140  $2DOL

144  $2DTR

148  bct  5,$2BEG

152  cmp

156  l  13,$FIRSTSV

160  sr  15,15

164  st  16,16(13)

168  ln  14,12,12(13)

172  mov  12(13),x'FF'

176  be  14
## Example of Debug Facilities

<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJECT CODE</th>
<th>ADDR1</th>
<th>ADDR2</th>
<th>STM</th>
<th>SOURCE</th>
<th>STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>000156</td>
<td>47FC D1DA</td>
<td>0012E</td>
<td></td>
<td>61+</td>
<td>SUBX</td>
<td>PROC SAVE=(3,5)</td>
</tr>
<tr>
<td>000157</td>
<td>02BC D1F0</td>
<td>00204</td>
<td></td>
<td>62+</td>
<td>SUBX</td>
<td>B $P02AA</td>
</tr>
<tr>
<td>000158</td>
<td>47FC D22C</td>
<td>00249</td>
<td></td>
<td>63+</td>
<td>SUBX</td>
<td>DC CL4 'SUBX'</td>
</tr>
<tr>
<td>000159</td>
<td>D22C</td>
<td></td>
<td></td>
<td>64+</td>
<td></td>
<td>STX 14,12,$P02SV+12</td>
</tr>
<tr>
<td>000204</td>
<td>0000000000000270</td>
<td></td>
<td></td>
<td>65+</td>
<td></td>
<td>$P02DD</td>
</tr>
<tr>
<td>000240</td>
<td>5810 D49C</td>
<td>00480</td>
<td></td>
<td>66+</td>
<td></td>
<td>OP</td>
</tr>
<tr>
<td>000244</td>
<td>D2PF 1000 1001 0000 00001</td>
<td></td>
<td></td>
<td>67+</td>
<td></td>
<td>DC X'FF020000' FLAG (PF=FINISHED,FE=ENTERED), ID, COUNT</td>
</tr>
<tr>
<td>00024A</td>
<td>9202 1100</td>
<td>00100</td>
<td></td>
<td>68+</td>
<td></td>
<td>$P02SV</td>
</tr>
<tr>
<td>00024B</td>
<td>4310 D155</td>
<td>001FA</td>
<td></td>
<td>69+</td>
<td></td>
<td>$P02SV EQU $P02SV</td>
</tr>
<tr>
<td>000252</td>
<td>4111 0001</td>
<td>00001</td>
<td></td>
<td>70+</td>
<td></td>
<td>DC (15) 'F0'</td>
</tr>
<tr>
<td>000256</td>
<td>4010 D155</td>
<td>001FA</td>
<td></td>
<td>71+</td>
<td></td>
<td>EQU $P02SV</td>
</tr>
<tr>
<td>00025A</td>
<td>D1E4</td>
<td>001F8</td>
<td></td>
<td>72+</td>
<td></td>
<td>DC A(O,$P02NXT)</td>
</tr>
<tr>
<td>00025B</td>
<td>5830 D484</td>
<td>00498</td>
<td></td>
<td>73+</td>
<td></td>
<td>MVC O(256,1),1(1)</td>
</tr>
<tr>
<td>000262</td>
<td>90BC D260</td>
<td>0027C</td>
<td></td>
<td>74+</td>
<td></td>
<td>MVI 256(1),X'02'</td>
</tr>
<tr>
<td>000266</td>
<td>D1E4</td>
<td>001F8</td>
<td></td>
<td>75+</td>
<td></td>
<td>LA 1,$P02SV+2</td>
</tr>
<tr>
<td>00026A</td>
<td>9835 D204</td>
<td>0021B</td>
<td></td>
<td>76+</td>
<td></td>
<td>STH 1,$P02SV+2</td>
</tr>
<tr>
<td>00026E</td>
<td>07EF</td>
<td></td>
<td></td>
<td>77+</td>
<td></td>
<td>MVNI $P02SV, X'FE'</td>
</tr>
<tr>
<td>000270</td>
<td>D1E4</td>
<td>001F8</td>
<td></td>
<td>78+</td>
<td></td>
<td>DC A($P02SV,$P02FWD)</td>
</tr>
<tr>
<td>000274</td>
<td>000001D20000002C3</td>
<td></td>
<td></td>
<td>79+</td>
<td></td>
<td>L 3,XID</td>
</tr>
<tr>
<td>00027C</td>
<td>0000000000000000</td>
<td></td>
<td></td>
<td>80+</td>
<td></td>
<td>L 3,XID</td>
</tr>
<tr>
<td>00027E</td>
<td>07EF</td>
<td></td>
<td></td>
<td>81+</td>
<td></td>
<td>L 3,XID</td>
</tr>
<tr>
<td>00027E</td>
<td>07EF</td>
<td></td>
<td></td>
<td>82+</td>
<td></td>
<td>L 3,XID</td>
</tr>
<tr>
<td>00027E</td>
<td>07EF</td>
<td></td>
<td></td>
<td>83+</td>
<td></td>
<td>L 3,XID</td>
</tr>
<tr>
<td>00027E</td>
<td>07EF</td>
<td></td>
<td></td>
<td>84+</td>
<td></td>
<td>L 3,XID</td>
</tr>
<tr>
<td>00027E</td>
<td>07EF</td>
<td></td>
<td></td>
<td>85+</td>
<td></td>
<td>L 3,XID</td>
</tr>
<tr>
<td>00027E</td>
<td>07EF</td>
<td></td>
<td></td>
<td>86+</td>
<td></td>
<td>L 3,XID</td>
</tr>
</tbody>
</table>

23 JAN 74
### Example of Debug Facilities

<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJECT CODE</th>
<th>ADDR1</th>
<th>ADDR2</th>
<th>STATE</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>000233</td>
<td>47F0 D2AC</td>
<td>042C</td>
<td>0</td>
<td></td>
<td>PROC DEBUG=NOCORPVALUES</td>
</tr>
<tr>
<td>000234</td>
<td>82FC2B8</td>
<td>042C</td>
<td>97+SUBY</td>
<td>B</td>
<td>$P03AA</td>
</tr>
<tr>
<td>000235</td>
<td>902C D2C0</td>
<td>0324</td>
<td>9+P03AA</td>
<td>STM 14,12,$P03SV+12</td>
<td></td>
</tr>
<tr>
<td>000236</td>
<td>47F0 D2FC</td>
<td>0034</td>
<td>100+</td>
<td>B</td>
<td>$P03DD</td>
</tr>
<tr>
<td>000237</td>
<td>FF030000</td>
<td>102+</td>
<td>DC</td>
<td>X'FF030000' FLAG (FF=FINISHED, FE=ENTERED), ID, COUNT</td>
<td></td>
</tr>
<tr>
<td>000238</td>
<td>FF030000</td>
<td>103+</td>
<td>$P03PD</td>
<td>EQU</td>
<td>$P03SV</td>
</tr>
<tr>
<td>000239</td>
<td>0000000000036D</td>
<td>104+</td>
<td>DC</td>
<td>A($P03RP,$P03NX)</td>
<td></td>
</tr>
<tr>
<td>000240</td>
<td>5810 D49C</td>
<td>0049C</td>
<td>105+</td>
<td>DC</td>
<td>(15)F^*</td>
</tr>
<tr>
<td>000311</td>
<td>47F0 D2B6</td>
<td>0024</td>
<td>106+</td>
<td>B</td>
<td>SPO3DD</td>
</tr>
<tr>
<td>000312</td>
<td>5810 D45C</td>
<td>00480</td>
<td>107+</td>
<td>DC</td>
<td>X'FF030000' FLAG (FF=FINISHED, FE=ENTERED), ID, COUNT</td>
</tr>
<tr>
<td>000313</td>
<td>9203 1000</td>
<td>00130</td>
<td>108+</td>
<td>TF</td>
<td>X'FF030000'</td>
</tr>
<tr>
<td>000314</td>
<td>4810 D2B6</td>
<td>002CA</td>
<td>109+</td>
<td>LA</td>
<td>1, $P03SV+2</td>
</tr>
<tr>
<td>000315</td>
<td>1111 0001</td>
<td>00201</td>
<td>110+</td>
<td>LA</td>
<td>1, 1(1)</td>
</tr>
<tr>
<td>000316</td>
<td>4010 D2B6</td>
<td>002CA</td>
<td>111+</td>
<td>STH</td>
<td>1,$P03SV+2</td>
</tr>
<tr>
<td>000317</td>
<td>9203 D2B4</td>
<td>002C8</td>
<td>112+</td>
<td>MVI</td>
<td>$P03SV, X'FF'</td>
</tr>
<tr>
<td>000318</td>
<td>5630 D48B</td>
<td>0049C</td>
<td>113+</td>
<td>:</td>
<td></td>
</tr>
<tr>
<td>000319</td>
<td>0700</td>
<td>114+</td>
<td>L</td>
<td>3,YID</td>
<td></td>
</tr>
<tr>
<td>000320</td>
<td>0000000000036D</td>
<td>115+</td>
<td>DC</td>
<td>A($P03RP,$P03NX)</td>
<td></td>
</tr>
<tr>
<td>000321</td>
<td>5810 D49C</td>
<td>0049C</td>
<td>116+</td>
<td>CALL</td>
<td>NEXTPROG</td>
</tr>
<tr>
<td>000322</td>
<td>9203 D2B6</td>
<td>0034C</td>
<td>117+</td>
<td>CNOP</td>
<td>0,4</td>
</tr>
<tr>
<td>000323</td>
<td>00000000</td>
<td>0034C</td>
<td>118+</td>
<td>B</td>
<td>#8 BRANCH AROUND VCON</td>
</tr>
<tr>
<td>000324</td>
<td>00000000</td>
<td>0034C</td>
<td>119+</td>
<td>DC</td>
<td>V(NEXTPROG) ENTRY POINT ADDRESS</td>
</tr>
<tr>
<td>000325</td>
<td>00000000</td>
<td>0034C</td>
<td>120+</td>
<td>L</td>
<td>15, INBO0040B LOAD 15 WITH ENTRY ADDR</td>
</tr>
<tr>
<td>000326</td>
<td>050F</td>
<td>00338</td>
<td>121+</td>
<td>BALR</td>
<td>14, 15 BRANCH TO ENTRY POINT</td>
</tr>
<tr>
<td>000327</td>
<td>5030 D2D4</td>
<td>00228</td>
<td>122+</td>
<td>:</td>
<td></td>
</tr>
<tr>
<td>000328</td>
<td>92FF D2B4</td>
<td>00228</td>
<td>123+</td>
<td>:</td>
<td></td>
</tr>
<tr>
<td>000329</td>
<td>92FF D2B4</td>
<td>00228</td>
<td>124+</td>
<td>CORD</td>
<td>RETURN=3</td>
</tr>
<tr>
<td>000330</td>
<td>98EC D2C0</td>
<td>00204</td>
<td>125+</td>
<td>ST</td>
<td>3,$P03SV+32</td>
</tr>
<tr>
<td>000331</td>
<td>97FF D2C0</td>
<td>00204</td>
<td>126+</td>
<td>MVI</td>
<td>$P03SV, X'FF'</td>
</tr>
<tr>
<td>000332</td>
<td>97FF D2C0</td>
<td>00204</td>
<td>127+</td>
<td>LM</td>
<td>14, 12,$P03SV+12</td>
</tr>
<tr>
<td>000333</td>
<td>0700</td>
<td>00204</td>
<td>128+</td>
<td>BR</td>
<td>14</td>
</tr>
</tbody>
</table>

---

23 Jan 74

00024000
EXAMPLE OF DEBUG FACILITIES

LOC OBJECT CODE ADDR1 ADDR2 STMT SOURCE STATEMENT

130 SUBZ PROC *
131*, STRC0106 PROC SUBZ, DEBUG ID=X'04'
132*SUBZ  B $PO4AA
133*DC CL4'SUBZ'
134*STM 14,12,$PO4SV+12
135*  B $PO4DD
136*PO4SV+12', FLAG (FF=FINISHED, PE=ENTERED), ID, COUNT
137*DC X'PO400000', ID, COUNT
138*PO3XNT SQU $PO4SV
139*DC A($PO4SV,$PO3XNT)
140*DC (15)'F'
141*PO4DD L 1,=[A($STR_POS)]
142*PO4DD MVC 0(256,1),1(1)
143*PO4DD NVI 256(1),X'ON'
144*PO4DD IN 1,$PO4SV+2
145*PO4DD STM 14,12,$PO4CRP+12
146*PO4DD L 3,ZID 00036000
147*PO4DD NVI $PO4SV,X'FE'
148*PO4DD MVI A($PO4BCK+12*4,$PO4SV+12
149*PO4DD ST 6,$PO4BCK+44
150*PO4DD MVI $PO4SV,X'FC040000'
151*PO4DD BR 14
152*PO4DD DC X'FB040000', ID, COUNT
153*PO4DD DC X'FD040000'
154*PO4DD DC X'FC040000'
155*PO4DD DC X'FB040000'
156*PO4DD DC X'FC040000'
157*PO4DD DC X'FB040000'
158*PO4DD DC X'FB040000'
159*PO4DD DC X'FB040000'
160*PO4DD DC X'FB040000'
161*PO4DD DC X'FB040000'
162*PO4DD DC X'FB040000'
163*PO4DD DC X'FB040000'
164*PO4DD DC X'FB040000'
165*PO4DD DC X'FB040000'
166*PO4DD DC X'FB040000'
167*PO4DD DC X'FB040000'
168*PO4DD DC X'FB040000'
169*PO4DD DC X'FB040000'
170*PO4DD DC X'FB040000'
171*PO4DD DC X'FB040000'
172*PO4DD DC X'FB040000'
173*PO4DD DC X'FB040000'
174*PO4DD DC X'FB040000'
175*PO4DD DC X'FB040000'
176*PO4DD DC X'FB040000'
177*PO4DD DC X'FB040000'
178*PO4DD DC X'FB040000'
179*PO4DD DC X'FB040000'
180*PO4DD DC X'FB040000'
181*PO4DD DC X'FB040000'
182*PO4DD DC X'FB040000'
183*PO4DD DC X'FB040000'
184*PO4DD DC X'FB040000'
185*PO4DD DC X'FB040000'
186*PO4DD DC X'FB040000'
187*PO4DD DC X'FB040000'
188*PO4DD DC X'FB040000'
189*PO4DD DC X'FB040000'
190*PO4DD DC X'FB040000'
191*PO4DD DC X'FB040000'
192*PO4DD DC X'FB040000'
193*PO4DD DC X'FB040000'
194*PO4DD DC X'FB040000'
195*PO4DD DC X'FB040000'
196*PO4DD DC X'FB040000'
197*PO4DD DC X'FB040000'
198*PO4DD DC X'FB040000'
199*PO4DD DC X'FB040000'
200*PO4DD DC X'FB040000'
201*PO4DD DC X'FB040000'
202*PO4DD DC X'FB040000'
203*PO4DD DC X'FB040000'
204*PO4DD DC X'FB040000'
205*PO4DD DC X'FB040000'
206*PO4DD DC X'FB040000'
207*PO4DD DC X'FB040000'
208*PO4DD DC X'FB040000'
209*PO4DD DC X'FB040000'
210*PO4DD DC X'FB040000'
211*PO4DD DC X'FB040000'
212*PO4DD DC X'FB040000'
213*PO4DD DC X'FB040000'
214*PO4DD DC X'FB040000'
215*PO4DD DC X'FB040000'
216*PO4DD DC X'FB040000'
217*PO4DD DC X'FB040000'
218*PO4DD DC X'FB040000'
219*PO4DD DC X'FB040000'
220*PO4DD DC X'FB040000'
221*PO4DD DC X'FB040000'
222*PO4DD DC X'FB040000'
223*PO4DD DC X'FB040000'
224*PO4DD DC X'FB040000'
225*PO4DD DC X'FB040000'
226*PO4DD DC X'FB040000'
227*PO4DD DC X'FB040000'
228*PO4DD DC X'FB040000'
229*PO4DD DC X'FB040000'
230*PO4DD DC X'FB040000'
231*PO4DD DC X'FB040000'
232*PO4DD DC X'FB040000'
233*PO4DD DC X'FB040000'
234*PO4DD DC X'FB040000'
235*PO4DD DC X'FB040000'
236*PO4DD DC X'FB040000'
237*PO4DD DC X'FB040000'
238*PO4DD DC X'FB040000'
239*PO4DD DC X'FB040000'
240*PO4DD DC X'FB040000'
241*PO4DD DC X'FB040000'
242*PO4DD DC X'FB040000'
243*PO4DD DC X'FB040000'
244*PO4DD DC X'FB040000'
245*PO4DD DC X'FB040000'
246*PO4DD DC X'FB040000'
247*PO4DD DC X'FB040000'
248*PO4DD DC X'FB040000'
249*PO4DD DC X'FB040000'
250*PO4DD DC X'FB040000'
251*PO4DD DC X'FB040000'
252*PO4DD DC X'FB040000'
253*PO4DD DC X'FB040000'
254*PO4DD DC X'FB040000'
255*PO4DD DC X'FB040000'

PAGE 5
## Example of Debug Facilities

<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJECT CODE</th>
<th>ADDR1</th>
<th>ADDR2</th>
<th>STAT</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>000450</td>
<td></td>
<td></td>
<td></td>
<td>FINAL</td>
<td></td>
</tr>
<tr>
<td>000496</td>
<td>27276727</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00049C</td>
<td>38383838</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0004AD</td>
<td>99999999</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0004A8</td>
<td>000001F8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0004AC</td>
<td>00000450</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0004B0</td>
<td>0000007E</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0004B4</td>
<td>00000301</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>184</td>
<td>185+$LASTSAV EQU $P04BCK</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>186</td>
<td>$P04#WD EQU 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>187</td>
<td>DS OF</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>188</td>
<td>XID DC C'XXX'</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>189</td>
<td>YID DC C'YYY'</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>190</td>
<td>ZID DC C'ZZZ'</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>191</td>
<td>LTORG</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>192</td>
<td>=A(SFIRSTSV)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>193</td>
<td>=A($LASTSAV)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>194</td>
<td>=A($TRACE)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>195</td>
<td>=F'1'</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>197</td>
<td>END</td>
<td></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:

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

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

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

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

(*)

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

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

is the same as

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

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.

| ATEXIT       | [block-name] |
**BLEND—Terminate Current Block**

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

<table>
<thead>
<tr>
<th>BLEND</th>
<th>[block-name] [other-ops]</th>
</tr>
</thead>
</table>

**block-name**

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

**other-ops**

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

*BLOCK—Define a Simple Block of Code*

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

\[
\begin{array}{|c|c|}
\hline
\text{bname} & \text{BLOCK} \\
\hline
\end{array}
\]

*bname*  \hspace{1cm} *sym*

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

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

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

*blname*  sym

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

**MISC**

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

*index-list*

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

*char-index-list*

A list of values for which this CASE will be chosen. This form is coded when the immediately surrounding DOCASE is of the character-string format (indicated by the specification (index, length ) on the DOCASE macro).
Each value in the list is interpreted as a character string and may be one of the following:

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

conditional-test

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

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

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

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

A-9
CORP

CORP—Terminate a Procedure

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

<table>
<thead>
<tr>
<th>label</th>
<th>CORP</th>
<th>proc-name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[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>(value)</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>

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

proc-name

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

RESTORE=(first,last)

dec dig, sym

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

RETURN=reg-list

dec dig, sym

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

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

RC=value

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

RC=(reg)

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

If RC= is not coded the defaults are:

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

LINK=NONE

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

LINK=linkreg

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

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

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

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

biname

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

FOREVER

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

WHILE,looping-group

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

UNTIL,looping-group

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

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

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

OR

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

looping-group

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

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

looping-branch

One of the special looping instructions specified as:

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

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

cond-test

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

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

A DO macro with no operands defaults to a "DO FOREVER".
DOCASE—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}</th>
<th>{index-reg}</th>
<th>{'SIMPLE}'</th>
<th>{'IFANY}'</th>
<th>{'ONLY}'</th>
<th>{'SPARSE}'</th>
</tr>
</thead>
</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)

dec dig, sym

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

(index,W)

RX-type

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

(index,H)

RX-type

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

A-14
Indicates the DOCASE index is located in the byte at address index.

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

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

SIMPLE

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

SPARSE

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

IFANY

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

ONLY

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

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

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

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

| DOEND | [block-name] |
ELSE

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.

| ESAC | [block-name] |

block-name

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.

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

block-name

Indicates the ESACOD is intended to match the outstanding DOCASE block named block-name. If coded, checks will be made to assure that it is the current block.
EXIT—Abnormally Exit to the End of a Containing Block

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

| [label] | EXIT | [block-name] |

**label**

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

**block-name**

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

**FI—Terminate a Conditional Block**

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

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

*block-name*  

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

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

**IF—Define Conditional Block**

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

```
[blname] IF

{ ASYNCH
  cond-test }

[EXIT= exit-block]

[ELSE=else-block]
```

**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

ONEND—Define Normal Loop Termination Code

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

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

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

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

\[\text{block-name} \quad \text{sym}\]

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

PROC—Define a Proc

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

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

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

LINKAGE=OS

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

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

ID=None

No in-line identifier is to be generated.

ID=id-string

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

ID=*  

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

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

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

SAVE=None

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

SAVE=(first,last)  

dec dig, sym

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

SAVE=(, DYNAM)

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

A-31
PROC

SAVE=(,),NONE)

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

SAVE=(,,savearea

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

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

SAVE=(,,length)

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.

BASENONE

Indicates that no base register loading is to be performed.

BASE=basereg

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)

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

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

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

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

A-32
For non-OS procs— BASE=NONE
   (If the first suboperand of baselist is omitted
   for non-OS procs, it defaults to 12.)

WORK=NONE

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

WORK=workreg

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

If the WORK-operand is omitted, defaults are:

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

EXIT=severity

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

DEBUG= options-list

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

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

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

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

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

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

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

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

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

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

To turn off any of the options, prefix the name by NO- or the abbreviation by N- (e.g., "NOPROCTRACE" or "NPT"). When the CORP is generated, options will revert to their status before the PROC macro. To avoid the restoring of the options' status at CORP time, include "GLOBAL" (or "GBL") 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>[PROCEND]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>[proc-name]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[RC=NONE]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(RC=value)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(RC=reg)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[RESTORE=(first [, last])]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[RETURN=reglist]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[LINK=NONE]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(LINK=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 (possibly empty as "call BLEND ( ; )"), and the proc calls never have argument lists. In addition, procs always have multi-word names whose first word indicates the macro of which the proc is a part. (For example, the proc "DOCASE_GENERAL_SETUP" is a part of the DOCASE macro.)

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

SIMPL-M:

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

OS MACRO:

    MACRO

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

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

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

    int   - Integers
    bit   - Logical variables
    char  - Character strings

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

    int to bit:  0 \rightarrow false; all else to true

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

    bit to int:  false \rightarrow 0; true \rightarrow 1
bit to char: false → '0'; true → '1'

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

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

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

```
int X(20)
```

and are referred to as

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

The first element of the array has the index 1.

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

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

```
I := 1
```

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

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

Relations may include implied operands. For example:

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

is the same as

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

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

```
if J ≠ 0 and I / J = 4
```

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

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

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

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

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

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

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

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

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

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

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

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

\[ /*\{ . . . \} */ \]
DO Macro Decision Table

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

The complete form of a DO macro is

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

The complete form of the code generated is given by the partial flow chart:
The following decision table shows the connections which must be made for the various formats. Those shown lightly shaded occur without branching (control falls through to the indicated node). Boxes shown cross-hatched do not occur for that operand combination. An example follows the table.
<table>
<thead>
<tr>
<th>FOREVER</th>
<th>✓</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 O O O O O</td>
</tr>
<tr>
<td>cond-test A</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>AND/OR</td>
<td>A O A A O O O O O O 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 O O O O O O O</td>
</tr>
<tr>
<td>cond-test B</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
</tbody>
</table>

### Entry point - 0

<table>
<thead>
<tr>
<th>B</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>'Branch' leg 2</td>
<td>Condition: T F F T T F F F T T T T T T T T T</td>
</tr>
<tr>
<td>To node</td>
<td>12 7 12 12 7 7 12 9 12 9 12 9 12 12 12</td>
</tr>
<tr>
<td>'Fall thru' leg 3</td>
<td>Condition: F T F F F T T T F F F F F F F F F</td>
</tr>
<tr>
<td>To node</td>
<td>4 4 4 4 4 4 4 4 9 7 9 7 9 7 7 7 7 7</td>
</tr>
<tr>
<td>'Branch' leg 5</td>
<td>Condition: F F F F F F F F F F F F F F F F F</td>
</tr>
<tr>
<td>To node</td>
<td>12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12</td>
</tr>
<tr>
<td>'Fall thru' leg 6</td>
<td>Condition: T T T T T T T T T T T T T T T T T</td>
</tr>
<tr>
<td>To node</td>
<td>7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7</td>
</tr>
</tbody>
</table>

### After Code - 8

<table>
<thead>
<tr>
<th>B</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>'Branch' leg 10</td>
<td>To node</td>
</tr>
<tr>
<td>'Fall thru' leg 11</td>
<td>To node</td>
</tr>
</tbody>
</table>

### OPERAND FORMAT

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
<th>21</th>
<th>22</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example of the code generated for the macro:

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

Operand format number 6.

Code generated:

```
B $18W1
S18U1 CLI SPOT,C'X'
BE $18END
S18W1 LTR 3,3
BNP $18END

<BLOCK'S CODE>
BCT 5,$18U1
S18END DS OH
```

B-8
APPENDIX C

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

135. /* GENERAL PURPOSE GLOBALS. */
136. PROC_COUNTER, /* Number for last special proc label "$Pep". The 
137. -Ep is PROC_COUNTER in hex. */
138. LAST_BLOCK_NUMBER, /* Number used in labels of most recently generated 
139. block. */
140. HEX_IN /* Input value to XHEX macro. */
141. int 
142. PROC_COUNTER, /* Number for last special proc label "$Pep. 
143. Ep is PROC_COUNIER in hex. */
144. LAST_BLOCK_NUMBER, /* Number used in labels of most recently generated 
145. block. */
146. HEX_IN /* Input value to XHEX macro. */
147. bit 
148. BLOCK_LABEL_PREFIX, /* Unique character string for each block 
149. for use in generating labels. */
150. EXIT_SEVERITY, /* Mnote severity for EXIT target message. Can be 
151. set by PROC macros. */
152. HEX(16), /* Constants used in converting decimal to hex by XHEX. */
153. HEX_OUT, /* Output value from XHEX macro. */
154. PREV_SAVETRACE_AREA, /* Holds label generated on last local PROC 
155. save area to be used in producing the static chain for SAVETRACE. */
156. PREV_SAVETRACE_PTR /* Holds label generated as forward pointer 
157. within last local PROC save area for static chain for SAVETRACE. */
158. bit 
159. BLOCK_LABEL_PREFIX, /* Unique character string for each block 
160. for use in generating labels. */
161. EXIT_SEVERITY, /* Mnote severity for EXIT target message. Can be 
162. set by PROC macros. */
163. HEX(16), /* Constants used in converting decimal to hex by XHEX. */
164. HEX_OUT, /* Output value from XHEX macro. */
165. PREV_SAVETRACE_AREA, /* Holds label generated on last local PROC 
166. save area to be used in producing the static chain for SAVETRACE. */
167. PREV_SAVETRACE_PTR /* Holds label generated as forward pointer 
168. within last local PROC save area for static chain for SAVETRACE. */
169. /* DEBUG FLAGS. */
170. bit 
171. DEBUG_BLOCKCOUNTERS_REQD, /* Causes code and counters to keep execution 
172. counts on all blocks. */
173. DEBUG_BLOCKNAMES_REQD, /* Causes block names to be generated as 
174. inline character constants to aid in locating within dumps. */
175. DEBUG_REGVALUES_REQD, /* Causes register values at end of proc's 
176. CORP macro start) to be saved in inline save areas for reference. */
177. DEBUG_MACHNAMES_REQD, /* Causes various intermediate values to be 
178. printed during macro processing for debugging the macros. */
179. DEBUG_LIST_BLOCKS_REQD, /* Causes notes to be generated at the start 
180. and end of all blocks listing their name, number, and static 
181. nesting depth. */
182. DEBUG_MACHNAMES_REQD, /* Causes notes to be generated whenever any 
183. macros are entered (including inner macros) which list the macro's 
184. name; for debugging the macros. */
185. DEBUG_PROC_LOCATIONS_REQD, /* Causes code and counters to keep execution 
186. counts on PROC blocks only. */
187. DEBUG_HASHMAP_REQD, /* Causes proc names to be generated as 
188. inline character constants to aid in locating within dumps. */
189. DEBUG_PROC_TRACE_REQD, /* Causes a trace vector to be generated and 
190. code to be generated to keep track of the last 257 PROCs entered. */
191. DEBUG_SAVETRACE_REQD /* Causes all local save areas to be statically 
192. chained together and code to be generated to link the chain to the 
193. OS save area to provide OS formatting within ABEND dumps. */
Global Definitions -- 14 July 1973

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

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

bit END_LABEL_REQD(100), /* Indicates whether Ith block needs an END label generated during POP_OLD_BLOCK. */
EXIT_LABEL_REQD(100) /* Indicates whether Ith block needs an EXIT label generated during POP_OLD_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 */
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. */
INDICES IN 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. */
FAILTHRU_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, FAILTHRU_LABEL_USED must be set true (by CEP) to insure next sequential instruction following conditional expression receives label definition. */
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). */
```

C-3
InMicro -- 21 June 1973

11001. BASED IF (USER_NAME; REL, MASK, EXIT, ELSE)
11003. /* Initiate a block in the structure. Save any information needed
11004. by ELSE or fi. For ASYNCH type, generate branch around block.
11005. For normal IF (EXIT= not specified), generate conditional expression
11006. tests with branch around block (or to ELSE) for false and fall
11007. through for true; if EXIT= specified, then generate branch to
11008. proper block end for true, fall through for false, and delete
11009. IF block from structure. Put USER_NAME on first executable
11010. instruction if one specified. */
11011. bit
11012. VALID_EXIT
11013. /* VALID_EXIT is true if EXIT= was specified and no errors have
11014. been found to cause the EXIT to be ignored. */
11015. char
11016. EXIT_LABEL, /* Label for EXIT= branch, when deferred until
11017. after block count has been incremented. */
11018. LABEL /* Outstanding label, waiting to be generated. */
11020. call TRACE PRINTER ( ; 'IF') /* Prints macro name "IF" in mnote if tracing ca. */
11021. call PUSH.NEW.BLOCK (USER_NAME; 11022. BLOCK.TYPE.VALUE='IF',
11023. OPERAND1.VALUE=ELSE,
11024. END.LABEL.VALUE=true)
11025. /* Define new block; add to stack. Initialize block specifications.
11026. Assume block will require an END label. Note block type and save
11027. name of ELSE block if one specified here. Set up unique
11028. BLOCK_LABEL_PREFIX for use in generating unique labels. */
11029. call IF_CONDITIONAL_GENERATOR /* Generate conditional tests according to IF macro operands and
11030. the current conditional test specs. */
11031. call IF_CONDITIONAL TEST SPECS
11032. /* Set all conditional test specifications in globals required to
define the action to be performed by the conditional test
send
11033. generators. */
11034. if VALID_EXIT /* i.e., if EXIT specified and still valid... */
11035. then
11036. call IF.EXIT_SPECS /* Reset conditional test specs according to EXIT target provided
11037. no conflicting parameters exist (in that case, set VALID.EXIT to
11038. false). */
11039. call IF_CONDITIONAL GENERATOR /* Generate conditional tests according to IF macro operands and
11040. the current conditional test specs. */
11041. call IF_CONDITIONAL TEST SPECS
11042. /* Set all conditional test specifications in globals required to
define the action to be performed by the conditional test
generators. */
11043. if VALID_EXIT /* i.e., if EXIT specified and still valid... */
11044. then
11045. call IF.EXIT_SPECS /* Reset conditional test specs according to EXIT target provided
11046. no conflicting parameters exist (in that case, set VALID.EXIT to
11047. false). */
11048. call IF_CONDITIONAL GENERATOR /* Generate conditional tests according to IF macro operands and
11049. the current conditional test specs. */
11050. call IF_CONDITIONAL TEST SPECS
11051. /* Set all conditional test specifications in globals required to
define the action to be performed by the conditional test
generators. */
11052. if VALID.EXIT /* i.e., if EXIT specified and still valid... */
11053. then
11054. call IF.EXIT_SPECS /* Reset conditional test specs according to EXIT target provided
11055. no conflicting parameters exist (in that case, set VALID.EXIT to
11056. false). */
11057. call IF_CONDITIONAL GENERATOR /* Generate conditional tests according to IF macro operands and
11058. the current conditional test specs. */
11059. call IF_CONDITIONAL TEST SPECS
11060. /* Set all conditional test specifications in globals required to
define the action to be performed by the conditional test
generators. */
11061. if VALID.EXIT /* i.e., if EXIT specified and still valid... */
11062. then
11063. call IF.EXIT_SPECS /* Reset conditional test specs according to EXIT target provided
11064. no conflicting parameters exist (in that case, set VALID.EXIT to
11065. false). */
11066. call IF_CONDITIONAL GENERATOR /* Generate conditional tests according to IF macro operands and
11067. the current conditional test specs. */
11068. call IF_CONDITIONAL TEST SPECS
11069. /* Set all conditional test specifications in globals required to
define the action to be performed by the conditional test
generators. */
11070. if VALID.EXIT /* i.e., if EXIT specified and still valid... */
11071. then
11072. call IF_CONDITIONAL GENERATOR /* Generate conditional tests according to IF macro operands and
11073. the current conditional test specs. */
11074. then
11075. call IF_CONDITIONAL TEST SPECS
11076. /* Set all conditional test specifications in globals required to
define the action to be performed by the conditional test
generators. */
11077. if VALID.EXIT /* i.e., if EXIT specified and still valid... */
11078. then
11079. call IF_CONDITIONAL GENERATOR /* Generate conditional tests according to IF macro operands and
11080. the current conditional test specs. */
11081. call IF_CONDITIONAL TEST SPECS
11082. /* Set all conditional test specifications in globals required to
define the action to be performed by the conditional test
generators. */
11083. if VALID.EXIT /* i.e., if EXIT specified and still valid... */
11084. then
11085. call IF_CONDITIONAL GENERATOR /* Generate conditional tests according to IF macro operands and
11086. the current conditional test specs. */
11087. call IF_CONDITIONAL TEST SPECS
11088. /* Set all conditional test specifications in globals required to
define the action to be performed by the conditional test
generators. */
11089. if VALID.EXIT /* i.e., if EXIT specified and still valid... */
11090. then
11091. call IF_CONDITIONAL GENERATOR /* Generate conditional tests according to IF macro operands and
11092. the current conditional test specs. */
11093. call IF_CONDITIONAL TEST SPECS
11094. /* Set all conditional test specifications in globals required to
define the action to be performed by the conditional test
generators. */
11095. if VALID.EXIT /* i.e., if EXIT specified and still valid... */
11096. then
11097. call IF_CONDITIONAL GENERATOR /* Generate conditional tests according to IF macro operands and
11098. the current conditional test specs. */
11099. call IF_CONDITIONAL TEST SPECS
11100. /* Set all conditional test specifications in globals required to
define the action to be performed by the conditional test
generators. */
11101. if VALID.EXIT /* i.e., if EXIT specified and still valid... */
11102. then
11103. call IF_CONDITIONAL GENERATOR /* Generate conditional tests according to IF macro operands and
11104. the current conditional test specs. */
11105. call IF_CONDITIONAL TEST SPECS
11106. /* Set all conditional test specifications in globals required to
define the action to be performed by the conditional test
generators. */
11107. if VALID.EXIT /* i.e., if EXIT specified and still valid... */
11108. then
11109. call IF_CONDITIONAL GENERATOR /* Generate conditional tests according to IF macro operands and
11110. the current conditional test specs. */
11111. call IF_CONDITIONAL TEST SPECS
11112. /* Set all conditional test specifications in globals required to
define the action to be performed by the conditional test
generators. */
11113. if VALID.EXIT /* i.e., if EXIT specified and still valid... */
11114. then
11115. call IF_CONDITIONAL GENERATOR /* Generate conditional tests according to IF macro operands and
11116. the current conditional test specs. */
11117. call IF_CONDITIONAL TEST SPECS
11118. /* Set all conditional test specifications in globals required to
define the action to be performed by the conditional test
generators. */
11119. if VALID.EXIT /* i.e., if EXIT specified and still valid... */
11120. then
11121. call IF_CONDITIONAL GENERATOR /* Generate conditional tests according to IF macro operands and
11122. the current conditional test specs. */
11123. call IF_CONDITIONAL TEST SPECS
11124. /* Set all conditional test specifications in globals required to
define the action to be performed by the conditional test
generators. */
11125. if VALID.EXIT /* i.e., if EXIT specified and still valid... */
11126. then
11127. call IF_CONDITIONAL GENERATOR /* Generate conditional tests according to IF macro operands and
11128. the current conditional test specs. */
11129. call IF_CONDITIONAL TEST SPECS
IEW 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. /* Give error message if EXIT specified. Generate branch to
11082. end of IF block. */
11083. if VALID_EXIT
11084. then
11085. generate (LABEL || ' B ' || BLOCK_LABEL_PREFIX || 'END')
11086. /* Branch around asynchronous IF block. */
11087. LABEL := ""
"If" Macro -- 21 June 1973

11090. PROC IF
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. */
11104. /* Set the normal conditional test specs. */
11105. ULTIMATE_BRANCH_LABEL := BLOCK_LABEL_PREFIX || 'END'
11106. /* Branch target for false result is END label—end of IF or
11107. start of ELSE. */
11108. ULTIMATE_FALLTHRU_LABEL := BLOCK_LABEL_PREFIX || 'BEG'
11109. /* Fall-through label to be used is BEG. */
11110. ULTIMATE_FALLTHRU_CONDITION := true
11111. /* Fall through if conditional test yields true result. */
11112. FALLTHRU_LABEL_USED := false /* Assume not required. */
11113. corp
IFDEF EXIT_SPECS
    /* An EXIT= operand has been specified; insure ELSE= not also
       specified. If valid, change conditional test specs to standard
       for EXIT-type IF including the assigning of the branch target
       as the XIT label of the block specified by the EXIT keyword. */
    CHK_HOLD /* Temporary. */
    if ELSE " */ IF ELSE was not omitted... */
    then
      mnote (@, "STRC1103 EXIT= IGNORED WITH ELSE")
      VALID_EXIT := false
    else
      HOLD := ULTIMATE_BRANCH_LABEL
      /* Save old Branch label, we may need it yet. */
      call EXIT_FIND ( ; EXIT)
      /* Sets ULTIMATE_BRANCH_LABEL to XIT label of block whose name
         is specified in the argument; if none specified ("EXIT=*"),
         use block surrounding IF macro; if no such block, issue message,
         leave ULTIMATE_BRANCH_LABEL unmodified, and set ERROR_OCCURRED to
         true. Mark target block as requiring XIT label. */
      if DEBUG_BLOCKCOUNTS_REQD
        then
          EXIT_LABEL := ULTIMATE_BRANCH_LABEL
          ULTIMATE_BRANCH_LABEL := HOLD
          /* Make EXIT-type IF act like regular IF (i.e., fall through on true)
             so we can count the number of times the exit is taken; save the
             EXIT_LABEL for a branch after the count is made and make the
             ULTIMATE_BRANCH_LABEL whatever it would have been had this been
             a regular IF. */
        else
          ULTIMATE_FALLTHRU_LABEL := BLOCK_LABEL_PREFIX
          ULTIMATE_FALLTHRU_CONDITION := false /* Fall through on false. */
          END_LABEL_REQD(CURRENT_NEST_LEVEL) := false
          if ERROR_OCCURRED /* on EXIT_FIND... */
            then
              /* Exit point not found and message has been issued. Make branch
                 point same as fall-through point and clear error (i.e., fix up
                 and continue). */
              ULTIMATE_BRANCH_LABEL := BLOCK_LABEL_PREFIX
              FALLTHRU_LABEL_USED := true /* ...since it's also the branch label. */
              ERROR_OCCURRED := false
            fi
          fi
        fi
      fi
    fi
    endif
ENDIF
11163. PROC IF_CONDITIONAL_GENERATOR
11164. /* Generate code to pass control to the ULTIMATE_FALLTHRU_LABEL (or
11165. to fall through to it) if the conditional test specified has the
11166. logical value which is stored in ULTIMATE_FALLTHRU_CONDITION;
11167. else to pass control to the ULTIMATE_BRANCH_LABEL. Also generate
11168. fall-through label definition if FALLTHRU_LABEL_USED was ever
11169. turned on. */
11170. /* Set up further specifications required by CONDITIONAL_EXPRESSION_  
11171. PROCESSOR. */
11172. FIRST_INDEX := 1
11173. LAST_INDEX := 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 := '1
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
to collectively as SYSLIST). Only the SYSLIST can be passed
directly as arguments; the following variables are effectively
arguments but are passed in global variables:
FIRST_INDEX,
LAST_INDEX,
ULTIMATE_BRANCH_LABEL,
ULTIMATE_FALLTHRU_LABEL,
ULTIMATE_FALLTHRU_CONDITION,
UNIQUE_LABEL_ID,
FALLTHRU_LABEL_USED.
11181. Process operands of the SYSLIST beginning with SYSLIST(FIRST_INDEX)
11182. through SYSLIST(LAST_INDEX) [for the IF macro, this is the entire
11183. SYSLIST], generating the indicated tests to pass control as
indicated above. The UNIQUE_LABEL_ID is used to ensure unique
labels. If a branch is made to the ULTIMATE_FALLTHRU_LABEL, then
11184. FALLTHRU_LABEL_USED is set, else it is unaltered. */
11185. if FALLTHRU_LABEL_USED
11186. then
11187. LABEL := ULTIMATE_FALLTHRU_LABEL
11188. else
11189. LABEL := ''
11190. fi
11191. corp
11192. */

C-8
11206. PROC IP_BLOCK_COUNT
11207. /* If debugging in progress, generate block name and/or count of
11208. block execution. Note that ASYNCH blocks cannot be counted. */
11209. if DEBUG BLOCKNAMES_REQD
11210. then
11211. if SYSLIST(1,1) = 'ASYNCH'
11212. then
11213. generate ('DC ' || BLOCK_NAME(CURRENT_NEST_LEVEL) ||
11214. 'OH0')
11215. /* Asynch branch has already occurred; only name required. */
11216. else
11217. if DEBUG BLOCKCOUNTS_REQD or DEBUG BLOCKNAMES_REQD
11218. then
11219. if DEBUG BLOCKCOUNTS_REQD
11220. then
11221. generate ('LABEL || ' || BLOCKLABELPREFIX || 'IFC')
11222. LABEL := 'DC ' || BLOCK_NAME(CURRENT_NEST_LEVEL) || 'IFC'
11223. /* Generate code to increment block execution count. */
11224. generate (LABEL || ' || ' || BLOCKLABELPREFIX || 'IFC')
11225. else
11226. generate ('LABEL || ' || BLOCKLABELPREFIX || 'IFC')
11227. /* Generate branch around block name and/or count. */
11228. if EXIT LABEL = '
11229. then
11230. LABEL := BLOCKLABELPREFIX || 'GO'
11231. /* Branch directly around block name/count. */
11232. else
11233. generate ('LABEL || ' || BLOCKLABELPREFIX || 'IFC')
11234. /* Branch to end of EXIT= block, postponed to here so we could do the
11235. counting. */
11236. fi
11237. if DEBUG BLOCKNAMES_REQD
11238. then
11239. generate ('LABEL || ' || TARGET)
11240. if EXIT LABEL = '||
11241. then
11242. LABEL := TARGET
11243. /* Label for branch-around must be defined. */
11244. else
11245. LABEL := '||
11246. fi
11247. if DEBUG BLOCKCOUNTS_REQD
11248. then
11249. generate ('DC ' || BLOCK_NAME(CURRENT_NEST_LEVEL) || 'IFC')
11250. else
11251. generate ('DC ' || 'OH0')
11252. fi
11253. if DEBUG BLOCKCOUNTS_REQD
11254. then
11255. generate (BLOCKLABELPREFIX || 'IFC')
11256. else
11257. endif
11258. endif
11259. endif
11260. endif

"ELSE" Macro -- 26 June 1973

13001. **ELSE** (USER_NAME; BLEND=)
13002. /* Initiates a block which is to be executed if the currently active
13003. IF block was not executed, and to be skipped if the IF block was
13004. executed. The ELSE block terminates the IF block with a branch to
13005. the end of the ELSE block and initiates the new ELSE
13006. block. */
13007. ```
13008. int ELSE_BLOCK_NO /* Block number which will be assigned to
13009. upcoming ELSE block. */
13010. char IF_EXIT_LABEL /* Exit label for IF block if one was to have been
13011. generated, else null. */
13012. ```
13013. call TRACE_PRINTER ( ; 'ELSE') /* Print macro name "ELSE" in mnote if tracing on. */
13014. if CURRENT_NEST_LEVEL > NESTING_LIMIT then
13015. mexit
13016. call VERIFY_END( ; 'IF', BLEND)
13017. /* Verifies current block has the name specified
13018. by the BLEND= operand on the ELSE macro (if any) and that it is an IF block.
13019. Various errors receive messages and either intermediate blocks are
13020. BLENDed as a fixup or ERROR_OCCURRED is set. */
13021. if ERROR_OCCURRED then
13022. mexit
13023. else call POP OLD BLOCK ( ) /* Remove IF block from stack. */
13024. call PUSH NEW BLOCK(USER_NAME;
13025. BLOCKTYPE_VALUE='IF', /* Must be marked as "IF" block for FI's
13026. call to VERIFY_END. */
13027. END_LABEL_VALUE=true,
13028. OPERAND2_VALUE='ELSE',
13029. OPERAND3_VALUE=IF_EXIT_LABEL)
13030. /* Generate new block in structure. It is marked as an IF block
13031. to simplify FI checking; "ELSE" is stored in OPERAND2, however,
13032. to indicate that an ELSE has been generated for this IF. An END
13033. label will be required for the block and, if the IF block was
13034. marked as needing an XT label, save it in OPERAND3. */
13035. if USER_NAME != ''
13036. then
13037. generate (USER_NAME || ' DS OB') /* Define outstanding label. */
13038. fi
13039. read

```
"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. fi
15011. fi
15012. CALL VERIFY_END ( ; 'IF', USER_NAME)
15013. /* Verifies current block has the name specified by the USER_NAME
15014. operand on the FI macro (if any) and that it is an IF block.
15015. Various errors receive messages and either intermediate blocks are
15016. blended as a fixup or ERROR_OCCURRED is set.
15017. (Lemma: If CURRENT_NEST_LEVEL > 0 and
15018. [USER_NAME = '' or = BLOCK_NAME(CURRENT_NEST_LEVEL)] and
15019. BLOCK_TYPE(CURRENT_NEST_LEVEL) = 'IF', then
15020. ERROR_OCCURRED will be set false and CURRENT_NEST_LEVEL will be
15021. unmodified. */
15022. if ERROR_OCCURRED
15023. then
15024. mexit
15025. fi
15026. if OPERAND1(CURRENT_NEST_LEVEL) = ''
15027. then
15028. mnote (8, 'STRC1501 ELSE BLOCK
15029. "" NOT FOUND')
15030. fi
15031. CALL POP_OLD_BLOCK ( ; OPERAND3(CURRENT_NEST_LEVEL))
15032. /* Delete current block, generating END and EXIT 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. endif
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

12001. 

12002. /* Create a block in the structure. Save any information needed by

12003. conditional expressions. Insure UNTIL tests are not checked on first

12004. execution. Generate USER_NAME, if any, on first executable

12005. instruction. Order of generated code follows. (Each section may be

12006. omitted in certain cases; see decision table in documentation for

12007. details.)

12008.

12009. FLOW POINT 0: Branch to entry point for total block.

12010. FLOW POINT 1: Start of UNTIL tests, labeled $nnnUl.

12011. FLOW POINT 2: Branch on success of UNTIL tests.

12012. FLOW POINT 3: Fall-through on failure of UNTIL tests.

12013. FLOW POINT 4: Start of WHILE tests, labeled $nnnW1.

12014. FLOW POINT 5: Branch on failure of WHILE tests.

12015. FLOW POINT 6: Fall-through on success of WHILE tests.

12016. FLOW POINT 7: Start of internal looping code (user code between DO

12017. and termination of DO loop by OB, ATEND, or

12018. TERMINATEDOLOOP. */

12019. int

12020. WHILE_INDEX, /* Index within SYSLIST of start of WHILE operands;

12021. Initially points to WHILE keyword but eventually points to start of

12022. conditional test. */

12023. WHILE_END_INDEX, /* Index within SYSLIST of end of WHILE operands. */

12024. UNTIL_INDEX, UNTIL_END_INDEX, /* Same as WHILE counterparts. */

12025. OPERAND_FORMAT, /* Column number of DC decision table (see documenta-

12026. tion) which indicates what operands are present. */

12027. I, /* Temporary work variable. */

12028. LB, /* Index within SYSLIST of the looping branch (BCT, BXH, or

12029. BILE). */

12030. LB LABEL REQ, /* Indicates whether the looping branch requires a label. */

12031. WHILE_COND_TEST, UNTIL_COND_TEST, /* Indicates a conditional test of the given type is present. */

12032. THISCONDITIONAL_REQD, /* Indicates the currently processed type (WHILE or UNTIL) includes a conditional test. */

12033. char

12034. LB_OPCODE_ID, /* Opcode of looping branch: BCT, BXH, or BILE. */

12035. LB_OPERAND1, LB_OPERAND2, /* Operands of the looping branch, if present. */

12036. LB_LOGIC_OP, /* Logical operator ("AND" or "OR") which connects the looping branch

12037. and the conditional test, if both are present. */

12038. LOOPING_BRANCH_TYPE, /* "NONE", "WHILE", or "UNTIL"; indicates position of looping

12039. branch. */

12040. LABEL, /* Any outstanding label waiting to be generated. */

12041. FIRST_ID, /* Label required at start of conditional test being processed. */

12042. MAIN_OP, /* Logical operator ("AND" or "OR") which connects WHILE and UNTIL

12043. looping groups, if both are present. */

C-12
21075. call TRACE_PRINTER ( : 'DO')
21076. /* Prints macro name 'DO' in mnote if tracing on. */
21077. call PUSH_NEW_BLOCK (USER_NAME; BLOCK_TYPE_VALUE='EO')
21078. /* Define new block: add to stack. Initialize block specifications. 
21079. Note block type and set up unique BLOCK_LABEL_PREFIX for use in 
21080. generating unique labels. */
21081. if ERROR_OCCURRED
21082. then
21083. exit
21084. fi
21085. if REL == ' ' or MASK == ' '
21086. then
21087. mnote (8, 'STRC2113 REL= OR MASK= NOT IN PARENTHESES--IGNORED')
21088. fi
21089. LABEL := USER_NAME
21090. call DO.Scan OPERANDS
21091. /* Collect scanning information and looping branch (BCT, BXH, and 
21092. BXLE) information from operands. Set OPERAND_FORMAT based on 
21093. these values. */
21094. if OPERAND_FORMAT == 0 and # 10 and
21095. # 12 and # 19
21096. then
21097. END_LABEL_REQD(CURRENT_NEST_LEVEL) := true
21098. fi
21099. if DEBUG_BLOCKCOUNTS_REQD
21100. then
21101. qenerate (LABEL II 'SR 1,1')
21102. LABEL := 11
21103. qenerate ('STH 1,' II BLOCK_LABEL_PREFIX II 'DOL')
21104. fi
21105. if OPERAND_FORMAT == 0
21106. then /* Not infinite loop. */
21107. call DO.BRANCH FOR LOOP ENTRY
21108. /* Generate flow point 0 branch, if required, to proper label to issue 
21109. UNTIL tests are not made before first loop. */
21110. call DO GENERATE ALL CONDITIONAL TESTS
21111. /* Cause WHILE and UNTIL tests to be generated (flow points 1 through 
21112. 6) with proper labels. */
21113. fi
21114. call DO.LABEL BLOCK
21115. /* Store begin label (flow point 7) into LABEL. */
21116. call DO.INFO_SAVE
21117. /* Insert into stack all information required by TERMINATE_DO_LOOP to 
21118. close loop (flow points 8 through 12). */
21119. call DO.TRACE_COUNTERS
21120. /* Generate any debugging counters, etc. */
21121. if LABEL == '
21122. then
21123. generate (LABEL II 'DS 08')
21124. fi
21125. end
"DO" Macro -- 21 June 1973

21127.  DEF DO_SCAN_OPERANDS
21128.  /* Collect WHILE_INDEX, WHILE_END_INDEX, UNTIL_INDEX, UNTIL_END_INDEX,
21129.  (limits of corresponding conditional test's operands within the
21130.  SYSLIST) and note in WHILE_COND_TEST and UNTIL_COND_TEST whether
21131.  the corresponding keywords include a conditional test to be
21132.  generated; set looping branch information (LOOPING_BRANCH_TYPE,
21133.  LB_OPCODE_ID, LB_OPERAND1, LB_OPERAND2, LB_LABEL_ADDR, and IB_LOGIC_OP)
21134.  which must be passed to TERMINATE_DO LOOP to close loop; and set
21135.  OPERAND_FORMAT (case number code from decision table). */
21137.  CALL DO_FIND_KEYWORDS_AND_PRESENC
21138.  /* Put operand index of "WHILE" and "UNTIL" keywords into xxxx_INDEX
21139.  (or set to 0 if omitted) and note in xxxxx_PRESENT whether these
21140.  looping groups exist. Set LASTOP to index of last valid operand
21141.  in the SYSLIST. */
21142.  CALL DO_FIND_END_INDEXES_AND_MAIN_OP
21143.  /* For each type xxxxx (WHILE and UNTIL) which is present, put index
21144.  of the last operand of looping group for that type into xxxxx_END
21145.  INDEX; if both present, find logic operator which connects them
21146.  and put it into MAIN_OP, else put in null string. */
21147.  CALL DO_LOOPING_BRANCH_AND_FIRST_OPERAND
21148.  /* Collect looping branch information and step WHILE_INDEX and
21149.  UNTIL_INDEX to first operand of conditional test (not including
21150.  looping branch and following operator) or set to zero if not present.
21151.  Also set WHILE_INDEX and UNTIL_INDEX to indicate presence of
21152.  conditional tests. */
21153.  CALL DO_SET_FORMAT
21154.  /* Set type of operands according to decision table, using
21155.  WHILE_PRESENT, UNTIL_PRESENT, MAIN_OP, WHILE_COND_TEST,
21156.  UNTIL_COND_TEST, LOOPING_BRANCH_TYPE, and IB_LOGIC_OP to make
21157.  decision. */
21158.  CORP

***"DO" Macro -- 21 June 1973***

C-14
DO FIND KEYWORDS AND PRESENCE

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

WHILE_INDEX, UNTIL_INDEX := 0

if SYSLIST(1) = 'WHILE' or 'UNTIL' and
  SYSLIST(2) = 'FOREVER' or 'OMITTED'

then

  // SYSLIST(1) = 'WHILE'
  if WHILE_INDEX = 0
    while_INDEX := 1
  else
    start (8, 'STRC2101 OPERANDS AFTER SECOND "WHILE" IGNORED')
    LASTOP := I - 1
  fi

  // SYSLIST(1) = 'UNTIL'
  if UNTIL_INDEX = 0
    UNTIL_INDEX := 1
  else
    start (8, 'STRC2102 OPERANDS AFTER SECOND "UNTIL" IGNORED')
    LASTOP := I - 1
  fi

fi

if WHILE_INDEX = 0 and UNTIL_INDEX = 0

then
  start (8, 'STRC2114 SUPERFLUOUS LOOPING GROUP IGNORED')
fi

/* Decide whether WHILE and UNTIL looping groups are present. The
  possible operand formats are:
  DO UNTIL,looping-group
  DO WHILE,looping-group
  DO WHILE,looping-group,until,looping-group
  DO [no operand or single operand "FOREVER"

UNTIL_PRESENT := (UNTIL_INDEX > 0)
WHILE_PRESENT := (WHILE_INDEX > 0)

/* Last two alternatives are only to fix up when WHILE was
  omitted. */
DO Macro -- 21 June 1973

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

MAIN_OP := ""
WHILE_END_INDEX, UNTIL_END_INDEX := LASTOP /* As initial guess. */

if LASTOP = 1 and SYSLIST(I) = 'FOREVER'
then
  WHILE_PRESENT, UNTIL_PRESENT := false
else
  if WHILE_PRESENT
     then
        if UNTIL_PRESENT
            then
                I := WHILE_INDEX - 1
                if UNTIL_INDEX < I
                then
                    /* DO WHILE <test>,<and/or>,WHILE,<test> */
                    UNTIL_END_INDEX := I - 1 /* Point at end of UNTIL. */
                    if SYSLIST(I) # 'AND' and # 'OR'
                    then
                        /* Error message will be printed later. */
                fi
            else /* WHILE is first: "DO WHILE,<test>,<and/or>,UNTIL,<test>" */
                UNTIL_INDEX := I /* Point at UNTIL. */
                if SYSLIST(I) # 'AND' and # 'OR'
                then
                    /* Error message will be printed later. */
                fi
        else /* WHILE is first: "DO UNTIL,<test>,<and/or>,WHILE,<test>" */
            I := UNTIL_INDEX - 1 /* Point at <and/or>. */
            WHILE_END_INDEX := I
            if SYSLIST(I) # 'AND' and # 'OR'
            then
                /* Error message will be printed later. */
            fi
        else /* WHILE is first: "DO UNTIL,<test>,<and/or>,UNTIL,<test>" */
            UNTIL_INDEX := I
            /* One of the following was entered:
              "DO UNTIL,WHILE,***" or "DO UNTIL,AND,WHILE,***"
              Note (8, "STB2110 WHILE TEST IS VOID-IGNORED")
            WHILE_PRESENT := false
            fi
    else
        if UNTIL_INDEX = UNTIL_END_INDEX
            then
                /* One of the following was entered:
                  "DO UNTIL,WHILE,***" or "DO UNTIL,AND,WHILE,***"
                  Note (8, "STB2111 UNTIL TEST IS VOID-IGNORED")
                UNTIL_PRESENT := false
                fi
        fi
    fi
else /* WHILE_PRESENT but not UNTIL_PRESENT. */
    if WHILE_INDEX = WHILE_END_INDEX /* Which is equal to LASTOP. */
    then /* "DO WHILE" with no other operands. */
        WHILE_PRESENT := false /* Ignore to get infinite DO. */
    fi
else
    if UNTIL_PRESENT and WHILE_PRESENT
        then /* "DO WHILE AND UNTIL" with no other operands. */
            UNTIL_PRESENT := false /* Ignore to get infinite DO. */
    fi
else
    if WHILE_PRESENT and UNTIL_PRESENT
        then /* "DO WHILE AND UNTIL" */
            MAIN_OP := 'AND' /* Assumed. */
            if SYSLIST(I) = 'OR'
            then
                MAIN_OP := 'OR'
            else
                if SYSLIST(I) # 'AND'
                then
                    mnote (8, "STB2112 LOGIC OPERATOR BETWEEN "WHILE" AND "UNTIL""
                        "CANNOT BE ASSUMED")
                fi
            fi
    fi
else
    if WHILE_PRESENT and UNTIL_PRESENT
        then
            MAIN_OP := 'AND' /* Assumed. */
            if SYSLIST(I) = 'OR'
            then
                MAIN_OP := 'OR'
            else
                if SYSLIST(I) # 'AND'
                then
                    mnote (8, "STB2110 LOGIC OPERATOR BETWEEN "WHILE" AND "UNTIL""
                        "CANNOT BE ASSUMED")
                fi
            fi
    fi
else
    fi
```c
21302. PROC DO_LOOPING_BRANCH_AND_FIRST_OPERAND
21303. /* Step WHILE_INDEX and UNTIL_INDEX to first operand of conditional test or set to zero if not present. Collect all looping branch information. Set WHILE_ and UNTIL_COND_TEST to true if appropriate conditional test is present (as opposed to only a looping branch). */
21304. label : 0
21305. LOOPING_BRANCH_TYPE := 'NCNE'
21306. if WHILE_PRESENT
21307. then
21308. I, UNTIL_INDEX := UNTIL_INDEX + 1
21309. /* Move UNTIL_INDEX from pointing at "UNTIL" to pointing at first UNTIL operand. */
21310. if SYSLIST(I,1) = 'BCT' or = 'BXLE' or = 'BXH'
21311. then LOOPING_BRANCH_TYPE := 'UNTIL'
21312. UNTIL_COND_TEST := (UNTIL_END_INDEX > I)
21313. LB := I
21314. /* UNTIL_INDEX is still pointing at the looping branch; we aren't sure how far to advance it yet. */
21315. else UNTIL_COND_TEST := false
21316. fi
21317. fi
21318. if UNTIL_INDEX = 0
21319. UNTIL_INDEX := false
21320. /* Turn off all UNTIL stuff. */
21321. fi
21322. if WHILE_PRESENT
21323. then
21324. I, WHILE_INDEX := WHILE_INDEX + 1
21325. /* Move WHILE_INDEX from pointing at "WHILE" to point at first WHILE operand. */
21326. if SYSLIST(I,1) = 'BCT' or = 'BXLE' or = 'BXH'
21327. then LOOPING_BRANCH_TYPE := 'WHILE'
21328. WHILE_COND_TEST := (WHILE_END_INDEX > I)
21329. LB := I
21330. /* WHILE_INDEX is still pointing at looping branch; we aren't sure how far to advance it yet. */
21331. if SYSLIST(I,1) = 'BCT'
21332. then /* There is no UNTIL looping branch. */
21333. LOOPING_BRANCH_TYPE := 'WHILE'
21334. WHILE_COND_TEST := (WHILE_END_INDEX > I)
21335. LB := I
21336. /* WHILE_INDEX is still pointing at looping branch; we aren't sure how far to advance it yet. */
21337. if SYSLIST(I,1) = 'BCT'
21338. then /* There is no UNTIL looping branch. */
21339. LOOPING_BRANCH_TYPE := 'WHILE'
21340. WHILE_COND_TEST := (WHILE_END_INDEX > I)
21341. LB := I
21342. /* WHILE_INDEX is still pointing at looping branch; we aren't sure how far to advance it yet. */
21343. if SYSLIST(I,1) = 'BCT'
21344. then /* There is no UNTIL looping branch. */
21345. LOOPING_BRANCH_TYPE := 'WHILE'
21346. WHILE_COND_TEST := (WHILE_END_INDEX > I)
21347. LB := I
21348. /* WHILE_INDEX is still pointing at looping branch; we aren't sure how far to advance it yet. */
21349. if WHILE_INDEX = WHILE_INDEX + 1
21350. /* Move WHILE_INDEX from pointing at "WHILE" to point at first WHILE operand. */
21351. if SYSLIST(I,1) = 'BCT' or = 'BXLE' or = 'BXH'
21352. then LOOPING_BRANCH_TYPE := 'WHILE'
21353. WHILE_COND_TEST := (WHILE_END_INDEX > I)
21354. LB := I
21355. /* WHILE_INDEX is still pointing at looping branch; we aren't sure how far to advance it yet. */
21356. if SYSLIST(I,1) = 'BCT'
21357. then/* There is also an UNTIL looping branch. */
21358. LOOPING_BRANCH_TYPE := 'WHILE'
21359. WHILE_COND_TEST := false
21360. MAIN_OP := 1
21361. ELSE
21362. WHILE_INDEX := WHILE_INDEX + 1
21363. WHILE_END_INDEX := 0
21364. WHILE_COND_TEST := false
21365. IF DO_LOOPING_BRANCH_PROCESS
21366. /* Collect looping branch information and advance WHILE or UNTIL_INDEX over looping branch operands. */
```

**DO** Macro -- 21 June 1973

21374. PLOG DO_LOOPING_BRANCH_PROCESS
21375. /* Collect looping branch information:
21376. _LB_OPCODE_ID_ "BCT", "BXLE" or "BH"
21377. _LB_OPERAND1_ First loop or branch operand
21378. _LB_OPERAND2_ Second operand, null (or garbage) for BCT
21379. _LB_LOGIC_OP_ Logic operand connecting looping branch to
21380. rest of WHILE or UNTIL.
21381. _LB_LABEL_REQ_ Indicates whether looping branch will need
21382. a label.
21383. Also step WHILE_ or UNTIL_INDEX over looping branch: if any looping
21384. branch is present, LB contains its index; else, LB = 0. */
21385. int OP_COUNT /* Number of operands looping branch needs.
21386. */
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 SYSLIST(LB) = 1
21399. then /* Not a sublist */
21400. else /* Given as a sublist. */
21401. _LB_OPERAND1_ := SYSLIST(LB,2)
21402. _LB_OPERAND2_ := SYSLIST(LB,3)
21403. if SYSLIST(LB) ≠ OP_COUNT
21404. then
21405. /* SYSLIST(LB) ≠ OP_COUNT */
21406. fi
21407. fi
21408. LB := LB + 1
21409. /* LB should now point to logical operator which connects looping
21410. branch to conditional test, if both are present. */
21411. if (LOOPING_BRANCH_TYPE = "WHILE" and WHILE_COND_TEST)
21412. or (LOOPING_BRANCH_TYPE = "UNTIL" and UNTIL_COND_TEST)
21413. then
21414. _LB_LOGIC_OP_ := SYSLIST(LB)
21415. if _LB_LOGIC_OP_ = "AND" or = "OR"
21416. then
21417. LB := LB + 1 /* Step LB past logic operator. */
21418. else
21419. note (9, "STRC2108 INVALID NUMBER OF OPERANDS FOR '||' _LB_OPCODE_ID_)
21420. fi
21421. fi
21422. fi
21423. if LOOPING_BRANCH_TYPE = "WHILE"
21424. then
21425. WHILE_INDEX := LB
21426. else
21427. fi
21428. /* Set xxxx_INDEX to point at start of conditional test, if any. */
21429. fi
21430. fi
21431. fi
21432. _LB_LABEL_REQ_ := (LOOPING_BRANCH_TYPE = "WHILE")
21433. COPP
DO Macro -- 21 June 1973

21436. proc do_set_format
21437. /* Set OPERAND_FORMAT according to decision table (see documenta-
21438. tion). */
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
"DG" Macro — 21 June 1973

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

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

C-20
21533. /* if WHILE_PRESENT then ... */
21535. else
21536.endif
21537.endif
21538.endif
21539.endif
21540.endif
21541.endif
21542.endif
21543.endif
21544.endif
21545.endif
21546.endif
21547.endif
21548.endif
21549.endif
21550.endif
21551.endif
21552.endif
21553.endif
21554.endif
21555.endif
21556.endif
21557.endif
21558.endif

C-21
21560. **DO** DO_BRANCH_FOE_LOOP_ENTRY
21561. /* Generate branch at flow point 0. if required, to proper label to
21562. ensure UNTIL tests are not made before first loop. */
21564. decode OPERAND_FORMAT binary
21566. case (2, 3, 6-9)
21567. /* Branch around UNTIL conditional test to WHILE conditional test. */
21568. generate (LABEL || 'B' || BLOCK_LABEL_PREFIX || 'MT')
21569. LABEL := ' '
21570. esac
21571. case (10-18)
21572. /* Branch to WHILE looping branch first. */
21573. generate (LABEL || 'B' || BLOCK_LABEL_PREFIX || 'LPH')
21574. LABEL := ' '
21575. esac
21576. case (20, 21, 22)
21577. /* Branch around UNTIL conditional test to DO internal code. */
21578. generate (LABEL || 'B' || BLOCK_LABEL_PREFIX || 'REG')
21579. LABEL := ' '
21580. esac
21581. esac
DO Macro -- 21 June 1973

21564. PROC DO_GENERATE_ALL_CONDITIONAL_TESTS
21565. /* Cause WHILE and UNTIL conditional tests to be generated with proper labels. */
21566.
21567. int PASS /* Looping index. */
21568.
21569. PASS := 1
21570. while PASS <= 2
21571. do
21572. if PASS = 1
21573. then
21574. call DO UNTIL_PREPROCESS
21575. else
21576. call DO WHILE_PREPROCESS
21577. fi
21578. /* The DO_XXXXX_PREPROCESS proc must set:
21579. THISCONDITIONAL_REQD from XXXX_CONDTEST
21580. FIRST_INDEX from XXXX_INDEX
21581. LAST_INDEX from XXXX_END_INDEX
21582. UNIQUE_LABEL_ID with the first letter of XXXX
21583. ULTIMATE_BRANCH_LABEL with the branch target
21584. ULTIMATE_FALLTHRU_LABEL with the fallthru name
21585. ULTIMATE_FALLTHRU_CONDITION with the proper value
21586. FIRST_ID with the first label
21587. to insure proper test generation. */
21588. if THISCONDITIONAL_REQD
21589. then
21590. call DO_GENERATE_CONDITIONAL_SET /* 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 /* For those cases where the ULTIMATE_FALLTHRU_LABEL was not to
21601. follow the conditional test as the next sequential instruction,
21602. generate an unconditional branch to the ULTIMATE_FALLTHRU_LABEL
21603. and clear FALLTHRU_LABEL_USED. */
21604. if FALLTHRU_LABEL_USED
21605. then
21606. LABEL := ULTIMATE_FALLTHRU_LABEL /* Generate label at next opportunity. */
21607. fi
21608. fi
21609. fi
21610. fi
21611. /* Generate label at next opportunity. */
21612. fi
21613. if PASS = 1
21614. then
21615. call DO UNTIL_PCSTPROCESS
21616. /* For those cases where the ULTIMATE_FALLTHRU_LABEL was not to
21617. follow the conditional test as the next sequential instruction,
21618. generate an unconditional branch to the ULTIMATE_FALLTHRU_LABEL
21619. and clear FALLTHRU_LABEL_USED. */
21620. if FALLTHRU_LABEL_USED
21621. then
21622. LABEL := ULTIMATE_FALLTHRU_LABEL /* Generate label at next opportunity. */
21623. fi
21624. fi
21625. fi
21626. if PASS := PASS + 1
21627. od /* (Termination: PASS incremented only (not modifiable by called
21628. procs), must eventually exceed 2.) */
21629. CASE
21630. C-23
"DO" MACRO -- 21 June 1973

C-24
DO Macro

21675. \^[DO\_WHILE\_PREPROCESS\^]
21676.  \^[/* Must set up THIS\_CONDITIONAL\_REQD, FIRST\_INDEX, LAST\_INDEX,\^]
21677.  \^[UNIQUE\_LABEL\_ID, ULTIMATE\_BRANCH\_LABEL, ULTIMATE\_FALLTHRU\_LABEL,\^]
21678.  \^[ULTIMATE\_FALLTHRU\_CONDITION, and FIRST\_ID. */\^]
21680.  THIS\_CONDITIONAL\_REQD := WHILE\_COND\_TEST
21681.  \^[if WHILE\_COND\_TEST\^]
21682.  \^[then\^]
21683.  \^[FIRST\_INDEX := WHILE\_INDEX\^]
21684.  \^[LAST\_INDEX := WHILE\_END\_INDEX\^]
21685.  \^[ULTIMATE\_BRANCH\_LABEL := BLOCK\_LABEL\_PREFIX \| 'END'\^]
21686.  \^[/* Flow point 5 always branches to flow point 12. */\^]
21687.  \^[ULTIMATE\_FALLTHRU\_LABEL := BLOCK\_LABEL\_PREFIX \| 'BEG'\^]
21688.  \^[/* Flow point 6 always falls through to flow point 7. */\^]
21689.  \^[ULTIMATE\_FALLTHRU\_CONDITION := true\^]
21690.  \^[UNIQUE\_LABEL\_ID := 'W'\^]
21691.  \^[FIRST\_ID := BLOCK\_LABEL\_PREFIX \| 'W1'\^]
21692.  \^[fi\^]
21693.  \^[end\^]
DC Macro — 21 June 1973

21695. \texttt{PROC DO\_GENERATE\_CONDITIONAL\_SET}
21696. \texttt{/* Generate code to pass control to the ULTIMATEFLICTHUR\_LABEL (or}
21697. \texttt{to fall through to it) if the conditional test specified by}
21698. \texttt{SYSLIST(FIRST\_INDEX) through SYSLIST(LAST\_INDEX) has the}
21699. \texttt{logical value which is stored in ULTIMATEFLICTHUR\_CONDITION;}
21700. \texttt{else to pass control to the ULTIMATE\_BRANCH\_LABEL. Also}
21701. \texttt{see that FALLTHRU\_LABEL\_USED is set to true if branch to}
21702. \texttt{ULTIMATEFLICTHUR\_LABEL is generated. */}
21703. \texttt{FALLTHUR\_LABEL\_USED := false}
21704. \texttt{if LABEL = # then /* A label is waiting to be generated. */}
21705. \texttt{if FIRST\_ID = ' '}
21706. \texttt{then /* No special label is required at the beginning of this conditional}
21707. \texttt{test. */}
21708. \texttt{FIRST\_ID := LABEL /* Put the label on the conditional test. */}
21709. \texttt{else /* We also have a label waiting for the conditional test. */}
21710. \texttt{generate (LABEL || ' DS OH')}
21711. \texttt{/* Get the LABEL label out of the way. */}
21712. \texttt{fi}
21713. \texttt{LABEL := ' '}
21714. \texttt{fi}
21715. \texttt{call CONDITIONAL\_EXPRESSION\_PROCESSOR (FIRST\_ID; SYSLIST)}
21716. \texttt{/* Generate code corresponding to the operands of the current set}
21717. \texttt{(WHILE or UNTIL) of the DO operands (referred to collectively as}
21718. \texttt{SYSLIST). Only the SYSLIST can be passed directly as arguments;}
21719. \texttt{the following variables are effectively arguments but are passed}
21720. \texttt{in global variables:}
21721. \texttt{FIRST\_INDEX,}
21722. \texttt{ULTIMATE\_BRANCH\_LABEL,}
21723. \texttt{ULTIMATEFLICTHUR\_LABEL,}
21724. \texttt{ULTIMATEFLICTHUR\_LABEL\_USED.}
21725. \texttt{Process operands of the SYSLIST beginning with SYSLIST(FIRST\_INDEX)
21726. through SYSLIST(LAST\_INDEX), generating the indicated tests to pass}
21727. control as indicated above. If a branch is made to the}
21728. \texttt{ULTIMATEFLICTHUR\_LABEL, then FALLTHUR\_LABEL\_USED is set, else}
21729. \texttt{it is unaltered. */}
**DO** Macro -- 21 June 1973

```
21738. **PROC DO_UNTIL_POSTPROCESS**
21739.     /* Generate where required a branch to follow the UNTIL tests to
21740.     transfer control to a non-sequential ULTIMATE_FALLTHRU_LABEL.
21741.     See decision table, flow point 3. Insure FALLTHRU_LABEL_USED
21742.     is turned off so the label will not be generated on the next
21743.     sequential instruction. */
21745.     **ACASE OPERAND_FORMAT ifany
21746.     of
21747.     case (11, 15-18)
21748.     **generate ( 'B ' ** ULTIMATE_FALLTHRU_LABEL)**
21749.     FALLTHRU_LABEL_USED := false
21750.     esac
21751.     esac
21752.     **CORE**
```

C-27
"DO" Macro -- 21 June 1973

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

21757. CASE OPERAND_FORMAT IF
21758. CASE (0, 3, 5, 8-12, 14, 16-22)
21760. IF LABEL # BLOCK_LABELREFIX || 'BEG'
21761. THEN
21762. /* Begin label must be generated. */
21763. IF LABEL # ''
21764. LABEL
21765. LABELS (LABEL || ' DS OK')
21766. ELSE
21767. LABEL := BLOCK_LABELREFIX || 'BEG'
21768. ENDIF
21769. ESAC
21770. ESAC
21771. G00E
PROC DC_INFO_SAVE

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

chap $B0, $B1, $B2
/* One character codes indicating flow point to follow points 8,
10, and 11. */

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

/* Set $8. */

dcase OPERAND_FORMAT only of

case (2,3,11,12,15-18,20)
/* Until conditional test but no UNTIL looping branch. */
$8 := 'U' /* To flow point 1. */

case (9-10,13,14,19,21,22)
/* Until looping branch or no UNTIL but WHILE looping branch. */
$8 := 'U' /* Fall through to point 9. */

case (0) /* Infinite loop. */
$D := 'O' /* To flow point 7. */

esac
endcase

if LOOPING BRANCH_TYPE # 'NOW'
then

/* Set $10. */

dcase OPERAND_FORMAT only of

case (4,7,13,15,16)
$10 := 'W' /* To flow point 4. */

case (5,9-12,14,17-19,22)
$10 := 'U' /* To flow point 7. */

esac
endcase

/* Set $11. */

dcase OPERAND_FORMAT only of

case (4,6,10-13,15,16,19,21)
$11 := 'U' /* Fall through to flow point 12 (end of DO block). */

esac
endcase

"FA"

INFORMATION(CURRENT_WAIT_LEVEL) := $8 || $10 || $11 || LD_LABEL_REQ
false || false || false || false
/* Byte 5 is set true when the loop is terminated (by ATEND, ONEXIT,
or OD). */

Byte 6 is set true when an ATEND occurs for this DC.
Byte 7 is set true when an ONEXIT occurs for this DC.
Byte 8 is set true if a FIN label is required in the OD code. */

OPERAND1(CURRENT_WAIT_LEVEL) := LD_OPERAND1
OPERAND2(CURRENT_WAIT_LEVEL) := LD_OPERAND2
OPERAND3(CURRENT_WAIT_LEVEL) := LD_OPCODE_ID

C-29
"DO" macro -- 21 June 1973

1840. PROC DO_TRACE_COUNTERS
1841. /* If debugging, generate block name and/or counters for block and
1842. loop execution. */
1843. if DEBUG_BLOCKCOUNTS_REQD or DEBUG_BLOCKNAMES_REQD
1844. then
1845. if DEBUG_BLOCKCOUNTS_REQD
1846. then
1847. generate (LABEL || ' LB 1,,' || BLOCK_LABEL_PREFIX || 'DOL')
1848. LABEL := ''
1849. generate (' LA 1,1(1)')
1850. generate (' STH 1,,' || BLOCK_LABEL_PREFIX || 'DOL')
1851. generate (' LA 1,1(1)')
1852. generate (' STH 1,,' || BLOCK_LABEL_PREFIX || 'DTB')
1853. generate (' LA 1,1(1)')
1854. generate (' STH 1,,' || BLOCK_LABEL_PREFIX || 'DTB')
1855. fi /* Generate branch around block name and/or block counts. */
1856. generate (LABEL || ' B ' || BLOCK_LABEL_PREFIX || 'GO')
1857. LABEL := BLOCK_LABEL_PREFIX || 'GO'
1858. if DEBUG_BLOCKNAMES_REQD
1859. then
1860. generate (' DC C' || BLOCK_NAME(CURRENT_NEST_LEVEL) ||
1861. ' 00' ||
1862. ' DC 0')
1863. fi
1864. if DEBUG_BLOCKCOUNTS_REQD
1865. then
1866. generate (BLOCK_LABEL_PREFIX || 'DOL DC H' || CURRENT LOOP COUNT)
1867. generate (BLOCK_LABEL_PREFIX || 'DTB DC H' || OVERALL LOOP COUNT)
1868. fi
1869. fi
1870. corp
**ATEND** Macro  --  31 October 1973

```plaintext
ATEND ( ; USER_NAME)
/
The ATEND macro causes the generation of the loop-terminating code
for the surrounding DO block if such code has not yet been generated.

The target for normal loop termination is then defined to allow the

Code which follows to be executed upon normal loop termination. If

the ATEND has been preceded by an ONEXIT macro, the branch is generate
to the OD for the ONEXIT block. */

ATEND_GENNED, /* Indicates whether ATEND has been generated previously
for this block. */

TDL_GENNED, /* Indicates whether the TERMINATE_DO_LOOP macro has
been evoked for this DO by a previous macro (properly, only
by an ONEXIT). */

FIN_LABEL_REQD /* Indicates a branch to the label "F102FIN" has
been generated and must be defined at OD time. */

call TRACE_PRINTER ( ; 'ATEND')
/
Prints macro name "ATEND" in mnote if tracing on. */

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
note (' , 'STRC2301 MORE THAN ONE "ATEND" IS BLOCK')
mexit
fi

BLOCK_LABEL_PREFIX := 'S' [ [ BLOCKNUM (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. */

done /* TERMINATE_DO_LOOP must have been done by previous ONEXIT. */
genOD ( 'B ' [ [ BLOCK_LABEL_PREFIX 'I ' 'FIN'])

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

if END_LABEL_REQD (CURRENT_NEST_LEVEL)
then
GENEND (BLOCK_LABEL_PREFIX 'I ' 'END DO OD')

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

fi

INFORMATION (CURRENT_NEST_LEVEL) := INFO[1,4]
/
done /* TDL has now been generated. */

INFO[7,1] :=

FIN_LABEL_REQD /* Forward FIN_LABEL_REQD to OD. */

mend
```

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

25001. Macro ONEXIT ( ; USER_NAME)  
25002. /* The ONEXIT macro causes the generation of the loop-terminating code  
25003. for the surrounding DO block if such code has not yet been generated.  
25004. The target for abnormal loop termination (EXIT macros) is then defined  
25005. to allow the code which follows to be executed upon abnormal loop  
25006. termination. If the ONEXIT has been preceded by an ATEND macro,  
25007. the branch is generated to the OD for the ATEND block. */  
25008. 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 "FIN" has  
25014. been generated and must be defined at OD time. */  
25015.  
25016. char INFO /* Holds copy of INFORMATION(CURRENT_NEST_LEVEL). */  
25017.  
25018. call TRACE_PRINTER ( ; 'ONEXIT')  
25019. /* Prints macro name "ONEXIT" in mnote if tracing on. */  
25020. if CURRENT_NEST_LEVEL > NESTING_LIMIT  
25021. then  
25022. mexit  
25023. fi  
25024. call VERIFY_END ( ; 'DO' USER_NAME)  
25025. /* Verifies current block has the name specified  
25026. by the USER_NAME operand of the ONEXIT macro (if any) and that it is a DO block.  
25027. Various errors receive messages and either intermediate blocks are  
25028. BLENDed as a fixup or ERROR_OCCURRED is set. */  
25029. if ERROR_OCCURRED  
25030. then  
25031. mexit  
25032. fi  
25033. INFO := INFORMATION (CURRENT_NEST_LEVEL)  
25034. ONEXIT_GENNED := INFO[7,1]  
25035. /* See if we've already generated an ONEXIT. */  
25036. if ONEXIT_GENNED  
25037. then  
25038. anote (8, 'SRC2501 MORE THAN ONE "ONEXIT" IN BLOCK')  
25039. mexit  
25040. fi  
25041. if -EXIT_LABEL_REQD(CURRENT_NEST_LEVEL)  
25042. then  
25043. mnote (8, 'SRC2502 NO EXIT FOR THIS "DO"')  
25044. mexit  
25045. fi  
25046.  
25047. FIN_LABEL_REQD := INFO[8,1]  
25048. /* Note whether a FIN label has already been referenced. */  
25049. BLOCK_LABEL_PREFIX := IS'II BLOCK_NUMBER(CURRENT_NEST_LEVEL)  
25050. TDL_GENNED := INFO[5,1]  
25051. /* See if we've already generated the loop-terminating code. */  
25052. if TDL_GENNED  
25053. then  
25054. call TERMINATE_DO_LOOP ( ; )  
25055. if TDL_FALLTHRU_OCCURS  
25056. then /* Looping branch expects to fall through to END label. */  
25057. generate ('B' 'II BLOCK_LABEL_PREFIX 'II END')  
25058. fi  
25059. else /* TERMINATE_DO_LOOP must have been done by previous ATEND. */  
25060. generate ('B' 'II BLOCK_LABEL_PREFIX 'II FIN')  
25061. FIN_LABEL_REQD := true  
25062. /* Provide branch to FIN for ATEND block. */  
25063. fi  
25064. generate (BLOCK_LABEL_PREFIX 'II 'EXIT DS OD')  
25065. EXIT_LABEL_REQD(CURRENT_NEST_LEVEL) := false  
25066. /* Provide target for EXIT branch and note that it is no longer  
25067. needed. */  
25068. INFORMATION (CURRENT_NEST_LEVEL) := INFO[1,4]  
25069. TDL := INFO[6,1]  
25070. FIN_LABEL_BEQD := true  
25071. FIN_LABEL_REQD := true  
25072. /* Forward FIN_LABEL_REQD to OD. */  
25073. end
Macro -- 26 June 1973

27001.  MACHE OD ( ; USER_NAME)
27002.  /* Terminate DO loop if ATEND or ONEXIT have not done so and do
27003.  standard block closing. */
27004.  BAI
27005.  TDL_GENERED, /* Indicates whether the looping code has been generated
27006.  yet. */
27007.  FIN_LABEL_REQD /* Indicates whether a "$nnnFIN" label is
27008.  required. (It is used at the end of the ATEND or ONEXIT code.) */
27009.  SHAI
27010.  INFO /* Holds a copy of INFORMATION(CURRENT_NESTLEVEL). */
27011.  call TRACE_PRINTER ( ; 'OD') /* Prints macro name "OD" in mnote if tracing on. */
27012.  if CURRENT_NEST_LEVEL <= NESTING_LIMIT
27013.  then
27014.  call VERIFY_END ( ; 'DO', USER_NAME)
27015.  /* Verifies current block has the name specified by the USER_NAME
27016.  operand of the OD macro (if any) and that it is a DO block.
27017.  Various errors receive messages and either intermediate blocks are
27018.  BLENDed as a fixup or ERROR_OCCURRED is set.
27019.  {Lemma: IF CURRENT_NEST_LEVEL > 0 and
27020.  [USER_NAME = '' or = BLOCK_NAME(CURRENT_NEST_LEVEL)] and
27021.  BLOCK_TYPE(CURRENT_NEST_LEVEL) = 'DO', then
27022.  ERROR_OCCURRED will be set false and CURRENT_NEST_LEVEL will
27023.  not be modified. */
27024.  if ERROR_OCCURRED
27025.  then
27026.  exit
27027.  fi
27028.  INFO := INFORMATION(CURRENT_NEST_LEVEL)
27029.  TDL_GENERED := INFO[5,1]
27030.  FIN_LABEL_REQD := INFO[8,1]
27031.  if ~ TDL_GENERED
27032.  then
27033.  call TERMINATE_DO_LOOP ( ; ) /* Call separate macro to generate loop-terminating branches.
27034.  {Lemma: TERMINATE_DO_LOOP does not modify CURRENT_NEST_LEVEL. */
27035.  else /* ATEND or ONEXIT occurred; we may need FIN label. */
27036.  if FIN_LABEL_REQD
27037.  then
27038.  Generate "$ ( & BLOCK_NUMBER(CURRENT_NEST_LEVEL) & 'FIN DS OH')
27039.  fi
27040.  fi
27041.  fi
27042.  fi
27043.  fi
27044.  call POP_OLD_BLOCK ( ; ) /* Delete current block from the stack.
27045.  {Lemma: POP_OLD_BLOCK decrements CURRENT_NEST_LEVEL by exactly
27046.  one. } */
27047.  fi
27048.  fi
27049.  fi
27050.  /* (Lemma: IF CURRENT_NEST_LEVEL > 0 and
27051.  [USER_NAME = '' or = BLOCK_NAME(CURRENT_NEST_LEVEL)] and
27052.  BLOCK_NAME(CURRENT_NEST_LEVEL) = 'DO', at entry to OD, then
27053.  CURRENT_NEST_LEVEL will be decremented by exactly one. */
**DOCASE** Macro -- 26 June 1973

```
MACRO DOCASE (USER_NAME; INDEX, OPTION, RANGE)

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

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

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

/* Ground rules: LABEL is to be generated on the first of any
executable instruction sequence and then cleared to null; any label
which needs to be so generated may replace a null LABEL. BRANCH_TO
CASE must be set by any branch directly or indirectly to the first
CASE (i.e., by 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
   exit
endif

LABEL := USER_NAME

/* Generate macro's label at first opportunity. */

CALL EXTRACT_OPERANDS

/* Validate operands and issue any error-messages; set INDEX_beg, INDEX_type, INDEX_RANGE_ASSURED, INDEX_LENGTH and CASE_FORMAT. */

if CASE_FORMAT $ 'CONDTEST'

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

endif

CALL DEBUGSTUFF

/* Generates 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 GN)

fi

endif
```
/*
* I1nWP ST 01~. !P9TJT~eds
* adAt
* 4
* ox
* zd
* m
* R.
* xami

if option = 'SIMPLE' or = 'SPARSE'

then

case_format := option

else

if option = 'ONLY'

then /* Allow range specification as second operand of macro, also. */

index_range_assured := true

else

if option = ' ' and = 'IFANY'

then

notes (8, 'STBC3102 ' || option ||

' INVALID SECOND OPERAND—IGNORED')

fi

if index = ' ' then

case_format := 'CONTEST'

else

case_format := 'GENERAL'

if index[1] = 'IFANY' or = 'ONLY'

then

notes (4, 'STBC3101 WARNING—' || index[1] ||

'' ASSUMED AS INDEX; USE "DOCASE", || index[1] ||

'' FOR RANGE SPEC')

fi

fi

if index = ' ' then

index_range_assured := false

else

if range = ' ' and = 'IFANY'

then

notes (8, 'STBC3103 '' RANGE ''

'' INVALID THIRD OPERAND—IGNORED')

fi

fi

index_length := '0' /* Assume not CHARCOMP */

if 'INDEX = 1 and index[1] = '('

then /* A one-element sublist was specified; we take it to be a

register. */

index_reg := index[1] /* Index is specified as a register. */

else

index_reg := ''

if 'INDEX > 1'

then

index_type := index[2]

/* Get index type specified; should be "W" (Word), "H" (Halfword),

"B" (Byte), or length of CHARCOMP index. */

if index_type = 'W' and = 'W' and = 'W'

then

index_length := index_type /* Operand two is length specification. */

else

case_format := 'CHARCOMP' /* Change format to CHARCOMP. */

fi

fi

index_type := 'W' /* No type specified; "W" is default. */

fi

fi

**

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

31149. PROC DOCASE_INDEX_TO_REG
31150. /* If case format is GENERAL, SPARSE, or CHARCCAP 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 GPB1.
31153. Given: This proc is not called for CCNDTEST format. */
31154.
31155. INDEX_ADDR := '' /* Assume index will be stored in GPB1. */
31156. "DOCASE INDEX_TYPE ONLY"
31157. OR
31158. CASE ('E') /* Register index. */
31159. IF INDEX_REG "1"
31160. THEN
31161. generate (LABEL || ' LR 1,' || INDEX_REG)
31162. LABEL := ''
31163. FI
31164. ESAC
31165. CASE ('W') /* Word index. */
31166. generate (LABEL || ' L 1,' || INDEX(1))
31167. LABEL := ''
31168. ESAC
31169. CASE ('H') /* Halfword index. */
31170. generate (LABEL || ' LH 1,' || INDEX(1))
31171. LABEL := ''
31172. ESAC
31173. CASE ('B') /* Byte index. */
31174. IF CASE_FORMAT = 'SIMPLE'
31175. THEN
31176. generate (LABEL || ' SR 1,1')
31177. LABEL := ''
31178. ESAC
31179. ELSE
31180. INDEX_ADDR := INDEX(1)
31181. ESAC
31182. FI /* Postpone loading of index into register; we may want to do CLI's. */
31183. ESAC
31184. "ESAC"
31185. "CASE"

C-37
**DOCASE** macro -- 26 June 1973

31187. **PROC** DOCASE\_GENERAL\_SETUP
31188. /* Generate branch to beginning of general handler for general format
31189. DOCASE. In any case, advance GCASE\_NEST\_LEVEL for the GCASE stack
31190. and initialize the GCASE globals. It is assumed that this proc is
31191. called only for GENERAL, SPARSE, and CHARCCMP 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. LABEL := **
31199. BRANCH\_TO\_CASE1 := true /* Albeit indirectly. */
31200. fi
31201. GCASE\_NEST\_LEVEL := GCASE\_NEST\_LEVEL + 1 /* Advance GCASE stack. */
31202. if GCASE\_NEST\_LEVEL <= GCASE\_NEST\_LIMIT
31203. then /* Clear GCASE globals. */
31204. MAX\_VALUE(GCASE\_NEST\_LEVEL) := -1
31205. /* Maximum branch vector value found. */
31206. NEXT\_COMP\_LABEL\_NO(GCASE\_NEST\_LEVEL) := 1
31207. /* Case number for next comparison case label to be generated. */
31208. J := GCASE\_NEST\_LEVEL * 256
31209. I := J - 256
31210. if CASE\_FORMAT = 'GENERAL'
31211. then /* Clear CASE\_OCCURS bits. */
31212. CASE\_OCCURS(I) := false
31213. else /* GCASE stack overflow. */
31214. mnote (12, 'STRC3104 GENERAL/SPARSE/CHARCCMP GCASE NESTING LEVEL ' ||
31215. GCASE\_NEST\_LEVEL || ' EXCEEDS MAXIMUM OF ' ||
31216. GCASE\_NEST\_LIMIT || '---MACROS MUST BE MODIFIED')
31217. fi
31218. fi
31219. corp
"DOCASE" Macro -- 26 June 1973

31226. ifdef DOCASE_DEBUG_STUFF
31227.  /* Generates last-case variable and block-name constant if required. */
31228. char X /* Temporary. */
31229. endif
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. fi
31243. generate (LABEL II 'B' || BLOCK_LABEL_PREFIX II X)
31244. BRANCH_TO_CASE1 := 3
31245. LABEL := '
31246. endif
31247. if DEBUG_BLOCKNAMES_REQD
31248. then
31249. generate (" DC C" || BLOCK_NAME(CURRENT_NESTLEVEL) || "")
31250. endif
31251. if DEBUG_BLOCKCOUNTS_REQD
31252. then
31253. generate (BLOCK_LABEL_PREFIX II "LSC DC X'00' LAST CASE NUMBER")
31254. endif
31255. endif
31256. endif
31257. endif
31258. endif
31259. endif
"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_CASE1 || CASE1 || INDEX_RANGE_ASSURED

/* Information: Byte 1: Indicates whether first CASE requires a label. Byte 2: Indicates whether a MISC CASE has been found. Byte 3: Indicates whether all self-defined operands are divisible by 4. Byte 4: Indicates whether any unexpected operands were found for general case processing (i.e., any operands which were not equal to their own sequential CASE number). Byte 5: Indicates whether index test for out-of-range value may be omitted. */
C-40
"CASE" Macro -- 27 June 1973

33001. macro CASE (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.
33011. int 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 CASE_LABEL_REQD, /* True unless DOCASE is falling through into
33017. first CASE. */
33018. INDEX_RANGE_ASSURED, /* True if we have been assured (by
33019. "DOCASE --- ONLY") that no values other than those specified
33020. by CASE operands will occur. */
33021. EQUAL_TEST_OUTSTANDING,
33022. /* Indicates that a compare for the current operand has been generated
33023. but the "BE" to the beginning of the block (or "BH" around the
33024. block) has not been generated yet. */
33025. RANGE_TEST_OUTSTANDING,
33026. /* Indicates that a compare for the current range operand has been
33027. generated as well as the branch if below the range; the branch
33028. if within the range to the beginning of the block (or "BH" around
33029. the block) has not been generated yet. */
33030. MISC_FOUND, /* Indicates whether MISC has been found yet. */
33031. MULTIPLESOF4, /* Indicates whether all the self-defining operands
33032. of the CASE macros processed so far are multiples of 4. */
33033. UNEXPECTED_OPERANDS_FOUND /* Indicates whether any operands have
33034. been found so far in the CASE macros' operands which either were
33035. symbolic or were self-definers not equal to their own case number. */
33036.
33037. char CASE_FORMAT, /* Type of CASE operands expected: GENERAL,
33038. SPARSE, SIMPLE, CHARCOMP, or CONDIST. */
33039. INDEX_ADDR, /* Symbolic address of byte or CHARCOMP operand. */
33040. LABEL, /* Outstanding label waiting to be generated. */
33041. NEXT_CASE, /* Label to be generated on next SIMPLE or CONDIST
33042. CASE macro. */
33043. INDEX_LENGTH /* Length of CHARCOMP index. */

C-41
"CASE" Macro -- 27 June 1973

33046. call TRACE_PRINTER (: "CASE")
33047. /* Print macro name "CASE" in anote if tracing on. */
33050. call CASE_POSITION_CHECK
33051. /* Verifies mother block is a DOCASE or attempts fixup with up to 2
33052. BLENDs. Indicates whether un-fixup-able ERROR_OCCURRED. */
33053. if ERROR_OCCURRED
33054. then
33055. anote ('S, 'STRC3304 "CASE" TREATED AS "BLOCK" MACRO')
33056. call BLOCK (USER_NAME;)
33057. anote
33058. fi
33059.
33060. call PUSH_NEW_BLOCK (USER_NAME; BLOCK_TYPE_VALUE='CASE')
33061. /* Define new block; add to stack. Initialize block specifications.
33062. Note block type. Set up unique BLOCK_LABEL_PREFIX for use in
33063. generating unique labels. */
33064. if ERROR_OCCURRED /* during PUSH_NEW_BLOCK (viz., stack overflow) */
33065. then
33066. mexit
33067. fi
33068.
33069. if REL # '' or MASK # ''
33070. then
33071. anote ('S, 'STRC3310 REL= OR MASK= NOT IN PARENTHESES—IGNORED')
33072. call CASE_GET_DOCASEINFO
33073. /* Extract CASE_FORMAT, CASE_LABEL_REQD, CASE_COUNTER, MISC_FOUND,
33074. MAMA_BLOCK_PREFIX, INDEX_RANGE_ASSUMED, INDEX_ADDR, and INDEX_LENGTH
33075. from mother DOCASE block. */
33076. if USER_NAME #
33077. then
33078. generate (USER_NAME II 'DS OH')
33079. /* Any USER_NAME on a CASE macro is just a comment since a branch to
33080. it will produce unpredictable results. If one was specified, get it
33081. out of the way now. */
33082. fi
33083.
33084. if SYSLIST(1) = 'MISC'
33085. then
33086. call CASE_MISC_PROCESS
33087. /* Completely process miscellaneous CASE block. */
33088. else
33089. if CASE_FORMAT = 'GENERAL' or = 'SPARSE' or = 'CHARCOMP'
33100. then
33101. if GCASE_NEST_LEVEL = GE GCASE_NEST_LIMIT
33102. then
33103. call CASE_PROCESSCOMPAREOPERANDS
33104. /* Generate code to handle all "symbolic" operands
33105. (i.e., all those which cannot be handled with the branch vector),
33106. or for all operands in the SPARSE or CHARCOMP format. These are
33107. all handled by generating compare-and-branch sequences. */
33108. if CASE_FORMAT = 'GENERAL'
33109. then
33110. call CASE_PROCESSVECTOROPERANDS
33111. /* Generate labels and save information about any operands which
33112. are to be handled via branch vector. */
33113. fi
33114. fi
33115.
33116.
33117.
33118.
33119.
33120.
33121.
33122.
33123.
33124.
33125.
33126.
33127. send
CASE Macro -- 27 June 1973

33129. \#include CASE_POSITION_CHECK
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.
33134. ERROR_OCCURRED := false /* Assumed. */
33135. if BLOCK_TYPE(CURRENT_NEST_LEVEL) # 'DOCASE'
33136. then
33137. mnote (8, 'STRC3301 "CASE" NOT IMMEDIATE DAUGHTER OF "DOCASE"')
33138. ERROR_OCCURRED := true /* Assume no fixup possible. */
33139. if CURRENT_NEST_LEVEL > 1 and
33140. BLOCK_TYPE(CURRENT_NEST_LEVEL-1) = 'DOCASE'
33141. then
33142. mnote (8, 'STRC3302 ASSUMING "BLEND" OMITTED--INSERTED')
33143. call BLEND (
33144. ERROR_OCCURRED := false /* Note patch up. */
33145. else
33146. if CURRENT_NEST_LEVEL > 2 and
33147. BLOCK_TYPE(CURRENT_NEST_LEVEL-2) = 'DOCASE'
33148. then
33149. mnote (8, 'STRC3303 ASSUMING TWO "BLENDS" OMITTED--INSERTED')
33150. call BLEND ( )
33151. call BLEND ( )
33152. ERROR_OCCURRED := false
33153. fi
33154. fi
33155. fi
33156. core
"CASE" Macro -- 27 June 1973

/ 33156. proc case_get_docase_info
33157.     /* Extract DOCASE information being maintained in mother's stack
33158.     position. */
33159.
33160.     char x
33161.     # non /* temporaries. */
33162.
33163.     non := current_nest_level - 1
33164.     case_counter := operand1(non)
33165.     if stelist(1) ≠ 'misc'
33166.     then
33167.     case_counter := case_counter + 1
33168.     endif
33169.
33170.     index_adds := operand2(non)
33171.     case_format := operand3(non)
33172.     index_length := operand4(non)
33173.     x := information(non)
33174.     case_label_reqd := x[1,1]
33175.     misc_name := x[2,1]
33176.     multiplesop := x[3,1]
33177.     unexpected_operands_reqd := x[4,1]
33178.     index_range_assumed := x[5,1]
33179.     para_block_prefix := "$" || block_number(mbr)
33180. code

*previous error messages*
CASE Macro -- 27 June 1973

33182. BEG CASE_PROCESS_COMPARE_OPERANDS
33183. /* Generate compare-and-branch sequences for all "symbolic" operands
33184. (i.e., those which cannot be handled by the branch vector: all non-
33185. self-defining terms, all self-defining operands which are not in
33186. the range 0-255 inclusive, and all "range" operands \(m,R\)
33187. where either \(m\) or \(n\) is either non-self-defining or outside the
33188. range 0-255) or for all operands if LOCASE was flagged as SPARSE or
33189. CHARCOMP. */
33190.
33191. I := 1 /* Start search with first operand. */
33192. COMP_LABEL_NO := NEXT_COMP_LABEL_NO(GCASE_NEST_LEVEL)
33193. /* Note the next compare label number. */
33194. EQUAL_TEST_OUTSTANDING, RANGE_TEST_OUTSTANDING := false
33195. while I ≤ N'SYSLIST
33196. do /* (CASE_FORMAT = 'SPARSE' OR = 'CHARCOMP' OR
33197. /* [N'SYSLIST(I) ≤ 1 AND (T'SYSLIST(I) = 'N' OR
33198. OR (N'SYSLIST(I) > 1 AND (T'SYSLIST(I,1) = 'N' OR
33199. SYSLIST(I,1) < 0 OR > 255) OR
33200. T'SYSLIST(I,2) = 'N' OR
33201. SYSLIST(I,2) < 0 OR > 255))
33202. then /* EQUAL_TEST_OUTSTANDING
33203. equal := true
33204. else /* RANGE_TEST_OUTSTANDING
33205. end
33206. LABEL := BLOCK_LABEL_PREFIX || 'BEG'
33207. generate (' BEG' II LABEL)
33208. /* After leaving this proc, someone will generate the BEG label
33209. at the beginning of the block. */
33210. EQUAL_TEST_OUTSTANDING := false
33211. else /* RANGE_TEST_OUTSTANDING
33212. LABEL := BLOCK_LABEL_PREFIX || 'BEG'
33213. generate (' BNE' II LABEL)
33214. /* Again, by leaving BEG label in LABEL, it will be generated
33215. after leaving this proc. */
33216. RANGE_TEST_OUTSTANDING := false
33217. else /* Must be first time through. */
33218. if CASE_LABEL_REQD
33219. then COMP_LABEL := MAMA_BLOCK_PREFIX || 'C' || COMP_LABEL_NO
33220. /* Generate label name to be attached to first instruction. */
33221. fi
33222. fi
33223. if UNEXPECTED_OPERANDS_FOUND := true
33224. fi
33225. fi
33226. fi
33227. fi
33228. fi
33229. call CASE_GER_COMPARE
33230. /* Generate compare for the single compare operand at SYSLIST(I)--
33231. either general case non-self-definer or any SPARSE or CHARCOMP
33232. operand. GCASE index is at INDEX_ADDR unless that's null, then is
33233. GENL. Length is in INDEX_LENGTH for CHARCOMP type. Any label to be
33234. generated is in COMP_LABEL; once defined, COMP_LABEL_NO must be
33235. increased. Any branch target outstanding at exit is to be
33236. put into COMP_LABEL. Also on exit, EQUAL_TEST_OUTSTANDING or
33237. RANGE_TEST_OUTSTANDING should be set to indicate which type of
33238. operand was processed. */
33239. fi
33240. I := I + 1 /* Advance to next operand of CASE. */
33241. od /* (Termination: I is incremented above and not modified by
33242. called proc; N'SYSLIST is fixed; I must eventually exceed
33243. N'SYSLIST.) */
33244. fi
33245. /* All compare operands have now been processed. */
33246. if EQUAL_TEST_OUTSTANDING
33247. then
33248. generate (' BNE ' II MAMA_BLOCK_PREFIX || 'C' ||
33249. COMP_LABEL_NO)
33250. fi
33251. else /* RANGE_TEST_OUTSTANDING
33252. generate (' BNE ' || COMP_LABEL)
33253. /* Generate branch to next compare case. Label was left in COMP_LABEL
33254. when we branched on lower end of range. */
33255. fi
33256. fi
33257. NEXT_COMP_LABEL_NO(GCASE_NEST_LEVEL) := COMP_LABEL_NO
33258. /* Store case number of next symbolic case to be defined. */
33259.}
"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 operand. DOCASE index is at INDEXADDR unless that's null, then in
33264. GPR1. Length is in INDEX_LENGTH for CHARCOMP type. Any label to be
33265. generated is in COMP_LABEL; any branch target at exit is to be
33266. put into COMP_LABEL. Also on exit, EQUAL_TEST_OUTSTANDING or
33267. RANGE_TEST_OUTSTANDING should be set to indicate which type of
33268. operand was processed. Operands may be of the form $ or $(\text{m}, \text{n})
33269. the latter implying the range from $\text{m}$ to $\text{n}$. $\text{m}$ and $\text{n}$ may be
33270. self-defining terms or symbols EQUated to absolute expressions for
33271. GENERAL or SPARSE format; for CHARCOMP, they may be absolute or
33272. symbolic addresses of character strings or may be literals (with the
33273. leading "=" and, for character literals, "C" possibly omitted). */
33274. CHAR INSERT /* Temporary. */
33275. C-46
"CASE" Macro -- 27 June 1973

33276. if INDEX_ADDR = ''
33277. then /* Index is in GPR1. */
33278. generate (COMP_LABEL || ' LA 0,' || SYSLIST(I,1))
33279. else /* Index is at INDEX_ADDR. */
33280. if CASE_FORMAT = 'CHARCMP'
33281. then
33282. INSERT := ''
33283. if SYSLIST(I,1)[1,1] = ''
33284. then /* Character string. */
33285. INSERT := 'C'
33286. else
33287. generate (COMP_LABEL || ' CLC ' || INDEX_ADDR || '(' || INDEX_LENGTH | ')')
33288. fi
33289. fi
33290. fi
33291. fi
33292. generate (COMP_LABEL || ' CLI ' || INDEX_ADDR || ',')
33293. fi
33294. fi
33295. if COMP_LABEL = ''
33296. then
33297. COMP_LABEL := ''
33298. COMP_LABEL_NO := COMP_LABEL_NO + 1
33299. fi
33300. if #SYSLIST(I) ≤ 1
33301. then /* Operand is not a range. */
33302. EQUAL_TEST_OUTSTANDING := true
33303. else /* A range has been specified: */
33304. RANGETESTOUTSTANDING := true
33305. /* Generate another label. */
33306. generate (BL ' ' ' CLC ' || INDEX_ADDR || '(' || INDEX_LENGTH | ')')
33307. fi
33308. fi
33309. fi
33310. fi
33311. /* Generate branch on out of range to next compare test label (in this CASE macro or some other CASE in this DOCASE). */
33312. if INDEX_ADDR = ''
33313. then
33314. generate (' LA 0,' || SYSLIST(I,2))
33315. generate (' CR 1,0')
33316. fi
33317. fi
33318. if CASE_FORMAT = 'CHARCOMP'
33319. then
33320. /* Go through the same business figuring out the insert for each suboperand as we did for a. */
33321. if SYSLIST(I,2)[1,1] = ''
33322. then
33323. if SYSLIST(I,2)[K'SYSLIST(I,2),1] = ''
33324. then
33325. fi
33326. else
33327. generate (' CLI ' || INDEX_ADDR || ',')
33328. fi
33329. fi
33330. fi
33331. fi
33332. fi
33333. fi
33334. if SYSLIST(I,2)[1,1] = ''
33335. then
33336. if SYSLIST(I,2)[K'SYSLIST(I,2),1] = ''
33337. then
33338. INSET := 'C'
33339. else
33340. generate (' CLI ' || INDEX_ADDR || ',')
33341. fi
33342. fi
33343. fi
33344. fi
33345. fi
33346. fi
33347. fi
33348. fi
33349. fi
33350. fi
33351. fi
33352. fi
33353. fi
33354. fi

C-47
"CASE" Macro -- 27 June 1973

```plaintext
CASE PROCESS VECTOR OPERANDS
/* Generate labels and note that CASE OCCURS for any operands which can be handled via branch vector: viz., any of the form \( m \) or \( (m,n) \) where \( m \) and \( n \) are self-defining terms in the range 0-255 inclusive. This procedure assumes the CASE_FORMAT is general. */

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

33434. EMB CASE_ASSUMED_VECTOR_CASE
33435. /* Generate label for branch vector cases with no operands. Value is the next higher value than the maximum used so far. CASE_OCCURS is noted, and the SELFDEF_COUNT and MAX_CASE_VALUE are updated. If any previous operands have occurred which were not expected, a message is printed. */
33436.
33437. ini GUESS /* Assumed operand. */
33438. GUESS := MAX_CASE_VALUE(GCASE_NEST-LEVEL) + 1 /* Guess at what omitted operand was intended. */
33439. if GUESS ≤ 0 then /* First guess. */
33440. GUESS := 1
33441. fi
33442. MAX_CASE_VALUE(GCASE_NEST_LEVEL) := GUESS
33443. CASE_OCCURS(BASE+GUESS) := true
33444. generate (MAMA_BLOCK_PREFIX II 'G' II GUESS II 'ES OH')
33445. if UNEXPECTED_OPERANDS_FOUND then
33446. mnote (4, 'STIC3306 EARLIER UNEXPECTED OPERAND IMPLIES THIS TO BE CASE ' II GUESS)
33447. fi
33448. fi
33449. fi
33450. C-49
"CASE" Macro -- 27 June 1973

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

Code
33473. PESEQ CASE_BCT_GEN
33474. */ Generate BCT for this simple case. Verify operand, if any. */
33476. GENERATE (LABEL || BCT 1, || NEXT_CASE)
33477. LABEL := ""
33478. if T'SYSTLIST(1) = 'O'
33479. then /* An operand was specified. */
33480. if T'SYSTLIST(1) = 'W'
33481. then /* Operand is a self-defining term. */
33482. if SYSTLIST(1) = CASE_COUNTER
33483. then
33484. /* ABD (9, 'SYSC330F OPERAND INVALID VALUE OF SIMPLE CASE' ||
33485. CASE_COUNTER)
33486. fi
33487. else /* Operand is not self-defining term. */
33488. /* ABD (9, 'SYSC330F OPERAND MUST BE SELF-DEFINING TERM OR OMITTED' ||
33489. CASE_COUNTER)
33490. fi
33491. fi
33492. COLD
"CASE" Macro -- 27 June 1973

33494. PROC CASE_CONDITICNS
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. UNIQUE_LABEL_ID := 't'
33505. CALL CONDITIONAL_EXPRESSION_PROCESSOR (LABEL; SYSLIST)
33506. /* Generate code corresponding to the operands of the CASE macro */
33507. (referred to collectively as SYSLIST). Only the SYSLIST can be passed
33508. directly as arguments; the following variables are effectively
33509. arguments but are passed in global variables:
33510. FIRST_INDEX,
33511. LAST_INDEX,
33512. ULTIMATE_BRANCH_LABEL,
33513. ULTIMATE_FALLTHRU_LABEL,
33514. ULTIMATE_FALLTHRU_CONDITION,
33515. UNIQUE_LABEL_ID,
33516. FALLTHRU_LABEL_USED.
33517. Process operands of the SYSLIST beginning with SYSLIST(FIRST_INDEX)
33518. through SYSLIST(LAST_INDEX) (for the CASE macro, this is the entire
33519. SYSLIST), generating the indicated test to pass control to the
33520. ULTIMATE_FALLTHRU_LABEL if the test succeeds, else to the
33521. ULTIMATE_BRANCH_LABEL. The UNIQUE_LABEL_ID is used to insure
33522. unique labels where needed. If a branch is made to the
33523. ULTIMATE_FALLTHRU_LABEL, then set FALLTHRU_LABEL_USED; else
33524. it is unaltered. */
33525. if FALLTHRU_LABEL_USED
33526. then
33527. LABEL := BLOCK_LABEL_PREFIX || 'BEG'
33528. else
33529. LABEL := '
33530. fi
33531. fi
33532. dup
CASE Macro -- 27 June 1973

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

if MISC_FOUND
  then
    note (8, 'STRC3311 MULTIPLE MISC CASES IN DOCASE - THIS BLOCK " Quản" ')
    INDEX_RANGE_ASSURED := false
  else
    LABEL := MAMABLOCK_PREFIX || 'MSC'
    /* Make MISC label outstanding (generate on next instruction). */
    MISC_FOUND := true
  fi

if INDEX_RANGE_ASSURED
  then
    note (8, 'STRC3308 "DOCASE...ONLY" INVALID WITH MISC')
    INDEX_RANGE_ASSURED := false
  fi

C-53
CASE- Macro -- 27 June 1973

33556. proc CASE_TRACE_COUNTER
33557. /* Generate any debugging counters and/or labels requested. */
33558. if DEBUG_BLOCKCOUNTS_REQD or DEBUG_BLOCKNAMES_REQD
33559. then if DEBUG_BLOCKCOUNTS_REQD
33560. then /* Generate code to advance this case's counter. */
33561. generate (LABEL || 'LH 1,' || BLOCK_LABEL_PREFIX || 'CTR')
33562. generate ('LA 1,(1)')
33563. generate ('STI 1,' || BLOCK_LABEL_PREFIX || 'CTR')
33564. if SISLIST(1) = 'MISC' or CASE_COUNTER > 255
33565. then
33566. generate ('MVI ' || MAMA_BLOCK_PREFIX || 'LSC,X'FF')
33567. else
33568. HEX_IN := CASE_COUNTER
33569. call XHEX ( ; )
33570. generate ("STBC3313 CASE DEBUG ID=X'" || HEX_OUT || ";"")
33571. generate ("MVI ' || MAMA_BLOCK_PREFIX || 'LSC,X'"")
33572. endif
33573. endif
33574. endif
33575. /* Branch around count and/or block name and set up label to be defined eventually. */
33576. if DEBUG_BLOCKNAMES_REQD
33577. generate ("DC C" || BLOCK_NAME(CURRENT_NEST_LEVEL) || ";"")
33578. endif
33579. if DEBUG_BLOCKCOUNTS_REQD
33580. generate ("DC H'0' CASE COUNT")
33581. endif
33582. endif
33583. endif
33584. endif
33585. endif
33586. endif
33587. endif
33588. endif
33589. endif
33590. endif
33591. endif
33592. endif
33593. endif
33594. endif
33595. endif
33596. endif
33597. endif
33598. endif
33599. endif
33600. endif
33601. endif
33602. endif
CASE UPDATEINFO

Returns to mother DOCASE level possibly updated information which was extracted by CASE_GET_DOCASE_INFO. */

int MOM /* Index level of DOCASE block. */

MOM := CURRENT_NEST_LEVEL - 1
INFORMATION(MOM) := CASE | MISC_FOUND | MULTIPLE5OF4 |
UNEXPECTED_OPERANDS_FOUND | INDEX_RANGE_ASSURED

/* First byte indicates case label is required on next case. */
OPERAND1(MOM) := CASE_COUNTER

/* No need to update OPERAND2 (INDEX_ADDR) or OPERAND3 (CASE_FORMAT)
or OPERAND4 (INDEX_LENGTH). None ever change. */
C-55
"ESAC" Macro -- 3 July 1973

35001. \texttt{ESAC} \{ ; \texttt{USER\_NAME}\}
35002. /* Generate end to match CASE block. Do standard block closing, then
35003. generate branch to end of another END block. */
35004. \texttt{call TRACE\_PRINTER} \{ ; \texttt{"ESAC"}\}
35005. /* Print macro name "ESAC" in mnote if tracing on. */
35006. \texttt{if CURRENT\_NEST\_LEVEL} \leq \texttt{NESTING\_LIMIT}
35007. \texttt{then}
35008. \texttt{call VERIFY\_END} \{ ; \texttt{"CASE"}, \texttt{USER\_NAME}\}
35009. /* Verifies current block has the name specified by the USER\_NAME
35100. operand of the ESAC macro (if any) and that it is a CASE block.
35101. Various errors receive messages and either intermediate blocks are
35102. BLENDed as a fixup or ERROR\_OCCURRED is set.
35103. (Lemma: If CURRENT\_NEST\_LEVEL > 0 and
35104. \texttt{[USER\_NAME = "" or = BLOCK\_NAME(CURRENT\_NEST\_LEVEL)] and
35105. BLOCK\_TYPE(CURRENT\_NEST\_LEVEL) = "CASE", then
35106. ERROR\_OCCURRED will be set \texttt{false} and CURRENT\_NEST\_LEVEL will not
35107. be modified.) */
35108. \texttt{if ERROR\_OCCURRED}
35109. \texttt{then}
35110. \texttt{mexit}
35111. \texttt{fi}
35112. \texttt{fi}
35113. \texttt{call POP\_OLD\_BLOCK} \{ \}
35114. /* Delete current block, generating END and XII labels as required, and
35115. popping stack. (Lemma: POP\_OLD\_BLOCK decrements CURRENT\_NEST\_LEVEL
35116. by exactly one.) */
35117. \texttt{if CURRENT\_NEST\_LEVEL} \leq \texttt{NESTING\_LIMIT}
35118. \texttt{then}
35119. \texttt{generate \{" \texttt{B \$' \texttt{\(II \) BLOCK\_NUMBER(CURRENT\_NEST\_LEVEL) \(II\) \texttt{END}\} \}}
35120. \texttt{\texttt{\"}}
35121. /* Generate branch to end of \texttt{END} block. */
35122. \texttt{fi}
35123. \texttt{fi}
35124. \texttt{end}
35125. /* (Lemma: If CURRENT\_NEST\_LEVEL > 0 and
35126. \texttt{[USER\_NAME = "" or = BLOCK\_NAME(CURRENT\_NEST\_LEVEL)] and
35127. BLOCK\_TYPE(CURRENT\_NEST\_LEVEL) = "CASE" at entry to \texttt{ESAC}, then
35128. CURRENT\_NEST\_LEVEL will be decremented by exactly one.) */
ESACOD Macro  --  3 July 1973

"ESACOD" Macro  --  3 July 1973

37001. BASED ESACOD ( ; USER_NAME)
37002. /* Generates final part of DOCASE processing: for SIMPLE, CONDTEST,
37003. or SPARSE type DOCASE, the EQU for the MISC block (or END of DOCASE)
37004. to the last generated branch target is required; for GENERAL type
37005. DOCASE, the branch vector and the transfer to any symbolic
37006. compares or MISC block must be generated. Finally, the block is
37007. popped. */
37008.
37009. int 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.
37015. int MISC_FOUND, /* Indicates whether a MISC CASE was found. */
37016. MULTIPLEOF4, /* Indicates whether all branch-vector operands were
37017. multiples of 4. */
37018. INDEX_RANGE_ASSURED, /* true if we have been assured (by
37019. "DOCASE ***ONLY") that no values other than those specified
37020. by CASE operands will occur. */
37021. ANY_COMP_CASES, /* Indicates whether any "compare" cases were
37022. generated (either CHARCOMP or symbolic general case operands). */
37023. ANY_SELFDEF_CASES, /* Indicates whether any "self-defining" cases (to
37024. be handled by branch vector) were generated. */
37025. RANGE_TEST_REQD /* Indicates that both branch vector and compare
37026. operands were present. */
37027. CASEFORMAT, /* Type of CASEs present: GENERAL, SPARSE,
37028. CHARCOMP, SIMPLE, or CONDTEST. */
37029. INDEX_ADDR, /* Address of DOCASE index. */
37030. NOCASE, /* Label for branch vector processing used for unspecified
37031. cases. */
37032. LABEL /* Any outstanding label waiting to be generated. */
37033. /* (Ground rules: No ESACOD proc modifies CURRENT_NEST_LEVEL.
37034. This can be shown by referring to the cross-reference index.) */
CALL TRACE_PRINTER ( ; 'ESACOD')

/* Print macro name "ESACOD" in mnote if tracing on. */

CALL VERIFY_END ( ; 'DOCASE', USER_NAME)

/* Verifies current block has the name specified by the USER_NAME operand of the ESACOD macro (if any) and that it is a DOCASE block. Various errors receive messages and either intermediate blocks are blended as a fixup or ERROR_OCCURRED is set. [Lemma: If CURRENT_NEST_LEVEL > 0 and [USER_NAME = ' ' or - BLOCK_NAME(CURRENT_NEST_LEVEL)] and BLOCK_TYPE = 'DOCASE', then ERROR_OCCURRED will be set false and CURRENT_NEST_LEVEL will not be modified. */

IF ERROR_OCCURRED THEN
EXIT
FI

CALL ESACOD_INFO_UNPACK
/* Extracts CASE_FORMAT, CASE_COUNTER, INDEX_ADDR, MISC_FOUND, BLOCK_LABEL_PREFIX, INDEX_RANGE_ASSURED, and MULTIPLESOF4 from stack. */

IF CASE_FORMAT = 'GENERAL' THEN
CALL ESACOD_GENERAL_CASE_CBOICE /* Generate all code to complete processing of general case. */
ELSE IF CASE_FORMAT = 'SPARSE' or = 'CHARCCMP' THEN
T := NEXT_COMP_LABEL_NO(GCASE_NEST_LEVEL)
/* We need to define last compare case target. */
GCASE_NEST_LEVEL := GCASE_NEST_LEVEL - 1
/* Pop GCASE stack. */
ELSE /* CONDTST or SIMPLE. */
T := CASE_COUNTER + 1
/* We need to define last conditional test target. */
FI
IF MISC_FOUND THEN
GENERATE (BLOCK_LABEL_PREFIX || 'C' || T || ' EQU ' || BLOCK_LABEL_PREFIX || 'TSC')
ELSE
GENERATE (BLOCK_LABEL_PREFIX || 'C' || T || ' DS OH')
FI
CALL POP_OLD_BLOCK ( ; ) /* (Lemma: POP_OLD_BLOCK decrements CURRENT_NEST_LEVEL by exactly one. ) */

/* (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. ) */
DEFO ESACOD_INFO_UNPACK

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

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

C-59
"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. */
37108.
37109. call ESACOD_GENERAL_CASE_INFO
37110. /* Pops MAX_SD_VALUE, CCMP_LABEL_NO, and BASE (of CASE_OCCURS array) out of GCASE stack. */
37111.
37112. if ~ ERROR_OCCURRED
37113. then
37114. if ANY_SELFDEF_CASES
37115. then
37116. call ESACOD_SELFDEF_GEN
37117. /* Handles branch vector-type implementation for all cases which contain self-defining terms (of value < 256). Also generates linkage for any other terms and/or MISC case which were used with the self-definers. */
37118.
37119. else
37120. /* No self-definers were present. */
37121. if ANY_COMP_CASES
37122. then
37123. call ESACOD_GENERAL_SYM_ONLY
37124. /* Generate linkage to process systolic operands and MISC in the absence of self-definers. */
37125.
37126. else
37127. error (8, 'SYNC:701 GCASE CONTAINS NO VALID CASES')
37128.
37129.
37130. fi
37131. fi
37132. fi
37133. corp
ESACOD macro -- 3 July 1973

37135. \texttt{DIGG ESACOD\_GENERAL\_SYMB\_ONLY}
37136. /* Generate linkage to process symbolic operands and MISC in the
37137. absence of self-definers (self-defined terms of value < 256). */
37138.
37139. \texttt{generate (BLOCK\_LABEL\_PREFIX | | 'SIG EQU' | |}
37140. \texttt{BLOCK\_LABEL\_PREFIX | | 'C7')}
37141. \texttt{if MISC\_FOUND}
37142. \texttt{then}
37143. \texttt{generate (BLOCK\_LABEL\_PREFIX | | 'C' | | COMP\_LABEL\_NO | | ' EQU' | |}
37144. \texttt{BLOCK\_LABEL\_PREFIX | | 'MISC')}
37145. \texttt{else}
37146. \texttt{generate (BLOCK\_LABEL\_PREFIX | | 'C' | | COMP\_LABEL\_NO | | ' DS 08')}
37147. \texttt{fi}
37148. \texttt{fi}
ESACOD Macro -- 3 July 1973

37151. BLOC ESACD_SELFDEF_GEN
37152. /* Handles branch vector generation for processing cases defined by
37153. self-defining terms (of value < 256). Also generates linkage for
37154. symbolic terms and/or MISC case following self definers. */
37156. LABEL := BLOCK_LABEL_PREFIX || 'BEG'
37157. /* Note that BEG label must be generated on first instruction. */
37158. docase ifany
37159. of
37160. case ANY_COMP_CASES
37161. NOCASE := BLOCK LABEL_PREFIX || 'C1'
37162. esac
37163. case MISC_FOUND
37164. NOCASE := BLOCK_LABEL_PREFIX || 'MSC'
37165. esac
37166. case misc
37167. NOCASE := BLOCK_LABEL_PREFIX || 'MSC'
37168. esac
37169. esac
37170. RANGE_TEST_REQD := (INDEX_RANGE_ASSURED or ANY_COMP_CASES)
37171. if RANGE_TEST_REQD
37172. then
37173. call ESACOD_OUTOF_RANGE_CHECK
37174. /* Generate check for index out of the range 0 through
37175. MAX_SD_VALUE. */
37176. fi
37177. call ESACOD_BRVCT_GEN
37178. /* Generate branch vector and all final constants and equates
37179. required. */
37180. cop
ESACOD Macro --- 3 July 1973

37182.  **PROC** ESACOD_GENERAL_CASE_INFO
37183.  /* Pops following information out of GCASE stack. Indicates success
37184.  (or lack thereof) in ERROR_OCCURRED. */
37185.  **INT** \( I \)
37187.  \( I \) := GCASE_NEST_LEVEL
37188.  GCASE_NEST_LEVEL := \( I - 1 \)
37189.  if \( I > GCASE_NEST_LIMIT \) then
37190.  ERROR_OCCURRED := true
37191.  \( I \) := GCASE_NEST_LEVEL
37192.  else
37193.  ERROR_OCCURRED := false
37194.  \( I \) := GCASE_NEST_LEVEL
37195.  \( I \) := NEXT_COMP_LABEL_NO(I)
37196.  \( I \) := (COMP_LABEL_NO > 1) ? MAX_CASE_VALUE(I) : 0
37197.  \( I \) := (BASE := ((I-1) * 256) + 1)
37198.  \( I \) := NEXT_COMP_LABEL_NO(I)
37199.  \( I \) := (COMP_LABEL_NO > 1) ? MAX_CASE_VALUE(I) : 0
37200.  **ENDPROC**
ESACOD Macro -- 3 July 1973

37202. ELOG ESACOD_OUT_OF_RANGE_CHECK

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

37207. if INDEX_ADDR = ""
37208. then /* Index is in GPR1 */
37209. generate (LABEL || ' LTS 1,1')
37210. if CASE_OCCURS(BASE)
37211. then /* CASE 0 occurs */
37212. generate (" BR " || NOCASE)
37213. else
37214. generate (" BNP " || NOCASE)
37215. fi
37216. generate (' C 1,1 || BLOCK_LABEL_PREFIX || 'SIZ')
37217. if MULTIPLESOF4
37218. then
37219. generate (" LA 0,3")
37220. generate (" BW 0,1")
37221. generate (" BNZ " || NOCASE)
37222. fi
37223. fi
37224. else
37225. generate (LABEL || ' CLI ' || INDEX_ADDR || ',' || MAX_SD_VALUE)
37226. if MULTIPLESOF4
37227. then
37228. generate (" TM " || INDEX_ADDR || ",B'00000011'"")
37229. generate (" BNZ " || NOCASE)
37230. fi
37231. fi
37232. fi
37233. LABEL := ""
37234. CEE
**ESACOD** Macro — 3 July 1973

```assembly
27236. PSEG ESACOD_BVCT_GEN
27237. /* Generate branch vector proper. */
27238. int I, INCX
27240. if INDELI_ADDR # "#
27241. then /* Generate code to put byte index into GPR1 */
27242. generate (LABEL || 'SR 1,1')
27243. LABEL := ".
27244. generate (' IC 1,1 "INDELI_ADDR")
27245. fi
27246. if MULTIPLOPS4
27247. then
27248. INCX := 4
27249. else
27250. INCX := 1
27251. generate (LABEL || ' S LA 1,2')
27252. LABEL := ";"
27253. fi
27254. if CASE_OCCURS(BASE) OR INDELI_ADDR = "#
27255. then /* Zero case must be included in branch vector. */
27256. generate (LABEL || ' B *{1}"
27257. I := 0
27258. else
27259. generate (LABEL || ' B *{1}"
27260. I := INCX
27261. /* Skip the zero case and start with case 1 (or case 4). */
27262. fi
27263. LABEL := ";"
27264. while I <= MAX_SD_VALUE
27265. do
27266. if CASE_OCCURS(BASE+I)
27267. then
27268. generate (' B ' || BLOCK_LABEL_PREFIX || 'G' || I"
27269. else
27270. generate (' B ' || NOCASE"
27271. fi
27272. I := I + INCX
27273. od /* Termination: INCX > 0, so I is incremented in loop;
27274. MAX_SD_VALUE is fixed, therefore I must eventually exceed
27275. BASE_SD_VALUE. */
27276. if RANGE_TEST_ADDR AND INDEX_ADDR = "#
27277. then
27278. generate (BLOCK_LABEL_PREFIX || "SIZ DC F" || MAX SD VALUE ||
27279. ""
27280. fi
27281. if ANY_COMP_CASES
27282. then
27283. if MISC_CASES
27284. then
27285. generate (BLOCK_LABEL_PREFIX || 'C' || COMP_LABEL_NO || 'EQU " ||
27286. BLOCK_LABEL_PREFIX || 'RSC')
27287. else
27288. generate (BLOCK_LABEL_PREFIX || 'C' || COMP_LABEL_NO || ' DS DH")
27289. fi
27290. fi
27291. end
```

C-65
ELOCK Macro

15 June 1973

41001. macro BLOCK (USER_NAME;)
41002. /* Generate simple one-in-one-out block in structure with name
41003. specified. */
41004. char LABEL /* Contains any outstanding label waiting to be generated. */
41007. call TRACE_PRINTER ( ; 'BLOCK') /* Prints macro name "BLOCK" in mnote if tracing on. */
41009. call PUSH_NEW_BLOCK(USER_NAME; BLOCK_TYPE_VALUE='BLOCK') /* Define new block; add to stack. Initialize block specifications. */
4111. Note block type and set up a unique BLOCK_LABEL_PREFIX for use in
4112. generating labels. */
4113. if ERROR_OCCURRED
4114. then
4115. mexit
4116. fi
4117. LABEL := USER_NAME
4118. call BLOCK_TRACE_COUNTERS /* If block counts were requested, generate counters and incrementing
4119. instructions. Any label waiting to be defined is returned in
4121. LABEL. */
4122. if LABEL # ''
4123. then
4124. generate (LABEL || ' DS OH') /* Define label if one required and not yet defined. */
4126. fi
4127. mexit
41029. PROC BLOCK_TRACE_COUNTERS
41030.  /* Generate debugging information required—block name constant
41031. and/or block counters. */
41032.
41033. IF DEBUG_BLOCKCOUNTERS_REQD OR DEBUG_BLOCKNAMES_REQD
41034. THEN
41035. IF DEBUG_BLOCKCOUNTERS_REQD
41036. THEN
41037. /* Generate block count incrementing instructions. */
41038. generate (LABEL || 'LB 1,' || BLOCK_LABEL_PREFIX || 'BLC')
41039. LABEL := 10 /* Clear LABEL to show it has been generated. */
41040. generate ('LA 1,1(1)')
41041. generate ('STH 1,' || BLOCK_LABEL_PREFIX || 'BLC')
41042. fi
41043. generate (LABEL || 'B' || BLOCK_LABEL_PREFIX || 'GO')
41044. /* Establish GO label as requiring definition. */
41045. LABEL := BLOCK_LABEL_PREFIX || 'GO'
41046. /* Generate branch around block name and/or count. */
41047. IF DEBUG_BLOCKNAMES_REQD
41048. THEN
41049. generate ('DC C'|| BLOCK_NAME(CURRENT_NEST_LEVEL) || '0')
41050. fi
41051. IF DEBUG_BLOCKCOUNTERS_REQD
41052. THEN
41053. generate (BLOCK_LABEL_PREFIX || 'BLC DC W0' 'BLOCK COUNT')
41054. fi
41055. fi
41056. fi
41057. SEE
**BLEND** Macro -- 15 June 1973

43001. **MACRO BLEND ( ; USER_NAME, RETURN=, LINK=, RESTORE=, RC=)**
43002. /* The BLEND macro acts as a generic name for IF, OD, ESAC, ESACOD, and
43003. of the former, the proper macro is invoked depending on the block type being terminated. For BLOCK blocks, the block is
43004. block type being terminated. For BLOCK blocks, the block is
43005. simply terminated. */
43007. **LINE I */ Temporary. */
43009. **CALL TRACE_PRINTER ( ; 'BLEND')**
43100. /* Prints macro name "BLEND" in mnote if tracing on. */
43101. **IF CURRENT_NEST_LEVEL > NESTING_LIMIT**
43102. **THEN**
43103. **CALL POP_OLD_BLOCK ( ; )**
43104. **ELSE**
43105. **IF CURRENT_NEST_LEVEL = 0**
43106. **THEN**
43107. **MNOTE (6, 'STC4301 NO BLOCK ACTIVE--"BLEND" IGNORED')**
43108. **ELSE**
43109. **I := CURRENT_NEST_LEVEL**
43110. **IF USER_NAME = ''**
43111. **THEN**
43112. **WHILE I > 0 and BLOCK_NAME(I) # USER_NAME**
43113. **DO /* (Termination: I is decremented--must eventually become**
43114. << EOC) */
43115. **IF I = 0**
43116. **THEN**
43117. **MNOTE (6, 'STC4302 BLOCK ACTIVE NAMED '|| USER_NAME ||**
43118. **'--"BLEND" IGNORED')**
43119. **MEXIT**
43120. **FI**
43121. **DOCASE BLOCK_TYPE(I) only**
43122. **of**
43123. **CASE 'IF'**
43124. **CALL FI ( ; USER_NAME)**
43125. **ESAC**
43126. **CASE 'DO'**
43127. **CALL OD ( ; USER_NAME)**
43128. **ESAC**
43129. **CASE 'CASE'**
43130. **CALL ESAC ( ; USER_NAME)**
43131. **ESAC**
43132. **CASE 'DOCASE'**
43133. **CALL ESACOD ( ; USER_NAME)**
43134. **ESAC**
43135. **CASE 'PROC'**
43136. **CALL CORP ( ; USER_NAME, RETURN=RETURN, LINK=, RESTORE=RESTORE, RC=RC)**
43137. **ESAC**
43138. **CASE 'BLOCK'**
43139. **CALL POP_OLD_BLOCK ( ; )**
43140. **ESAC**
43141. **ESAC**
43142. **END**
43143. /* (Lemma: If CURRENT_NEST_LEVEL > 0 and**
43144. **[USER_NAME = '' or = BLOCK_NAME(CURRENT_NEST_LEVEL)] at entry to**
43145. **BLEND, then CURRENT_NEST_LEVEL will be decremented by exactly**
43146. **one.) */
"FINAL" Macro -- 10 July 1973

53001. \texttt{macro FINAL ( ; )}
53002. /* Insure all blocks are closed. Then if SAVETRACE\_ON\_FIRST\_PROC,
53003. define label $\text{LASTSAV}$ to be PREV\_SAVETRACE\_AREA and EQUate
53004. PREV\_SAVETRACE\_PTR to 0. */
53005.
53006. \texttt{call TRACE\_PRINTER ( ; 'FINAL')}
53007. /* Print macro name "FINAL" in note if tracing on. */
53008. \texttt{while CURRENT\_NEST\_LEVEL > 0}
53009. \texttt{do}
53010. \texttt{if CURRENT\_NEST\_LEVEL > NESTING\_LIMIT}
53011. \texttt{then}
53012. \texttt{mnote (8, 'STRC53C1 BLEND OF OUTSTANDING BLOCK ASSUMED')}
53013. \texttt{else}
53014. \texttt{mnote (8, 'STRC5301 BLEND OF ' || BLOCK\_NAME(CURRENT\_NEST\_LEVEL) || ' \* ASSUMED')}
53015. \texttt{fi}
53016. \texttt{call BLEND ( ; )}
53017. /* (Lemma: If CURRENT\_NEST\_LEVEL > 0 and no BLEND operands are
53018. specified, CURRENT\_NEST\_LEVEL will be decremented by exactly
53019. one.) */
53020. \texttt{od} /* (Termination: CURRENT\_NEST\_LEVEL decreases monotonically
53021. and therefore must eventually become \texttt{0}.) */
53022.
53023. \texttt{if SAVETRACE\_ON\_FIRST\_PROC}
53024. \texttt{then}
53025. \texttt{if PREV\_SAVETRACE\_PTR = 'FIRSTSV'}
53026. \texttt{then} /* No non-OS proc occurred; generate dummy area. */
53027. \texttt{generate ('$FIRSTSV DC OF'0',XF'FFFFFFF',A(0,0) DUMMY SAVEAREA')}
53028. \texttt{generate ('$LASTSAV EQU $FIRSTSV')}
53029. \texttt{else}
53030. \texttt{generate ('$LASTSAV EQU ' || PREV\_SAVETRACE\_AREA)}
53031. \texttt{generate (PREV\_SAVETRACE\_PTR || ' EQU 0')}
53032. \texttt{fi}
53033. \texttt{fi}
53034. \texttt{fi}
53035. \texttt{fi}
53036. \texttt{fi}
53037. \texttt{fi}
53038. \texttt{mend}
"EXIT" macro -- 10 July 1973

55001. macro EXIT (USER_NAME; EXIT_TARGET)
55002.    /* Find exit point. Generate branch. */
55004.      call TRACE_PRINTER ( ; 'EXIT')
55005.      /* Print macro name "EXIT" in mnote if tracing on. */
55006.      if CURRENT_NEST_LEVEL > NESTING_LIMIT
55007.      then
55008.      mexit
55009.      fi
55010.      call EXIT_FIND ( ; EXIT_TARGET)
55011.      /* Set ULTIMATE_BRANCH_LABEL to point to end of block whose name
55012.      is the argument and ,all as needed as it label; if no such block,
55013.      issue message and set ERROR_OCCURRED. */
55014.      if ~ ERROR_OCCURRED
55015.      then
55016.      generate (USER_NAME || $ B $ || ULTIMATE_BRANCH_LABEL)
55017.      fi
55018.      send
"PROC" Macro -- 5 July 1973

81001. MACRO PROC (USER_NAME = LINKAGE, ID=, BASE=, NCBE=, SAVE=, DEBUG=, XIT=)
81002. /* Defines a procedure block. If LINKAGE=OS is specified, standard
81003. OS save area conventions are followed; otherwise a simple non-
81004. linked save area is provided. A base register is established
81005. (unless BASE=NONE is specified under CS_LINKAGE or BASE= is
81006. omitted on local PROCs). Register values upon entry are
81007. saved to allow restoring at CORP time. */
81008.
81009. bit
81010. FIRST_PROC, /* Indicates whether this is the first PROC macro coded
81011. in this assembly. */
81012. FIRST_VALUE_KNOWN, /* Indicates whether the first SAVE= operand was
81013. a self-defining term (or omitted) or if it was symbolic. */
81014. OS_LINKAGE, /* Indicates whether LINKAGE=(OS, ..) was
81015. entered. */
81016. SPECIAL_PREF, /* Indicates whether the BLOCK_LABEL_PREF was
81017. changed to the special debugging form "#Pgr": */
81018. USING13, /* Indicates whether the base register is GPR13. */
81019. MULTIBASE, /* Indicates more than one base register was
81020. requested, but adcons for loading have not yet been generated. */
81021. WORKREG_USED /* Indicates whether the value in WORKREG was
81022. modified and its contents saved in register 0. */
81023.
81024. char
81025. COMMA2, /* Contain "," and "H" respectively if a range of
81026. registers is to be saved, or the null string if a single register
81027. is to be saved. Used to generate "STM" or "STM" instruction. */
81028. FIRST, LAST, /* First and last register in range to be saved. */
81029. LOCAL_POINT, OS_POINT, /* Instruction segments to generate
81030. store instruction for proper save area. */
81031. PREVIOUS_DEBUG_VECTOR, /* Holds value of debug switches on entry to
81032. PROC macro for restoring on exit from CORP. */
81033. PROC_ID_BYTE, /* Value of hex proc number (PROC_COUNTER in hex)
81034. used in various debugging instructions. */
81035. SAVE_LENGTH, /* Length of save area (in words), except length of
81036. register part only for local PROCs. */
81037. SAVETYPE, /* Type of save area generated: FULL (savetrace), OSSAVE,
81038. NORMAL, NORMALHDR, TRUNC, TRUNCHDR, or NONE. */
81039. SAVEREG, /* Register (work or base) which is pointing at new
81040. save area before chaining. */
81041. WORKREG /* Register used for setting up linkage, etc. */
81042.
81043. int
81044. OFFSET, /* Offset (in words) to either FIRST (if FIRST_VALUE_KNOWN),
81045. or to GP0 within save area. */
81046. OFFSET_TO_GP0, /* Offset in words to GP0 within save area. */
81047. SAP, SAL /* Register number to go into first register word of
81048. save area; this, for example, could be 14 even though FIRST
81049. is a symbolic register of unknown value at macro expansion time.
81050. SAL is similar but for last register. */

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

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

PROC

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

PROC_CODE := (@PROC_SCAN_OPTIONS)

if LINKAGE(1) # 'OS' and # ' '

then

mnote ($, 'STC8101 LINKAGE=' || LINKAGE(1) ||
 ' INVALID--"OS" ASSUMED')

else

fi

if LINKAGE(2) # 'CSECT' and # ' '

then

mnote ($, 'STC8102 SECOND LINKAGE OPERAND IGNORED')

else

WORKREG := WORK,

if WORKREG = 'NONE' or = ' ' and

then

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

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

if OS_LINKAGE

then

WORKREG := '2'

else

WORKREG := '1'

fi

EXIT_SEVERITY := EXIT

if LINKAGE(1) # 'OS' and # ' '

then

mnote ($, 'STC8101 LINKAGE=' || LINKAGE(1) ||
 ' INVALID--"OS" ASSUMED')

else

fi

CALL_PROC_DEBUG_SET

if SPECIAL_PREFIX

then

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

PROC_COUNTER := PROC_COUNTER + 1

if PROC_COUNTER < 255

then

mnote ($, 'STC8103 PROC_COUNTER=' || PROC_COUNTER ||
 ' INVALID--OVERFLOW')

else

fi

else

PROC_ID_BYTE := '00'

fi
PROC Macro

PROC_DEBUG_SET

81167. /* Save the previous DEBUG specifications so they can be restored at
81168. CORP time. Scan the DEBUG= suboperands setting the debug flags
81169. indicated. If GLOBAL is specified save null restore value to
81170. suppress it. */
81171.
81172.
81173.
81174.
81175.
81176. / * Indicates whether "GLOBAL" has been found as an
81177. operand of DEBUG= */
81178.
81179.
81180. /* Set true if SAVETRACE is to be turned on; set
81181. false if SAVETRACE is to be turned off; else not set. */
81182.
81183. /* Save current value of debug switches. */
81184.
81185. /* List suboperand index. */
81186.
81187.
81188. /* Indicates whether "GLOBAL" has not yet been found. */
81189.
81190.
81191.
81192. while I I DEBUG
81193. do
81194. case DEBUG(I)
81195. of
81196. case ('GLOBAL', 'GBL')
81197. GLOBAL := true
81198. esac
81199. case ('BLOCKNAMES', 'BN')
81200. DEBUG_BLOCKNAMES_REQD := true
81201. /* BLOCKNAMES causes the name of each block to be generated as an
81202. in-line character constant at the start of each block (of any type,
81203. not just BLOCK macros) for ease of locating code in dumps. */
81204. esac
81205. case ('PROCNAMES', 'PN')
81206. DEBUG_PROCNAMES_REQD := true
81207. /* PROCNAMES causes the name of each PROC to be generated as an
81208. in-line character constant at the start of the PROC for ease of
81209. locating code in dumps. */
81210. esac
81211. case ('LISTBLOCKS', 'LB')
81212. DEBUG_LISTBLOCKS_REQD := true
81213. /* LISTBLOCKS causes the name, number, and depth of each block to be
81214. generated in an mnote at the start and end of the block. */
81215. esac
81216. case ('NOBLOCKNAMES', 'NBN')
81217. DEBUG_BLOCKNAMES_REQD := false
81218. esac
81219. case ('NOPROCNAMES', 'NPN')
81220. DEBUG_PROCNAMES_REQD := false
81221. esac
81222. case ('NOLISTBLOCKS', 'NLB')
81223. DEBUG_LISTBLOCKS_REQD := false
81224. esac
81225. case ('BLOCKCOUNTS', 'BC')
81226. DEBUG_BLOCKCOUNTS_REQD := true
81227. /* BLOCKCOUNTS causes counters to be kept on the number of executions
81228. of all blocks. */
81229. esac
81230. case ('NOBLOCKCOUNTS', 'NBC')
81231. DEBUG_BLOCKCOUNTS_REQD := false
81232. esac
81233. case ('PROCOUNTS', 'PC')
81234. DEBUG_PROCOUNTS_REQD := true
81235. /* PROCOUNTS causes counters to be kept on the number of executions
81236. of all PROC blocks. */
81237. esac
81238. case ('NOPROCOUNTS', 'NPC')
81239. DEBUG_PROCOUNTS_REQD := false
81240. esac
81241. esac
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 ('NO_PROCTRACE', '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 ('NO_CORPVALUES', 'NCV')
  DEBUG_CORPVALUES_REQD := false
CASE ('SAVETRACE', 'ST')
  DEBUG_SAVETRACE_REQD, DEBUG_PROCNAMES_REQD, DEBUG_LISTBLOCKS_REQD,
  DEBUG_BLOCKCOUNTS_REQD, DEBUG_PROCTRACE_REQD,
  DEBUG_CORPVALUES_REQD, SAVETRACE_VALUE := (DEBUG(I) = 'ALL')
  /* Set (or reset) all main debug switches. */
CASE ('NO_SAVETRACE', 'NST')
  SAVETRACE_VALUE := false
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 ('NO_MACRONAMES', 'NMN')
  DEBUG_MACRONAMES_REQD := false
CASE ('DEBUGMACROS', 'DM')
  DEBUG_MACRONAMES_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_MACRONAMES_REQD := false
I := I + 1 /* Go on to next suboperand. */
/* (Termination: I is incremented, N'DEBUG is fixed in loop;
I must eventually exceed N'DEBUG.) */
"PROC" MACRO -- 5 July 1973

81310.  if SAVETRACE_CHECK
81311.    then
81312.      if SAVETRACE_VALUE
81313.        then
81314.          if FIRST_PROC
81315.            then
81316.              if OS_LINKAGE
81317.                then
81318.                  SAVETRACE_ON_FIRST_PROC := true
81319.                  DEBUG_SAVETRACE_REQD := true
81320.                  note (4, 'STBC8103 WARNING—SAVETRACE REQUIRES "FINAL" MACRO')
81321.                  else
81322.                    note (8, 'STBC8106 SAVETRACE REQUIRES FIRST PROC TO BE LINKAGE=OS')
81323.                fi
81324.              else /* Not first PROC. */
81325.                if SAVETRACE_ON_FIRST_PROC
81326.                  then /* SAVETRACE is being resumed. */
81327.                    DEBUG_SAVETRACE_REQD := true
81328.                  else
81329.                    note (8, 'STBC8105 SAVETRACE MUST BE SPECIFIED ON FIRST PROC')
81330.                  fi
81331.              fi
81332.            else
81333.              DEBUG_SAVETRACE_REQD := false
81334.              fi
81335.        fi
81336.    if GLOBAL
81337.      then
81338.        PREVIOUS_DEBUG_VECTOR := *
81339.      /* Null value suppresses restore by CORP. */
81340.      fi
81341.  fi

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

```
PROC HEADER

 LoginComponent = CSECT

if LINKAGE = (***,CSECT)
  generate a "CSECT" and "USING *,15" if ID

if LINKAGE = (OS,***)
  generate inline ID

if ID = specified.
  if OS_LINKAGE and ID <> 'NONE' ok

if OS_LINKAGE
  then
    SECT := 'PRIVATE' /* Default name if none specified. */
  else
    SECT := USER_NAME
    LENGTH := #SECT_NAME
  fi

if OS_LINKAGE
  then
    SECT := 'PRIVATE' /* Default name if none specified. */
  else
    SECT := USER_NAME
    LENGTH := #SECT_NAME
  fi

if ID = specified.
  QUOTE := '

if ID already contains surrounding quotes.
  QUOTE := 
  LENGTH := -2 /* Subtract 2 for the quotes. */

if OS_LINKAGE
  then
    LENGTH := ((K'*'/2)+1) /* Round up to odd number. */
    generate (DC AL(' ', LENGTH)), CL' | LENGTH |
    "" | SECT | ""
  else
    SECT := USER_NAME
    LENGTH := #SECT_NAME
    LENGTH := ((LENGTH+1)/2)+1 /* Round up to even number. */
    generate (DC CL' | LENGTH | QUOTE | ID | QUOTE)

if ID = specified.
  QUOTE := 

if ID already contains surrounding quotes.
  QUOTE := 
  LENGTH := -2 /* Subtract 2 for the quotes. */

if OS_LINKAGE
  then
    LENGTH := ((K'*'/2)+1) /* Round up to odd number. */
    generate (DC AL(' ', LENGTH)), CL' | LENGTH |
    "" | SECT | ""
  else
    SECT := USER_NAME
    LENGTH := ((LENGTH+1)/2)+1 /* Round up to even number. */
    generate (DC CL' | LENGTH | QUOTE | ID | QUOTE)

if ID = specified.
  QUOTE := 

if ID already contains surrounding quotes.
  QUOTE := 
  LENGTH := -2 /* Subtract 2 for the quotes. */

if OS_LINKAGE
  then
    LENGTH := ((K'*'/2)+1) /* Round up to odd number. */
    generate (DC AL(' ', LENGTH)), CL' | LENGTH |
    "" | SECT | ""
  else
    SECT := USER_NAME
    LENGTH := ((LENGTH+1)/2)+1 /* Round up to even number. */
    generate (DC CL' | LENGTH | QUOTE | ID | QUOTE)

if ID = specified.
  QUOTE := 

if ID already contains surrounding quotes.
  QUOTE := 
  LENGTH := -2 /* Subtract 2 for the quotes. */

if OS_LINKAGE
  then
    LENGTH := ((K'*'/2)+1) /* Round up to odd number. */
    generate (DC AL(' ', LENGTH)), CL' | LENGTH |
    "" | SECT | ""
  else
    SECT := USER_NAME
    LENGTH := ((LENGTH+1)/2)+1 /* Round up to even number. */
    generate (DC CL' | LENGTH | QUOTE | ID | QUOTE)

if ID = specified.
  QUOTE := 

if ID already contains surrounding quotes.
  QUOTE := 
  LENGTH := -2 /* Subtract 2 for the quotes. */

if OS_LINKAGE
  then
    LENGTH := ((K'*'/2)+1) /* Round up to odd number. */
    generate (DC AL(' ', LENGTH)), CL' | LENGTH |
    "" | SECT | ""
  else
    SECT := USER_NAME
    LENGTH := ((LENGTH+1)/2)+1 /* Round up to even number. */
    generate (DC CL' | LENGTH | QUOTE | ID | QUOTE)

if ID = specified.
  QUOTE := 

if ID already contains surrounding quotes.
  QUOTE := 
  LENGTH := -2 /* Subtract 2 for the quotes. */

if OS_LINKAGE
  then
    LENGTH := ((K'*'/2)+1) /* Round up to odd number. */
    generate (DC AL(' ', LENGTH)), CL' | LENGTH |
    "" | SECT | ""
  else
    SECT := USER_NAME
    LENGTH := ((LENGTH+1)/2)+1 /* Round up to even number. */
    generate (DC CL' | LENGTH | QUOTE | ID | QUOTE)

if ID = specified.
  QUOTE := 

if ID already contains surrounding quotes.
  QUOTE := 
  LENGTH := -2 /* Subtract 2 for the quotes. */

if OS_LINKAGE
  then
    LENGTH := ((K'*'/2)+1) /* Round up to odd number. */
    generate (DC AL(' ', LENGTH)), CL' | LENGTH |
    "" | SECT | ""
  else
    SECT := USER_NAME
    LENGTH := ((LENGTH+1)/2)+1 /* Round up to even number. */
    generate (DC CL' | LENGTH | QUOTE | ID | QUOTE)

if ID = specified.
  QUOTE := 

if ID already contains surrounding quotes.
  QUOTE := 
  LENGTH := -2 /* Subtract 2 for the quotes. */

if OS_LINKAGE
  then
    LENGTH := ((K'*'/2)+1) /* Round up to odd number. */
    generate (DC AL(' ', LENGTH)), CL' | LENGTH |
    "" | SECT | ""
  else
    SECT := USER_NAME
    LENGTH := ((LENGTH+1)/2)+1 /* Round up to even number. */
    generate (DC CL' | LENGTH | QUOTE | ID | QUOTE)

if ID = specified.
  QUOTE := 

if ID already contains surrounding quotes.
  QUOTE := 
  LENGTH := -2 /* Subtract 2 for the quotes. */

if OS_LINKAGE
  then
    LENGTH := ((K'*'/2)+1) /* Round up to odd number. */
    generate (DC AL(' ', LENGTH)), CL' | LENGTH |
    "" | SECT | ""
  else
    SECT := USER_NAME
    LENGTH := ((LENGTH+1)/2)+1 /* Round up to even number. */
    generate (DC CL' | LENGTH | QUOTE | ID | QUOTE)

if ID = specified.
  QUOTE := 

if ID already contains surrounding quotes.
  QUOTE := 
  LENGTH := -2 /* Subtract 2 for the quotes. */

if OS_LINKAGE
  then
    LENGTH := ((K'*'/2)+1) /* Round up to odd number. */
    generate (DC AL(' ', LENGTH)), CL' | LENGTH |
    "" | SECT | ""
  else
    SECT := USER_NAME
    LENGTH := ((LENGTH+1)/2)+1 /* Round up to even number. */
    generate (DC CL' | LENGTH | QUOTE | ID | QUOTE)

if ID = specified.
  QUOTE := 

if ID already contains surrounding quotes.
  QUOTE := 
  LENGTH := -2 /* Subtract 2 for the quotes. */

if OS_LINKAGE
  then
    LENGTH := ((K'*'/2)+1) /* Round up to odd number. */
    generate (DC AL(' ', LENGTH)), CL' | LENGTH |
    "" | SECT | ""
  else
    SECT := USER_NAME
    LENGTH := ((LENGTH+1)/2)+1 /* Round up to even number. */
    generate (DC CL' | LENGTH | QUOTE | ID | QUOTE)

if ID = specified.
  QUOTE := 

if ID already contains surrounding quotes.
  QUOTE := 
  LENGTH := -2 /* Subtract 2 for the quotes. */

if OS_LINKAGE
  then
    LENGTH := ((K'*'/2)+1) /* Round up to odd number. */
    generate (DC AL(' ', LENGTH)), CL' | LENGTH |
    "" | SECT | ""
  else
    SECT := USER_NAME
    LENGTH := ((LENGTH+1)/2)+1 /* Round up to even number. */
    generate (DC CL' | LENGTH | QUOTE | ID | QUOTE)

if ID = specified.
  QUOTE := 

if ID already contains surrounding quotes.
  QUOTE := 
  LENGTH := -2 /* Subtract 2 for the quotes. */

if OS_LINKAGE
  then
    LENGTH := ((K'*'/2)+1) /* Round up to odd number. */
    generate (DC AL(' ', LENGTH)), CL' | LENGTH |
    "" | SECT | ""
  else
    SECT := USER_NAME
    LENGTH := ((LENGTH+1)/2)+1 /* Round up to even number. */
    generate (DC CL' | LENGTH | QUOTE | ID | QUOTE)

if ID = specified.
  QUOTE := 

if ID already contains surrounding quotes.
  QUOTE := 
  LENGTH := -2 /* Subtract 2 for the quotes. */

if OS_LINKAGE
  then
    LENGTH := ((K'*'/2)+1) /* Round up to odd number. */
    generate (DC AL(' ', LENGTH)), CL' | LENGTH |
    "" | SECT | ""
  else
    SECT := USER_NAME
    LENGTH := ((LENGTH+1)/2)+1 /* Round up to even number. */
    generate (DC CL' | LENGTH | QUOTE | ID | QUOTE)

if ID = specified.
  QUOTE := 

if ID already contains surrounding quotes.
  QUOTE := 
  LENGTH := -2 /* Subtract 2 for the quotes. */
```

C-17
PROC Macro -- 5 July 1973

81416. proc proc_reg_save
81417. /* If SAVE=NONE and doing SAVE_TRACE generate store of all registers.
81418. If SAVE=NONE and no SAVE_TRACE, do nothing. For other than
81419. SAVE=NONE, extract from SAVE_TRACE register list to be saved.
81420. Link type and size of save area and put into SAVE_TYPE and
81421. SAVE_LENGTH. Generate instruction to store registers. */
81422. last = 1 /* Temporary. */
81423. first, last := 1
81424. offset, saf, sal, offset_to_gpro := 0
81425. /* Initialize save request information variables. */
81426. if save = 'none'
81427. then
81428. if debug
81429. then
81430. save_type := 'FULL'
81431. save_length := 15
81432. generate (LABEL || 'STM 14,12,' || block_label_prefix || 'SV+12')
81433. label :=
81434. else
81435. call proc_set_save_info
81436. /* Collect following save request information. Put character string
81437. first of first register to be saved in first, last in last, 'N' into
81438. MULT, and ',' into COMMA2; if only a single register is to be saved,
81439. MULT, COMMA2 get null strings. Register number which could
81440. go into first word of register part of save area goes into SAF, last
81441. into SAL. (Those, for example, could be 14 and 12 while FIRST and
81442. LAST are symbolic register designations of unknown value at macro
81443. expansion time.) Set FIRST_VALUEKNOWN if FIRST is not symbolic. */
81444. call proc_set_save_type
81445. /* Put type of save area to be generated into SAVE_TYPE. Set
81446. save_length with length (in words) of save area (except only length
81447. of register part for non-OS_LINKAGE areas). Offset in save area
81448. (in words) of register 0 is put into offset_to_gpro. Offset of
81449. offset to FIRST (if FIRST_VALUEKNOWN) or else of register 0 is
81450. put into offset. Also set LOCAL_POINTER to ' ' and OS_POINTER to
81451. '13' if OS_LINKAGE, else LOCAL_POINTER to
81452. (block_label_prefix || 'SV+12') and OS_POINTER to ' '. Thus
81453. LOCAL_POINTER || OFFSET || 'q' || OS_POINTER refers to the given
81454. offset in the proper area. */
81455. if save_type = 'full'
81456. then
81457. generate (LABEL || 'STM 14,12,' || block_label_prefix || 'SV+12')
81458. /* Save all registers on FULL (savetrace-required) save area. */
81459. else
81460. if first_value_known
81461. then
81462. i := offset_to_gpro * 4 /* calculate offset in bytes. */
81463. generate (LABEL || 'ST' || MULT || ' ' || FIRST || ' ' ||
81464. LAST || COMMA2 || LOCAL_POINTER || 'q' || OS_POINTER)
81465. else
81466. generate (LABEL || 'ST' || MULT || ' ' || FIRST || ' ' ||
81467. LAST || COMMA2 || LOCAL_POINTER || 'q' || Offset ||
81468. '-((' || FIRST || '(2/16*16)+q' || OS_POINTER))
81469. endif
81470. endif
81471. endif
81472. endif
81473. label := ""
PROC_SET_SAVE_INFO
PROC
/* Collect following save request information. Put character string
name of first register to be saved in FIRST, last in LAST, "*" into
MULT, and ",' into COMMA2; if only a single register to be saved,
LAST, MULT, and COMMA2 get null strings. Register number which could
go into first word of register part of save area goes into SAF, last
into SAL. (These, for example, could be 14 and 12 while FIRST and
LAST are symbolic register designations of unknown value at macro
expansion time.) Set FIRST_VALUE_KNOWN if FIRST is not symbolic. */

FIRST_VALUE_KNOWN := false
MULT := 'I'
COMMA2 :=

/* Assumed values. */
if T'SAVE(1) = 'O' /* At least first suboperand is omitted. */
then
FIRST := '14'
SAF := 14
LAST := '12'
SAL := 12
/* Default is to save all registers 14 through 12. */
else
if T'SAVE(1) = 'N' /* Self-defining term. */
then
SAF := SAVE(1) /* Store it as a number. */
*/ 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. */
SAP := 14 /* Just say first of save area is register 14. */
FIRST_VALUE_KNOWN := false
fi
else /* Second suboperand is omitted. */
then
LAST := 12
MULT, COMMA2 :=
if FIRST_VALUE_KNOWN
then
LAST := SAP /* 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
end
**PROC** Macro -- 5 July 1973

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDESAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE_SAVE_TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE

PROC DECIDE SAVE TYPE
PROG PROC_ESTABLISH_BASE

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

BIT
    INLIN_SAVepad /* Indicates whether an inline save area is to be generated. */

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

INT I, J /* Temporary. */
81621. if BASE = 'NONE' and (OS_LINKAGE or BASE != '1')
81622.  then /* Generate a base register. */
81623.  BASEREG := BASE(1)
81624.  INLIN_SAVEAREA := (SAVE # 'NONE' and SAVE(3) = 1)
81625.  if BASEREG = '13' and (INLINE_SAVEAREA and OS_LINKAGE)
81626.  then
81627.  endif
81628.  if BASEREG = 12
81629.  then /* A base register specified. */
81630.  endif
81631.  if OS_LINKAGE
81632.  then
81633.  endif
81634.  fi
81635.  fi
81636.  fi
81637.  fi
81638.  if BASEREG = '13'
81639.  then
81640.  BASEREG := WORKREG
81641.  if OS_LINKAGE
81642.  then
81643.  endif
81644.  endif
81645.  fi
81646.  fi
81647.  J := 0
81648.  K := 2
81649.  while I <= N'BASE
81650.  do
81651.  if BASE(I) = '13'
81652.  then
81653.  endif
81654.  Labeled (3, 'STRC109 REGISTER 13 INVALID--IGNORED')
81655.  label
81656.  INLINELABEL (LABEL || 'L' || BASE(I) || 'U' || IC)
81657.  label
81658.  J := 1
81659.  J := J + 1
81660.  endif
81661.  fi
81662.  endif
81663.  if BASE(R) = 'NONE'
81664.  then
81665.  endif
81666.  if INLINSAVEAREA
81667.  then
81668.  endif
81669.  fi
81670.  endif
81671.  if OS_LINKAGE
81672.  then
81673.  endif
81674.  if N'BASE > 1
81675.  then
81676.  endif
81677.  if N'BASE > 1
81678.  then
81679.  endif
81680.  if N'BASE > 1
81681.  then
81682.  endif
81683.  if N'BASE > 1
81684.  then
81685.  endif
81686.  if N'BASE > 1
81687.  then
81688.  endif
81689.  if N'BASE > 1
81690.  then
81691.  endif
81692.  if N'BASE > 1
81693.  then
81694.  endif
81695.  if N'BASE > 1
81696.  then
81697.  endif
PROC GEN SAVEAREA

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

if SAVE_TYPE = 'OSSAVE'
then
call 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 this is the first proc and SAVETRACE requested. */
else
if (SAVE # 'NONE' and SAVE_TYPE # 'NONE') OR
DBG_SAVETRACE_REQD
then
call PROC_GEN_LOCAL_SAVEAREA /* Generate local PROC save area according to SAVE_TYPE and SAVE_LENGTH and, if SAVETRACE requested, provide static save area chaining. */
fi
fi
else
/* OVERRIDE

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

81720. PROC PROC_GEN_OSSAVE_AREA
81721.      /* Generate OS save area and chain it up following OS linkage
81722.      conventions. Also link up static chain of local save areas if
81723.      SAVETRACE requested. */
81724.     call PROC_DEFHAS_NEW_OSSAVE
81725.     /* Generate linkage, out-of-line, or dynamic save area and point to
81726.     it via BASEREG or WORKREG; put register name in SAVEREG. */
81727.     if DEBUG_SAVETRACE_ADDR and FIRST_PROC
81728.     then
81729.     /* Static chain of local save areas must be linked to OS save areas. */
81730.     generate (LABEL $13,$FIRSTSV+4)
81731.     LABEL :=
81732.     generate ('MVC 8(4,13,=A(SFIRSTSV))')
81733.     generate ('L 13,=A(SLASTSAV)')
81734.     PREV_SAVETRACE_PTR := 'FIRSTSV'
81735.     PREV_SAVETRACE_AREA := '0'
81736.     fi
81737.     fi
81738.     generate (LABEL 'ST ' II SAVEREG '13')
81739.     LABEL :=
81740.     generate ('ST 13,=A(SAVEMPL)')
81741.     if USING13
81742.        then /* 13 now loaded--issue USING. */
81743.        generate ('USING ' II BLOCK_LABEL_PREFIX 'ISV,13')
81744.        fi
81745.     if DEBUG_PROCTRACE_REQD and FIRST_PROC and SAVE(3) "
81746.     then
81747.        /* Store pointer to PROC trace vector in word 1 of OS save area. */
81748.        if WORK = ' Thực and ~ WORKREG_USED
81749.           then
81750.              generate ('LR 0,WORKREG')
81751.              fi
81752.        fi
81753.        generate ('LA ' II WORKREG 'STRACE')
81754.        WORKREG := base
81755.       fi
81756.     fi
81757.     end
```assembly
/*PROC DEFINE OSSAVE

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. */

Action

 evaluating (LABEL || ' LA 0,' || SAVE_LENGTH || ')**'

if WORK == 'NONE'

if LAST == '*' or FIRST == '?'

SAVEREG := BASEREG

else

SAVEREG := WORKREG

endif

if DBJ; ROCTRACE_ADDR

endif

else

if SAVE(3) == 'OUT'

endif

endif

endif

/* Area has been generated and address is in SAVEREG register. */
```
"PROC" Maco  --  5 July 1973

01833. proc proc_jin_local_savearea
01834.   /* Generate local PROC save area according to SAVE_TYPE and SAVE_LENGTH
01835.   and, if SAVETRACE requested, provide static save area chaining. */
01836.   char pwd_ptr /* Name used for next save area. */
01837.
01838.   if label = '
01839.   then
01840.     label := block_label_prefix || 'DD'
01841.   generate ('  B  ' || label)
01842.   fi
01843.   generate (block_label_prefix || 'SV DS 00')
01844.   if save_type = 'FULL' or SAVE_TYPE[6,3] = 'HDR'
01845.   then /* word one should contain PROC count and ID byte. */
01846.     generate ('  DC X'PF' || proc_id_byte || '0000' ||
01847.     '  FLAG (FNENTERED, FFINISHED), ID, COUNT')
01848.   fi
01849.   if save_type = 'FULL'
01850.   then
01851.     FWD_PTR := block_label_prefix || 'WRT'
01852.     prev_savearea_ptr := prev_savearea_area || 'EQU ' ||
01853.     block_label_prefix || 'SV')
01854.     generate ('  DC A(' || prev_savearea_area || ')')
01855.     prev_savearea_ptr := #id_ptr
01856.     /* Save label used as forward pointer. */
01857.     prev_savearea_area := block_label_prefix || 'SV'
01858.     /* Same name of this save area. */
01859.
01860.   /*
01861.   generate ('  DC (' || save_length || ')' ||
01862.   call proc_mdlfunc Area_name
01863.   goto

...
**PROC** Macro -- 5 July 1973

```
81865. PROC_DEBUG_STUFF
81866. /* Generate trace and count code for debugging. */
81867. bit PCT_GENNED_WITH_VECTOR /* Indicates whether -PCT labeled halfword
81868. which holds PROC counter was generated following the trace vector. */
81869. UCHAR COUNT_SPOT /* Suffix of label for PROC counter. */
81870. if DEBUG_PROCTrace_REQD
81871. then
81872. TRACe_GENERATED = true
81873. endif
81874. if DEBUG_PROCPA院COUNTERS_REQD
81875. then
81876. PC_COUNT_GENERATED = true
81877. endif
81878. endif
```

```
81879. if DEBUG_PROCPA院COUNTERS_REQD
81880. then
81881. if TRACe_GENERATED
81882. then
81883. if WORK = 'NONE' and WORKREG
81884. then
81885. generate (LABEL || ' LR 0,' || WORKREG)
81886. endif
81887. endif
81888. endif
81889. endif
81890. endif
81891. endif
81892. endif
81893. endif
81894. endif
81895. endif
81896. endif
81897. endif
81898. endif
81899. endif
81900. endif
81901. endif
81902. endif
81903. endif
81904. endif
81905. endif
81906. endif
81907. endif
81908. endif
81909. endif
81910. endif
81911. endif
81912. endif
81913. endif
81914. endif
81915. endif
81916. endif
81917. endif
81918. endif
81919. endif
81920. endif
81921. endif
81922. if DEBUG_PROCPA院COUNTERS_REQD and DEBUG_BLOCKCOUNTS_REQD
81923. then
81924. if OS_LINKAGE or (SAVE = 'NONE' and DEBUG_SAVETRACE_REQD)
81925. then
81926. if PCT_GENNED_WITH_VECTOR
81927. then
81928. generate (LABEL || ' LR 0,' || WORKREG)
81929. endif
81930. endif
81931. endif
81932. endif
81933. endif
81934. endif
81935. endif
81936. endif
81937. endif
81938. endif
81939. endif
81940. endif
81941. endif
81942. endif
```

```
81943. ifdef
81944. endif
```
PROC PROC_INFO_SAVE

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

81949. OPERAND1(CURRENT_NEST_LEVEL) := FIRST
81950. OPERAND2(CURRENT_NEST_LEVEL) := LAST
81951. OPERAND3(CURRENT_NEST_LEVEL) := SAFE_LENGTH
81952. OPERAND4(CURRENT_NEST_LEVEL) := PREVIOUS_DEBUG_VECTOR
81953. I := OFFSET_TO_CORP + 50
81954. /* Bias value by 50 and convert to two-digit character string. */
81955. INFORMATION(CURRENT_NEST_LEVEL) :=
81956. I || GS_LINEAGE || (SAVE(3) = 'DYNAN') || FIRST_VALUE_EARNE ||
81957. PROC_NUM || SPECIAL_PREFIX
81958. CORE
PROC Macro

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

int I, J /* Preparations. */

if MULTIBASE /* Multibase adcons required but not yet generated. */
    if LABEL = "" /* Search around adcons. */
        LABEL := BLOCK_LABEL_PREFIX || 'FF'
        generate ('B' || LABEL)
    fi
    I := 2
    J := 4096
    X := BLOCK_LABEL_PREFIX || 'MBR'
while I ≤ 3^BASE
    generate (X || DC A(" || BLOCK_LABEL_PREFIX || 'MBR' || I
    X := "
    I := I + 1
    J := J + 4096
od /* Termination: I is incremented during loop, N'BASE is fixed; I must eventually exceed N'BASE. */
MULTIBASE := false
fi
C-89
/* Defines the end of a procedure block. The register or registers indicated by RESTORE are restored with the exception of those listed in RESTORE=, if RESTORE= is omitted, all saved registers are restored (except those in the RETURN= list). The return code is set from the RC= operand and return is made to the address specified by the LINK= operand, unless LINK=NONF is specified. */

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

/* Any registers were saved in this proc. */

/* Indicates whether any registers were saved in this proc. */

/* Indicates whether SAVE=(***,DYNAM,*** was coded on PROC. */

/* Indicates whether first register to be restored (in REST1) is other than symbolic. */

/* Indicates whether LINKAGE=(OS,..) was coded on PROC. */

/* Any outstanding label waiting to be generated. */

/* First and last registers saved at PROC time. */

/* Holds either a "M" and "," respectively or else nulls to allow generation of either a "LM" or "L" instruction. */

/* Instruction segments to generate load instructions from proper save area. */

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

/* Length of save area. */

/* First and last register to be restored. */

/* One-byte hex number used as identifier of current proc in traces and the like. */

/* Value of debug switches (packed) before encountering this PROC or [if DEBUG=(...,GLOBAL) specified] null. */

/* Print macro name "CORP" in mnote if tracing on. */

/* Generate label at first opportunity. */
call CORP_GET_PROC_INFO
  /* Get info saved at PROC macro: FIRST_REG_SAVED, LAST_REG_SAVED, */
  OS_LINKAGE, FIRST_SAVEAREA_REG, FIRST_VALUE_KNOWN, DYNAMIC_SAVEAREA, SAVE_LENGTH, OFFSET_SAVEAREA_REG, PREVIOUS_DEBUG_VECTOR, PROC_TO_BYTE, /*
  GET_OFFSET_STRING, BLOCK_LABEL_PREFIX, and ANY_REGS_SAVED. */
  call CORP_SET_RESTORE_RANGES
  /* Set RESTORE_AREA to 'SV'. Set REST1 and REST2 to RESTORE- */
  special or, if omitted, then to FIRST_ and LAST_REG_SAVED. */
call CORP_GET_SKIP_LABEL
  /* If an X1Y label is required, put it into LABEL (generating */
  any label already there). */
if OS_LINKAGE
  then
    call CORP_TEST_SKIP
      /* Move register 11 pointer to point to previous save area, saving */
      pointer to current area in register 1 if it is dynamic. */
  if DYNAMIC_SAVEAREA
    then
      call CORP_SET_DYN_SAVEAREA
      /* Issue FREE_core for dynamic core. */
  else
    call CORP_DISABLE_SAVES
      /* If DEBUGCOREVALUES_REQD, copy registers into CRP save area. If */
      CORVALUES and RETURN- (or RC=) specified, copy SV save area to BCK */
      save area, set RESTORE_AREA to 'BCK', and set BCK_AREA_REQD. */
  fi
  call CORP_SET_RESTORE_CODE
    /* If RC=value (or implied zero), load it to GPR15, except that if it */
    is in a register other than 15, leave it in that register. */
  call CORP_SAFE_RETUNING_REGS
    /* If any registers are to be restored, do the following: for the */
    register containing the return code and all those listed in */
    RETURN, store each register into the appropriate word of the */
    save area from which the ultimate LN instruction will be issued. */
  if OS_LINKAGE and DEBUG_SAVETRACE_REQD
    then
      generate (LABEL = 'IVI ' II BLOCK_LABEL_PREFIX II 'SV,X'FF')
    LABEL :='
  fi
  call CORP_SET_RESTORE_REGS
    /* Restore REGIST through REST2 from proper save area if saved. */
  if OS_LINKAGE
    then
      generate (LABEL = 'IVI 12(13),X'FF')
    LABEL :='
  fi
  if LINK # 'NONE'
    then
      generate (LABEL = '38 ' II LINK)
    LABEL :='
  fi
  call CORP_SET_REG_VALUES
    /* Set flag in previous save area to show return. */
  if OS_LINKAGE
    then
      generate LABEL = '
  fi
  call POP_OLD_BLOCK
    /* Delete PROC block from the stack. (Lemma: POP_OLD_BLOCK */
    increases CURRENT_NEST_LEVEL by exactly one.) */
  call CORP_SETTINGS_DEBUG_ENVIRONMENT
    /* Restore value of debugging switches from PREVIOUS_DEBUG_VECTOR. */
  if [Lemma: If CURRENT_NEST_LEVEL > 0 and */
    (PROC_NAME = 'I' or = BLOCK_NAME(CURRENT_NEST_LEVEL)) and */
    BLOCK_FPP(CURRENT_NEST_LEVEL) = 'PROC', then */
    CURRENT_NEST_LEVEL will be decremented by exactly one. */
  endif
endif
"CORP" macro -- 6 July 1973

83148. PROC CORP_GET_PROC_INFO
83149. /* Get info saved by PROC macro. */
83150. char x /* Temporary. */
83152. FIRST_REG_SAVED := OPERAND1(CURRENT_NEST_LEVEL)
83153. LAST_REG_SAVED := OPERAND2(CURRENT_NEST_LEVEL)
83154. X := INFORMATION(CURRENT_NEST_LEVEL)
83155. OFFSET_TO_GPREG := X[1,2] - 50 /* Stored biased by 50. */
83156. if OFFSET_TO_GPREG < 0
83157. then
83158. GPREG_OFFSET_STRING := '-' | OFFSET_TO_GPREG
83159. /* In string conversion, absolute value is taken; restore sign. */
83160. else
83161. GPREG_OFFSET_STRING := '+' | OFFSET_TO_GPREG
83162. fi
83163. OS_LINKAGE := X[3,1]
83164. DYNAMIC_SAVEMAP := X[4,1]
83165. FIRST_VALUE_KNOWN := X[5,1]
83166. SAVE_LENGTH := OPERAND3(CURRENT_NEST_LEVEL)
83167. PREVIOUS_DEBUG VECTOR := OPERAND4(CURRENT_NEST_LEVEL)
83168. ANY_REGS_SAVED := (FIRST_REG_SAVED ?
83169. PROC_ID_BYTE := X[6,2]
83170. if X[8,1]
83171. then /* Special PROC prefix. */
83172. BLOCK_LABEL_PREFIX := '$P' | PROC_ID_BYTE
83173. else /* Standard prefix. */
83174. BLOCK_LABEL_PREFIX := '+' | BLOCK_NUMBER(CURRENT_NEST_LEVEL)
83175. fi
83176. if FIRST_VALUE_KNOWN
83177. then
83178. FIRST_SAVEMAP_REG := FIRST_REG_SAVED /* Convert to integer. */
83179. else
83180. FIRST_SAVEMAP_REG := 14
83181. fi
83182. /* FIRST_SAVEMAP_REG is similar to the variable SAP of PROC. */
83183. C-92
/* Set RES1 and RES2 to RESTORE= operands if present, else to FIRST_ and LAST_2= operands. Set RESTORE_AREA to "SV". Also set MULT and COMMA= to proper values. */

/* Assume: */

FEST1 := FIRST_REG_SAVED
REST2 := LAST_REG_SAVED
MULT := "M"
COMMA2 := ""
FIRST_REST_VALUEKNOWN := FIRST_VALUEKNOWN
/* Now find out. */

if ANY_REG_SAVED

then

  if RESTORE = ""

  then

    (3, "CORC301 NO REGISTERS SAVED--RESTORE IGNORED")

  fi

else

  fi

fi

if REST2 = ""

then

  ("", MULT, COMMA2 := ""

  RESTORE_AREA := "SV"

  fi

*/

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

C-93
C-94
PROC CORP_RESTORE_ASG13

/* If current save area is dynamic, save pointer to it in GP1. In any case, load GP813 to point to previous save area. Given:

OS_LINKAGE is true. */

if DYNAMIC_SAVEAREA
then
    generate (LABEL || LR 1,13')
LABEL := **
else
    if SAVETRACE.OS_FLAG_PROC and PROC_ID_BYTE = '01'
then
    generate (LABEL || L 13,EPBTSF+4*)
else
    generate (LABEL || L 13,4(13'))
endif
LABEL := **
grep

cor
"CORP" Macro -- 6 July 1973

83278. proc CORP_SET_RETURN_CODE
83279. /* If RC=value (or implied zero), load value into GPR15, but nop if
83280. RC=(reg). Note in RC_REG what register (if any) contains RC at
83281. exit. */
83282.
83283. RC_REG := '1' /* Indicate no return code. */
83284. if RC = '1'
83285. then
83286. if OS_LINKAGE
83287. then
83288. generate (LABEL || 'SR 15,15')
83289. /* Clear 15 for normal OS return. */
83290. RC_REG := '15'
83291.
83292. fi
83293. else
83294. if RC[1,1] = '('
83295. then /* Register was specified. */
83296. RC_REG := RC(1)
83297. /* Note what register return code is in. */
83298. else /* Value was specified. */
83299. if RC = 'NONE'
83300. then
83301. if RC = '0'
83302. then
83303. generate (LABEL || 'SR 15,15')
83304. else
83305. fi
83306. generate (LABEL || 'LA 15,' || RC)
83307. LABEL := ''
83308. RC_REG := '15'
83309.
83310. fi
83311. fi
83312. fi

C-96
83314. PROC CORP_SAVE_RETIRNING_REGS
83315. /* For the register containing the return code and all those listed
83316. in RETURN, store each register into the appropriate word of the
83317. save area from which the ultimate LM instruction will occur
83318. (setting OS_POINTER and LOCAL_POINTER to indicate this save area).
83319. However, if no registers are to be restored, then this proc
83320. is a nop. */
83321. int OFFSET, I
83322. if ANY_REGS_SAVED
83323. then
83324. OS_LINKAGE := OS_POINTER := LOCAL_POINTER := LOCAL_POINTERS :=
83325. 0; LOCAL_POINTERS := BLOCK_LABEL_PREFIX RESTORE_AREA ++
83326. if RC_REG = *
83327. then /* All must be restored. */
83328. if FIRST_SAVAREA_REG < 14
83329. then /* We will not be changing RC; load it now. */
83330. OS_POINTER := OS_POINTER := LOCAL_POINTER :=
83331. LOCAL_POINTERS := LABEL (++ LABEL ++ LR 15, ++ RC_REG)
83332. OFFSET := (OFFSET_TO_GPREGS - 1) * 4
83333. GENERATE (LABEL ++ ST ++ RC_REG ++ ' ' ++ LOCAL_POINTER ++ OFFSET ++ OS_POINTER)
83334. if FIRST_SAVAREA_REG = 14 and N'RETURN > 0
83335. then
83336. WRITE ('STRT932 WARNING--NO CHECK MADE TO INSURE RETURNING REGISTERS ARE AMONG THOSE SAVED IN TRUNCATED SAVE AREA')
83337. N'RETURN = 1
83338. while I < N'RETURN
83339. if RETURN(I) = 'N' /* Self-defining term. */
83340. then
83341. OFFSET := (OFFSET_TO_GPREGS + RETURN(I) - (RETURN(I) + 2)/16*16) * 4
83342. GENERATE (LABEL ++ ST ++ LOCAL_POINTER ++ OFFSET ++ OS_POINTER)
83343. if FIRST_SAVAREA_REG < 14
83344. then
83345. GENERATE (LABEL ++ ST ++ RETURN(I) ++ ' ' ++ LOCAL_POINTER ++ OFFSET ++ OS_POINTER)
83346. GENERATE (LABEL ++ ST ++ RETURN(I) ++ ' ' ++ LOCAL_POINTER ++ OFFSET ++ OS_POINTER)
83347. fi
83348. fi
83349. I := I + 1
83350. fi
83351. fi
83352. fi
83353. fi
83354. while I < N'RETURN
83355. if RETURN(I) = 'N' /* Self-defining term. */
83356. then
83357. OFFSET := (OFFSET_TO_GPREGS + RETURN(I) - (RETURN(I) + 2)/16*16) * 4
83358. GENERATE (LABEL ++ ST ++ LOCAL_POINTER ++ OFFSET ++ OS_POINTER)
83359. if FIRST_SAVAREA_REG < 14
83360. then
83361. GENERATE (LABEL ++ ST ++ RETURN(I) ++ ' ' ++ LOCAL_POINTER ++ OFFSET ++ OS_POINTER)
83362. GENERATE (LABEL ++ ST ++ RETURN(I) ++ ' ' ++ LOCAL_POINTER ++ OFFSET ++ OS_POINTER)
83363. fi
83364. fi
83365. I := I + 1
83366. fi
83367. fi
83368. fi
83369. fi
83370. generate (LABEL ++ ST ++ RETURN(I) ++ ' ' ++ LOCAL_POINTER ++ OFFSET ++ OS_POINTER)
83371. fi
83372. fi
83373. fi
83374. fi
83375. fi
83376. od /* Termination: I is incremented; N'RETURN is fixed in loop;
83377. I must eventually exceed N'RETURN. */
83378. fi
83379. C-97
"CORP" macro -- 6 July 1973

83381. PROC CORP_FREE_DYN/savearea
83382. /* Issue FREEMAIN for dynamic save area. */
83383. Generate (LABEL || ' LA 0,' || SAVE_LENGTH || '*')
83384. LABEL := ''
83385. Generate (' FREEMAIN B,LY=(0),A=(1)'"
83386. GOTO
```assembly
03388.    Proc Corp_Restore_Registers
03389.    /* Restore registers REST1 through REST2 from proper save area if
03390.    saved. */
03391.    Int OFFSET, I /* Temporaries. */
03393.    If Any_REGS_Saved
03394.        Then
03395.            If First_REST_value_Known
03396.                Else
03397.                    Offset := REST1 /* Convert to integer. */
03398.                    OFFSET := (OFFSET + GPRO * (I - ((I + 2)/16*16)) * 4
03399.                    GPRO := (LABEL | ' L' | MOLT | ) ' REST1 | ' I
03400.                    REST2 | COMMA | LOCAL_POINTER | OFFSET | OS_POINTER
03401.                    Else
03402.                        If First_Savearea_REGS < 14
03403.                            else
03404.                                Offset := (LABEL | ' L' | MOLT | ) ' REST1 | ' I
03405.                                REST2 | COMMA | LOCAL_POINTER | '(' | REST1 | I
03406.                                GPRO_OFFSET_STRING | ')' | '4' | 'I
03407.                                Else
03408.                                    else
03409.                                        Offset := (LABEL | ' L' | MOLT | ) ' REST1 | ' I
03410.                                        REST2 | COMMA | LOCAL_POINTER | '(' | REST1 | I
03411.                                        GPRO_OFFSET_STRING | '-' | '(' | REST1 | '*' | '2')/16*16 | '4' | 'I
03412.                                        OS_POINTER
03413.                                        Label := **
03414.                                    Else
03415.                                        Label := **
03416.                                        Corp
```
"COPR" Macro -- 6 July 1973

93418. PROC CORP_RESTORE_DEBUG_ENVIRONMENT
93419. /* Restore debug flags which were in progress before the PROG (unless
93420. GLOBAL cause 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_MUTEX_REQ := PREVIOUS_DEBUG_VECTOR[1,1]
93426.     DEBUG_PROC_MUTEX_REQ := PREVIOUS_DEBUG_VECTOR[2,1]
93427.     DEBUG_MUTEX_REQ := PREVIOUS_DEBUG_VECTOR[3,1]
93428.     DEBUG_METHOD_MUTEX_REQ := PREVIOUS_DEBUG_VECTOR[4,1]
93429.     DEBUG_PROC_METHOD_MUTEX_REQ := PREVIOUS_DEBUG_VECTOR[5,1]
93430.     DEBUG_PROC_MUTEX_REQ := PREVIOUS_DEBUG_VECTOR[6,1]
93431.     DEBUG_OS_MUTEX_REQ := PREVIOUS_DEBUG_VECTOR[7,1]
93432.     DEBUG_IDLE_MUTEX_REQ := PREVIOUS_DEBUG_VECTOR[8,1]
93433.   fi
93434. end

C-100
/* CORP * Macro -- 6 July 1973

83446. P222 CORP_TRO_CRP_BCK_AREAS
83447. /* If requested, genenate CRP and BCK save areas. */
83448. /* OS_LINKAGE and DEBUG_CVALUES_REQD
83449. then /* We need a CRP save area. */
83450. if OS_LINKAGE and DEBUG_CVALUES_REQD
83451. /* Last area is now the last one generated. */
83452. LAST_AREA := BLOCK_LABEL_PREFIX || 'CRP'
83453. generate (LAST_AREA || ' DS OF')
83454. if DEBUG_SAVETRACE_REQD
83455. then
83456. generate ('DC X'FC' || PROC_ID_BYTE || '0000')
83457. if BCK_AREA_REQD
83458. LAST_AREA := BLOCK_LABEL_PREFIX || 'BCK'
83459. /* The BCK area is now the last one generated. */
83460. LAST_AREA := BLOCK_LABEL_PREFIX || 'CRP'
83461. generate (LAST_AREA || ' DS OF')
83462. if DEBUG_SAVETRACE_REQD
83463. generate ('DC X'FC' || PROC_ID_BYTE || '0000')
83464. if DEBUS_CJRPVALUES_REQD
83465. then
83466. generate ('DC X'FC' || PROC_ID_BYTE || '0000')
83467. if DEBUS_CJRPVALUES_REQD
83468. then
83469. generate ('DC X'FC' || PROC_ID_BYTE || '0000')
83470. if DEBUG_SAVETRACE_REQD
83471. /* We need a CRP save area. */
83472. if LINK = '033Z'
83473. if LINK = '033Z'
83474. if DEBUG_SAVETRACE_REQD
83475. then
83476. generate ('DC X'FC' || PROC_ID_BYTE || '0000')
83477. if DEBUG_SAVETRACE_REQD
83478. then
83479. generate ('DC X'FC' || PROC_ID_BYTE || '0000')
83480. if DEBUG_SAVETRACE_REQD
83481. then
83482. DEBUG_SAVETRACE_PTR := FWD_PTR
83483. DEBUG_SAVETRACE_PADA := LAST_AREA
83484. if DEBUG_SAVETRACE_PTR := FWD_PTR
83485. if DEBUG_SAVETRACE_PTR := FWD_PTR
83486. if DEBUG_SAVETRACE_PTR := FWD_PTR
83487. if DEBUG_SAVETRACE_PTR := FWD_PTR
83488. if DEBUG_SAVETRACE_PTR := FWD_PTR
83489. if DEBUG_SAVETRACE_PADA := LAST_AREA
83490. if DEBUG_SAVETRACE_PADA := LAST_AREA
83491. if DEBUG_SAVETRACE_PADA := LAST_AREA
83492. if DEBUG_SAVETRACE_PTR := FWD_PTR
83493. if DEBUG_SAVETRACE_PTR := FWD_PTR
83494. if DEBUG_SAVETRACE_PTR := FWD_PTR
83495. if DEBUG_SAVETRACE_PTR := FWD_PTR
83496. if DEBUG_SAVETRACE_PTR := FWD_PTR
83497. if DEBUG_SAVETRACE_PTR := FWD_PTR
83498. if DEBUG_SAVETRACE_PTR := FWD_PTR
83499. if DEBUG_SAVETRACE_PTR := FWD_PTR
83500. if DEBUG_SAVETRACE_PTR := FWD_PTR
"CORR" Macro -- 6 July 1973

03488. PEBC COMP.item_EXIT_LABEL
03489. /* If an EXIT label is required, put it into LABEL (generating
03490. any label already there). Issue note regarding EXIT references. */
03491. 03492. if EXIT_LABEL_HERE(CURRENT_NEST_LEVEL)
03493. then
03494. if LABEL = **
03495. then
03496. "GENERATE (LABEL || 'DS OX')"
03497. LABEL := BLOCK_LABEL_PREFIX || 'EXIT'
03498. if EXIT_SEVERITY = 'I'
03499. then
03500. EXIT_SEVERITY := '4'
03501. fi
03502. fi
03503. quote (EXIT_SEVERITY,
03504. "SPEC301 ONE OR MORE EXIT'S REFERENCE THIS POINT")
03505. EXIT_LABEL_REQUIRED(CURRENT_NEST_LEVEL) := false
03506. /* EXIT label will have been generated by POP_OLD_BLOCK time. */
03507. 22L2
/*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 EXIT label. */
91007.  int I /* Temporary. */
91008.  CALL 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 = ' ' or *=
91013.  then /* We must search for the right block. */
91014.  while I > 0 and REQD_NAME # BLOCK_NAME(I) and
91015.  BLOCK_TYPE(I) = 'PROC'
91016.  do
91017.  I := I - 1
91018.  od /* (Termination: I is decremented and would eventually become
91019.  < 0 even if other tests never occurred.) */
91020.  if I <= 0 or
91021.  (REQD_NAME # BLOCK_NAME(I) and *' ' and *=*)
91022.  then /* Not found in search. */
91023.  ERROR_OCCURRED := true
91024.  else /* Didn't even search; EXIT not nested. */
91025.  BLOCK_NAME(I) = BLOCK_NAME(CURRENT_NEST_LEVEL)
91026.  then /* At End or OnExit */
91027.  ERROR_OCCURRED := true
91028.  else
91029.  if REQD_NAME = BLOCK_NAME(CURRENT_NEST_LEVEL)
91030.  then /* No block active named "" || REQD_NAME || "" */
91031.  ERROR_OCCURRED := true
91032.  else
91033.  if BLOCK_NAME(I) = 'DO' and
91034.  INFORMATION(I)[6,1]
91035.  then /* Just use special PROC prefix form. */
91036.  ULTIMATE_BRANCH_LABEL := '$P' || INFORMATION(I)[6,2] || 'EXIT'
91037.  else
91038.  ULTIMATE_BRANCH_LABEL := '$' || BLOCK_NUMBER(I) || 'EXIT'
91039.  fi
91040.  fi
91041.  else /* Proc. */
91042.  if BLOCK_TYPE(I) = 'DO' and
91043.  INFORMATION(I)[6,1]
91044.  then /* No block active named "" || REQD_NAME || "" */
91045.  ERROR_OCCURRED := true
91046.  else
91047.  if BLOCK_TYPE(I) = 'PROC' and INFORMATION(I)[8,1]
91048.  then /* Just use special PROC prefix form. */
91049.  ULTIMATE_BRANCH_LABEL := '$P' || INFORMATION(I)[6,2] || 'EXIT'
91050.  else
91051.  ULTIMATE_BRANCH_LABEL := '$' || BLOCK_NUMBER(I) || 'EXIT'
91052.  fi
91053.  fi
91054.  fi
91055.  fi
91056.  fi
91057.  fi
91058.  fi
91059. */
91060.
91061.  \*/
"POP_OLD_BLOCK" Macro -- 10 July 1973

9201.  MACRO POP_OLD_BLOCK ( ; OLD_EXIT)
9202.  /* Remove the current block from the stack. Also generate END and XIT
9203.    labels if required. */
9204.
9205.    CALL TRACE_PRINTER ( ; "POP")
9206.    /* emit macro name "POP" in mnote if tracing on. */
9207.    IF CURRENT_NEST_LEVEL < NESTING_LIMIT
9208.    THEN
9209.      IF END_LABEL_REQD(CURRENT_NEST_LEVEL)
9210.      THEN
9211.        GENERATE ('$' || BLOCK_NUMBER(CURRENT_NEST_LEVEL) || 'END DS OH')
9212.      ENDIF
9213.      IF EXIT_LABEL_REQD(CURRENT_NEST_LEVEL) OR OLD_EXIT #
9214.      THEN
9215.        IF EXIT_SEVERITY = *
9216.            EXIT_SEVERITY := '0'
9217.        ENDIF
9218.      ELSE
9219.        TRACE('STRC9200 ONE OR MORE EXIT'S REFERENCED THIS POINT')
9220.      ENDIF
9221.    ELSE
9222.        GENERATE ('$' || BLOCK_NUMBER(CURRENT_NEST_LEVEL) || 'XIT DS OH')
9223.      IF OLD_EXIT #
9224.        THEN
9225.          GENERATE (OLD_EXIT || ' DS OH')
9226.      ENDIF
9227.      ENDIF
9228.  ENDIF
9229.
9230.  IF DEBUG_LIST_BLOCKS_REQD
9231.  THEN
9232.    TRACE ('* ', 'STRC9303 END OF BLOCK ' || BLOCK_NUMBER(CURRENT_NEST_LEVEL)
9233.    || '(' || BLOCK_NAME(CURRENT_NEST_LEVEL) || ') AT DEPTH ' ||
9234.    CURRENT_NEST_LEVEL)
9235.    DEBUG ('
9236.    END
9237.    CURRENT_NEST_LEVEL := CURRENT_NEST_LEVEL - 1
9238.    END
9239.    ELSE
9240.    TRACE ('* (Lemma: Execution of POP_OLD_BLOCK always decrements
9241.    CURRENT_NEST_LEVEL by exactly one.) */
9242.
9243.
9244.
9245.
9246.
9247.
9248.
9249.
9250.
9251.
9252.
9253.
9254.
9255.
9256.
"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)
93002. operand_value := ''
93003. OPERAND1_VALUE := ''
93004. OPERAND2_VALUE := ''
93005. OPERAND3_VALUE := ''
93006. INFORMATION_VALUE := ''
93007. END_LABEL_VALUE := false
93008. /* Define new block; add to stack. Save block specifications.
93009. All macro operands unspecified default to null string except
93010. END_LABEL_VALUE defaults to false. */
93011. call TRACEPRINTER( ; 'PUSH')
93012. /* Print macro name "PJ33" in mnote if tracing on. */
93013. NESTING_LIMIT := 100
93014. /* Insure axiaal depta of stack is set in variable. Note that stack
93015. depth and this variable must match, but may be changed to any
93016. value. */
93017. SCASE_NEST_LIMI := 9
93018. /* Same for general CASE stack. */
93019. ERROR_OCCURRED := false
93020. CURRENT_NEST_LEVEL := CURRENT_NEST_LEVEL + 1
93021. if CURRENT_NEST_LEVEL > NESTING_LIMIT
93022. then
93023. mnote (1, 'STAC9301 BLOCK NESTING LIMIT OF ' II NESTING_LIMIT II
93024. 'IS EXCEEDED--MACROS MUST BE MODIFIED')
93025. ERROR_OCCURRED := true
93026. else
93027. LAST_BLOCK_NUMBER := LAST_BLOCK_NUMBER + 1
93028. /* Set block number for this block. */
93029. BLOCK_NUMBER(CURRENT_NEST_LEVEL) := LAST_BLOCK_NUMBER
93030. "HELPLABEL (CURRENT_NEST_LEVEL) := current
93031. OPERAND1(CURRENT_NEST_LEVEL) := OPERAND1_VALUE
93032. OPERAND2(CURRENT_NEST_LEVEL) := OPERAND2_VALUE
93033. OPERAND3(CURRENT_NEST_LEVEL) := OPERAND3_VALUE
93034. INFORMATION(CURRENT_NEST_LEVEL) := INFORMATION_VALUE
93035. END_LABEL(CURRENT_NEST_LEVEL) := END_LABEL_VALUE
93036. BLOCK_NAME(CURRENT_NEST_LEVEL) := BLOCK_NAME_VALUE
93037. if BLOCK_NAME_VALUE = '
93038. then
93039. BLOCK_NAME(CURRENT_NEST_LEVEL) := 'BLK' II LAST_BLOCK_NUMBER
93040. fi
93041. BLOCK_LABEL_PREFIX := '' II LAST_BLOCK_NUMBER
93042. if CURRENT_NEST_LEVEL > 1 and
93043. BLOCK_TYPE(CURRENT_NEST_LEVEL-1) = 'DOCASE' and
93044. BLOCK_TYPE_VALUE = 'CASE'
93045. then
93046. mnote (3, 'STAC9302 NON-CASE BLOCK IMMEDIATELY SURROUNDED BY DOCASE INVALID')
93047. fi
93048. if DEBUG_LISTBLOCKS
93049. then
93050. mnote (*, '******************************************************)
93051. mnote (*, 'STAC9302 START OF BLOCK ' II CURRENT_NEST_LEVEL II 'AT DEPTH ' II CURRENT_NEST_LEVEL)
93052. fi
93053. fi
93054. fi

C-105
"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. FALLTHROUGH_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. BRANCH_LABEL /* Label to go on bc instruction. */
94015.
94016. call TRACE_PRINTER ( ; 'SIMPCOND')
94017. /* Print name macro "SIMPCOND" in module if tracing on. */
94018. call SIMPCOND_PROC_MASK_REL
94019. /* Extract local mask or LOCAL_REL from OPER's. If LOCAL_REL is a
94020. external value (GT, GE, EQ, LT, or LE), replace it with the
94021. proper value (H, AL, E, L, or LH). */
94022. docase OP_CODE
94023. of
94024. case 1 /* Mask or relation only. */
94025. BC_TAG := LABEL
94026. esac
94027. case 2
94028. mnote (3, 'SIMPCOND INSUFFICIENT OPERANDS FOR TEST "' || OP_CODE || '")
94029. esac
94030. case 3
94031. generate (LABEL || ' ' || OP_CODE || ' ' || OPER)
94032. esac
94033. case 4
94034. generate (LABEL || ' ' || OP_CODE || ' ' || OPER1 || ' ' || OPER2)
94035. esac
94036. case 5
94037. generate (LABEL || ' ' || OP_CODE || ' ' || OPER1 || ' ' || OPER2 || ' ' || OPER3)
94038. esac
94039. case misc
94040. mnote (3, 'SIMPCOND SUPERFLUOUS OPERANDS FOR TEST "' || OP_CODE || '")
94041. esac
94042. esacdo
94043. if LOCAL_REL == ""
94044. then
94045. if FALLTHROUGH_CONDITION /* is true: */
94046. then /* Invert mask. */
94047. if LOCAL_REL[1,1] == "W"
94048. then
94049. LOCAL_REL := LOCAL_REL[2,7]
94050. else
94051. LOCAL_REL := "W" || LOCAL_REL
94052. fi
94053. endif
94054. endif
94055. endif
94056. endif
94057. endif...
94058. esacdo...
94059. esacdo...
"SIMPLE_CONDITIONAL" Macro -- 10 July 1973

94070. define SIMPCOND GET MASK ON REL
94071. /* Extract LOCALASK or LOCAL_REL from OPER's. If LOCAL_REL is a
94072.   external value (GT, GE, EQ, LT, or LE), replace it with the
94073.   proper value (H, NL, E, L, or NH). */
94074. if OP_COUNT = 0
94075. then
94076.   error (9, 'STRING NO CONDITION SPECIFIED--"MASK=0" ASSUMED')
94077.   LOCAL_MASK := '0'
94078.   else
94079.     if SYSLST(OP_COUNT)[1,5] = 'MASK=
94080.       then
94081.         LOCAL_MASK := SYSLST(OP_COUNT)[6,8]
94082.       else
94083.         if SYSLST(OP_COUNT)[1,4] = 'REL=
94084.             then
94085.               LOCAL_REL := SYSLST(OP_COUNT)[5,8]
94086.             else
94087.               LOCAL_REL := SYSLST(OP_COUNT)
94088.             fi
94089.         fi
94090.     fi
94091. case LOCAL_REL of
94092.     case 'GT'
94093.         LOCAL_REL := 'H'
94094.     case 'GE'
94095.         LOCAL_REL := 'NL'
94096.     case 'EQ'
94097.         LOCAL_REL := 'E'
94098.     case 'LT'
94099.         LOCAL_REL := 'L'
94100.     case 'LE'
94101.         LOCAL_REL := 'NH'
94102. fi
94103. fi
94104. end
"TRACE PRINTER" Macro — 12 July 1973

MACRO TRACE_PRINTE ( ; MACRO_NAME) /* Prints macro name if tracing on. */
if DEBUG_MACRONAMES_REQD
    lnme "SZRC95000 " || MACRO_NAME
fi
mend
Macro VERIFY_END

01. verify_end ( ; reqd_type, reqd_name)
02. /* Verifies current block has name specified by reqd_name operand, if 
03. any, and that it is of type reqd_type. Various errors receive 
04. messages and either intermediate blocks are blended as a fixup 
05. or error_occurred is set. */
06. int i /* Temporary. */
VERIFY_MACRO

--
10 July 1973

96008. call TRACE PRINTER ( ; 'VERIFY')
96009. /* Print macro name "VERIFY" in anote if tracing on. */
96010. ERROR_OCCURRED := false /* Assumed. */
96011. if REQ_NAME = ''
96012. then
96013. if CURRENT_NEST_LEVEL ≤ 0
96014. then
96015. trace (6, 'STRC9607 NO BLOCKS ACTIVE—MACRO IGNORED')
96016. ERROR_OCCURRED := true
96017. else
96018. if CURRENT_NEST_LEVEL ≥ 1 and
96019. BLOCK_TYPE(CURRENT_NEST_LEVEL) ≠ REQD_TYPE
96020. then
96021. if ERROR_OCCURRED := true
96022. then
96023. error (6, 'STRC9601 ONE BLEND ASSUMED TO GET TO "REQD_TYPE"
96024. CALL BLEND ( ; )
96025. else
96026. if CURRENT_NEST_LEVEL ≥ 2 and
96027. BLOCK_TYPE(CURRENT_NEST_LEVEL - 1) = REQD_TYPE
96028. then
96029. error (6, 'STRC9601 TWO BLEND ASSUMED TO GET TO "REQD_TYPE"
96030. CALL BLEND ( ; )
96031. else
96032. error (6, 'STRC9603 CURRENT BLOCK IS NOT "REQD_TYPE"
96033. ERROR_OCCURRED := true
96034. else
96035. if I := CURRENT NESTLEVEL
96036. while I > 0 and REQD_NAME ≠ BLOCK_NAME(I)
96037. do
96038. I := I - 1
96039. od /* (Termination: I is decremented and would eventually become
96040. ≤ 0 even if other test never occurs. */
96041. if I ≤ 0
96042. then
96043. trace (6, 'STRC9604 NO ACTIVE BLOCK NAMED "REQD_NAME"
96044. ERROR_OCCURRED := true
96045. else
96046. if REQD_TYPE ≠ BLOCK_TYPE(I)
96047. then
96048. if BLOCK_NAME(CURRENT_NESTLEVEL) ≠ "REQD_NAME"
96049. ERROR_OCCURRED := true
96050. else
96051. while CURRENT_NEST_LEVEL > I
96052. do /* BLEND any intermediate blocks. */
96053. trace (6, 'STRC9605 END OF BLOCK "REQD_NAME"
96054. BLOCK_NAME(CURRENT_NESTLEVEL) \""
96055. CALL BLEND ( ; )
96056. /* (Lemma: If CURRENT_NEST_LEVEL > 0 and no BLEND operands
96057. specified, BLEND will decrement CURRENT_NEST_LEVEL
96058. by exactly one.) */
96059. od /* (Termination: After all iterations, I is fixed and
96060. CURRENT_NEST_LEVEL > I > 0. But BLEND decrements
96061. CURRENT_NEST_LEVEL. Therefore, CURRENT_NEST_LEVEL must
96062. eventually become ≤ (actually =) I.) */
96063. fi
96064. fi
96065. fi
96066. fi
96067. mexit
96068. /* (Lemma: If CURRENT_NEST_LEVEL > 0 and
96069. [REQD_NAME = ''] or \= BLOCK_TYPE(CURRENT_NEST_LEVEL)] and
96070. REQUEST = BLOCK_TYPE(CURRENT_NEST_LEVEL), then
96071. ERROR_OCCURRED will always be set false and CURRENT_NEST_LEVEL
96072. will be unmodified. Proof: If the hypothesized conditions are
96073. true, the module calls TRACE PRINTER and sets ERROR_OCCURRED to
96074. false as its only actions. TRACE PRINTER modifies no
96075. globals.) */

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

```
97001. macro CONDITIONAL_EXPRESSION_PROCESSOR (FIRST_ID; )
97002. /* Process the positional operands (the SYSLIST) as passed directly
97003. from calling macro beginning with SYSLIST(FIRST_INDEX) through
97004. SYSLIST(LAST_INDEX) generating the indicated tests to pass control
to ULTIMATE_FALLTHRU_LABEL when the ULTIMATE_FALLTHRU_CONDITION is
97005. found to match the logical value tested and branches to the
97006. ULTIMATE_BRANCH_LABEL otherwise; the UNIQUE_LABEL_ID is used to
97007. insure unique labels; if a branch is made to the fall-through label,
97008. FALLTHRU_LABEL_USED is set, else unaltered. Only the "SYSLIST"
97009. operand is passed as an actual macro operand. The other variables
97110. mentioned are globals. */
97111. int
97112. CND_COUNT, /* Counts the simple conditionals within the
97113. conditional expression. */
97114. DEPTH, /* Angle bracket nesting depth of simple conditional being
97115. processed. */
97116. INDEX, /* Operand index within the SYSLIST of the simple conditional
97117. being processed. */
97118. OP_COUNT, /* Number of suboperands in the simple conditional being
97119. processed. */
97120. NEXT_INDEX, /* Index of the AND or OR which follows the current
97121. simple conditional. */
97122. NEXT_DEPTH, /* Angle bracket depth of NEXT_INDEX. */
97123. LA_DEPTH, /* Angle bracket depth during operand look-ahead. */
97124. I /* Operand index of operand being examined during
97125. look-ahead. */
97126. bit
97127. AND_OR_OUTSTANDING, /* Indicates whether an AND or OR follows
97128. the current simple conditional. */
97129. LOCAL_FALLTHRU_CONDITION, /* Logical value of the simple conditional
97130. being processed which is to lead to control falling through the
97131. test. */
97132. LOCAL_LABEL_REQD(20), /* Indicates whether the corresponding
97133. simple conditionals require a label due to branching logic. */
97134. char
97135. LABEL, /* Outstanding label waiting to be generated. */
97136. LOCAL_BRANCH_LABEL /* Label for branch target if current simple
97137. conditional has the opposite truth value from that stored in
97138. LOCAL_FALLTHRU_CONDITION. */
```
"CONDITIONAL_EXPRESSION_PROCESSOR" Macro -- 9 July 1973

97041. call TRACE_PRINTF ( ; "CEF")
97042. /* Print macro name "CEF" if tracing on. */
97043. LABEL := FIRST_INDEX
97044. /* Set calling label as outstanding. */
97045. COND_COUNT := 0
97046. DEPTH := 0
97047. INDEX := FIRST_INDEX
97048. while INDEX <= LAST_INDEX
dg
97049. call CEP_FIND_NEXT_CONDITION
97050. /* Step INDEX up to next simple conditional incrementing DEPTH for any
97051. "on" found and setting ERROR_OCCURRED on any syntax error. Increments
97052. COND_COUNT for the condition found. */
97053. if ERROR_OCCURRED
97054. then
97055. REEX
97056. fi
97057. AND_OR_OUTSTANDING := false
97058. call CEP_LOOKAHEAD
97059. /* Find LOCAL_BRANCH_LABEL or generate one. If LOCAL_BRANCH_LABEL
97060. is the ULTIMATE_FALLTHRU_LABEL, set FALLTHRU_LABEL_USED. Set
97061. LOCAL_FALLTHRU_CONDITION. Also set NEXT_INDEX and NEXT_DEPTH with
97062. the INDEX/DEPTH of the AND or OR following this conditional.
97063. (The value of NEXT_INDEX returned is greater than the value
97064. of INDEX entered.) */
97065. if ERROR_OCCURRED
97066. then
97067. REEX
97068. fi
97069. if LOCAL_LABZL_REQ(COND_COUNT)
97070. then
97071. LABEL := BLOCK_LABEL_PREPIL || UNIQUE_LABEL_ID || COND_COUNT
97072. fi
97073. GO_COUNT := N'SYSLIST(INDEX)
97074. call SIMPLE_CONDITIONAL (LABEL);
97075. SYSLIST(INDEX,1)
97076. SYSLIST(INDEX,2)
97077. SYSLIST(INDEX,3)
97078. SYSLIST(INDEX,4)
97079. >>LAST_INDEX,2)
97080. LOCAL_BRANCH_LABEL,
97081. LOCAL_FALLTHRU_CONDITION,
97082. GO_COUNT)
97083. LABEL := 'T'
97084. INDEX := NEXT_INDEX
97085. DEPTH := NEXT_DEPTH
97086. go /* (Termination: INDEX is incremented (by CEP_LOOKAHEAD's
97087. return of NEXT_INDEX) and LAST_INDEX is fixed in loop; INDEX must
97088. eventually exceed LAST_INDEX.) */
97089. if AND_OR_OUTSTANDING
97090. then
97091. note (0, "STRLG9701 INSUFFICIENT OPERANDS")
97092. fi
97093. if DEPTH > 0
97094. then
97095. note (0, "STRLG9702 INSUFFICIENT BRACKETS")
97096. fi
97097. end
57100. PROC CEP_FIND_NEXT_CONDITION
57101. /* Step up to next simple conditional, incrementing DEPTH for any "<" or "=" found. Start with SYSLIST(INDEX) and advance INDEX till found or LAST_INDEX. Set ERROR_OCCURRED on any syntax error, giving message. Increment COND_COUNT by 1. */
57102.
57103. ERROR_OCCURRED := false
57104. while INDEX ≤ LAST_INDEX and (SYSLIST(INDEX) = '<' or = '=')
57105. do
57106.    DEPTH := DEPTH + 1
57107.    INDEX := INDEX + 1
57108.    od /* Termination: INDEX is incremented, LAST_INDEX is fixed in loop; INDEX would eventually exceed LAST_INDEX even if other tests never occur. */
57109. if SYSLIST(INDEX)[1,1] = '('
57110.   then
57111.     mnote (8, 'STRC706 SYNTAX ERROR--LOOKING FOR SIMPLE "<" OR "="')
57112.     ERROR_OCCURRED := true
57113.   fi
57114. COND_COUNT := COND_COUNT + 1
57115. od
CONDITIONAL_EXPRESSION_PROCESSOR  #macro  9 July 1973

PROC CEP_LOOKAHEAD

/* Search operands beyond current simple conditionals. If AND/OR found,
label found or generated for that spot into LOCAL_BRANCH_LABEL. If
same as ULTIMATE_FALLTHRU_LABEL, set FALLTHRU_LABEL_USED. Decide
whether this test is to fallthru on true or false and set
LOCAL_FALLTHRU_CONDITION. If syntax error found during lookahead,
give message and set ERROR_OCCURRED. Also set NEXT_INDEX and
NEXT_DEPTH with the index/depth of the AND/OR. */

ERROR_OCCURRED := false
LA_DEPTH := DEPTH
I := INDEX + 1
while I <= LAST_INDEX and (SYSLIST(I) = '!' or = '/')
do
  if LA_DEPTH > 0 then
    LA_DEPTH := LA_DEPTH - 1
  else
    mnote (8, 'STEC9705 SUPERFLUOUS BRACKET IGNORED')
  fi
/* (Termination: I is incremented, LAST_INDEX is fixed in
loop; INDEX would eventually exceed LAST_INDEX even if other
tests never occur.) */
NEXT_INDEX := I + 1
/* (Lemma: NEXT_INDEX > I > INDEX.) */
NEXT_DEPTH := LA_DEPTH
if I > LAST_INDEX then
  LOCAL_BRANCH_LABEL := ULTIMATE_BRANCH_LABEL
  LOCAL_FALLTHRU_CONDITION := ULTIMATE_FALLTHRU_CONDITION
else
  if SYSLIST(I) = 'AND' or = 'OR'
    then
      /* Search ahead for branch target. Set AND_OR_OUTSTANDING, LOCAL_
FALLTHRU_CONDITION, and LOCAL_SEARCH_POINT. */
      CALL CEP_SCAN_FBRANCH
      else
        mnote (8, 'STEC9703 SYNTAX ERROR--LOOKING FOR "AND" OR "OR", \nFOUND "' | SYSLIST(I) | '"
      /* ERROR_OCCURRED := true */
    fi
  fi
/* (Lemma: The value of NEXT_INDEX returned is greater than the
value of INDEX entered.) */

C-114
BEHEMOTH_SCANFOR_BRANCH

/ * SYSLIST(I) is either "AND" or "OR". Set AND/OR_OUTSTANDING. Set
LOCAL_FALLTHRU_CONDITION according to current operation (AND or OR).

Continue scan over operands until simple conditional is found which
is target for LOCAL_BRANCHLABEL of current simple conditional; then
generate LABEL and set LOCAL_LABEL_BEG for target test. If no target
test found set LOCAL_BRANCHLABEL to either (a) the ULTIMATE_FALLTHRU-
LABEL (also setting FALLTHRU_LABEL_USED) or to (b) the ULTIMATE_
BRANCH_LABEL, depending on operation. */

CHAR LOCKFOB / * Operation ("AND" or "OR") opposite of SYSLIST(1). */

INT MAX Depths, / * The maximum depth at which the LOOKFOR'ed
operation is a possible branch target. */

TARGET / * Simple conditional number which is the target for
the branch. */
AND_GT_OUTSTANDING := true
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
/* Advance I to force skip of unnecessary search. */
fi

LOCAL_FALLTHRU := 'AND'
LOCKFCB := 'OR'
MAX_DEPTH := LA_DEPTH

else /* Operation is CR. */
LOCAL_FALLTHRU := false
fi

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

LOCKFOR := 'AND'
MAXDEPTH := LA_DEPTH - 1

fi

I := I + 1
TARGET := CONDCOUNT + 1
while I < LAST_INDEX and
(STYLIST(I) != LOOKFOR OR LA_DEPTH > MAX_DEPTH)
do
if SYSLIST(I) = '<' OR = '/*'
then
LA_DEPTH := LA_DEPTH + 1
else
if SYSLIST(I) = '>' OR = '/'
then
if LA_DEPTH > 0
then
LA_DEPTH := LA_DEPTH - 1
else
if SYSLIST(I)[1,1] = '('
then
TARGET := TARGET + 1
else
if SYSLIST(I) != 'AND' and != 'OR'
then
/* Block 8: 'STBC9700 SYNTAX ERROR—' II SYSLIST(I) II */
ERROR_OCCURRED := true
I := LAST_INDEX + 1
/* Force break out of loop. */
fi
fi
fi
else
if SYSLIST(I) != 'OR' and != 'AND'
then
/* Note that we are relying on the automatic initialization of the */
/* local_LABEL_REQD array to false at start of every invocation of */
/* CONDITIONAL_EXPRESSION_PROCESSOR. */
fi

/* Termination: I is either incremented or set to LAST_INDEX + 1; */
/* if other tests never occurred. */
if I > LAST_INDEX
then
(Locking = 'OR' /* Operand was AND. */ and
 ~ ULTIMATE_FALLTHRU CONDITION) or
(Locking = 'AND' /* Operand was OR. */ and
ULTIMATE_FALLTHRU CONDITION /* is true */)
then
LOCAL_BRANCH_LABEL := ULTIMATE_FALLTHRU_LABEL
FALLTHRU_LABEL USED := true
else
LOCAL_BRANCH_LABEL := NULL
fi

else
LOCAL_BRANCH_LABEL := BLOCK_LABEL_PREFIX ++ UNIQUE_LABEL_ID ++ TARGET
LOCAL_LABEL_REQD(TARGET) := true
/* Note that we are relying on the automatic initialization of the */
/* local_LABEL_REQD array to false at start of every invocation of */
/* CONDITIONAL_EXPRESSION_PROCESSOR. */
fi

end
**TERMINATE_DO_LOOP** Macro  —  26 June 1973

98001. macro TERMINATE_DO_LOOP ( ; )
98002. /* Called by DO macro to terminate the current DO block loop by
98003. generating the necessary loop-terminating branches. If control
98004. can fall out of the bottom of the code at loop termination, set
98005. TDL_FALLTHRU OCCURS to true; else set false. */
98006. bit LB_LABEL_REQ; /* Indicates whether label required on
98007. looping branch instruction. */
98008. char S8010. X; /* Temporary string. */
98009. bit LB_OPCODE, OPER1, OPER2; /* One character codes for
98010. flow points which are targets for flow points 8, 10, and 11. */
98011. if LB_LABEL_REQ then
98012. LABEL := BLOCK_LABEL_PREFIX || 'LPB');
98013. fi
98014. /
98015. Generate all looping instructions. */
98016. docase BRANCHES of
98017. case ('w', 'U')
98018. generate (LABEL || 'B ' || BLOCK_LABEL_PREFIX || 'BEG')
98019. esac
98020. case ('L')
98021. /* Nothing to generate; fall through to looping branch. */
98022. esac
98023. esacod
98024. TDL_FALLTHRU_OCCURS := false; /* Assume we will always branch. */
98025. if LB_OPCODE = 'BCT' then
98026. /* It must be 'W' or 'U'; looping branch fall-through is not to end
98027. of loop; generate branch to proper alternative conditional test. */
98028. generate (LABEL || 'B ' || BLOCK_LABEL_PREFIX || 'BEG')
98029. TDL_FALLTHRU_OCCURS := false; /* Branch defeats fall through. */
98030. fi
98031. fi
98032. mend;
98033. /* (Lemma: TERMINATE_DO_LOOP does not modify
98034. CURRENT_NEST_LEVEL.})

98017. call TRACE_PRINTER( ; 'TDL')
98018. /* Extract the following stored by DO macro: */
98019. X := INFORMATION(CURRENT_NEST_LEVEL)
98020. BRANCH8 := X[1,1]
98021. BRANCH10 := X[2,1]
98022. BRANCH11 := X[3,1]
98023. LB_OPCODE := OPERAND3(CURRENT_NEST_LEVEL)
98024. BLOCK_LABEL_PREFIX := '$' || BLOCK_NUMBER(CURRENT_NEST_LEVEL)
98025. if LB_OPCODE = 'BCT' then
98026. generate (LABEL || 'B ' || BLOCK_LABEL_PREFIX || 'BEG')
98027. fi
98028. if LB_OPCODE = '' or 'U'
98029. then if BRANCH10 = 'B'
98030. X := 'BEG'
98031. else if BRANCH10 = 'W' or 'U'
98032. X := BRANCH10 || '1'
98033. fi
98034. fi
98035. if LB_OPCODE = 'BCT'
98036. generate (LABEL || 'B ' || BLOCK_LABEL_PREFIX || 'BEG')
98037. fi
98038. if LB_OPCODE ='' or 'U'
98039. then if BRANCH11 = 'W'
98040. /* It must be 'W' or 'U'; looping branch fall-through is not to end
98041. of loop; generate branch to proper alternative conditional test. */
98042. generate (LABEL || 'B ' || BLOCK_LABEL_PREFIX || 'BEG')
98043. TDL_FALLTHRU_OCCURS := false
98044. /* Branch defeats fall through. */
98045. fi
98046. fi
98047. /
98048. /
98049. /
98050. /
98051. /
98052. /
98053. /
98054. /
98055. /
98056. /
98057. /
98058. /
98059. /
98060. /
98061. /
98062. /
98063. /
98064. /
98065. /
98066. /
98067. /
98068. /
98069. /
98070. /
98071. /
98072. /
98073. /
98074. /* (Lemma: TERMINATE_DO_LOOP does not modify
98075. CURRENT_NEST_LEVEL.})
"XHEX" Macro -- 19 December 1973

59001. MACRO XHEX ( : )
59002.   /* Converts the integer in HEX_IN to a two-character hex string is
59003.   HEX_OUT */
59004.   AIX I, J /* Temporaries */
59005.   call TRACE.PRINTER ( : 'XHEX')
59006.   /* Print macro name "XHEX" in case if tracing on. */
59007.   if HEX(I) # '0'
59008.       then /* Hex array must be initialized */
59010.       HEX(1) := "0" HEX(2) := "1" HEX(3) := "2" HEX(4) := "3"
59011.       HEX(5) := "4" HEX(6) := "5" HEX(7) := "6" HEX(8) := "7"
59012.       HEX(9) := "8" HEX(10) := "9" HEX(11) := "A" HEX(12) := "B"
59013.       HEX(13) := "C" HEX(14) := "D" HEX(15) := "E" HEX(16) := "F"
59014.     IA
59015.     I := HEX_IN/16
59016.     J := HEX_IN % 16 + 1
59017.     HEX_OUT := HEX(I+1) || HEX(J)
59018. endm
# CROSS-REFERENCE LISTING

<table>
<thead>
<tr>
<th>Label</th>
<th>Offset</th>
<th>Left</th>
<th>Right</th>
</tr>
</thead>
<tbody>
<tr>
<td>LO_LABEL_BLOCK</td>
<td>21114</td>
<td>.21754.</td>
<td>83264</td>
</tr>
<tr>
<td>DO_LOOPING_BRANCH_AND_FIRST_OP</td>
<td>21147</td>
<td>.21302.</td>
<td></td>
</tr>
<tr>
<td>DO_LOOPING_BRANCH_PROCESS</td>
<td>21369</td>
<td>.21374.</td>
<td></td>
</tr>
<tr>
<td>DO_SCAN_OPERANDS</td>
<td>21090</td>
<td>.21127.</td>
<td></td>
</tr>
<tr>
<td>DO_SEQ_FORMAT</td>
<td>21153</td>
<td>.21436.</td>
<td></td>
</tr>
<tr>
<td>DO_TRACE_COUNTERS</td>
<td>21119</td>
<td>.21840.</td>
<td></td>
</tr>
<tr>
<td>DO_UNTIL_POSTPROCESS</td>
<td>21621</td>
<td>.21738.</td>
<td></td>
</tr>
<tr>
<td>DO_UNTIL_PREPROCESS</td>
<td>21594</td>
<td>.21638.</td>
<td></td>
</tr>
<tr>
<td>DO_WHILE_PREPROCESS</td>
<td>21556</td>
<td>.21675.</td>
<td></td>
</tr>
<tr>
<td>DOCASE</td>
<td>a21001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DOCASE_SEQ_STUFF</td>
<td>31071</td>
<td>.31226.</td>
<td></td>
</tr>
<tr>
<td>DOCASE_SEQ_END</td>
<td>31086</td>
<td>.31287.</td>
<td></td>
</tr>
<tr>
<td>DOCASE_INDEX_TO_REG</td>
<td>31059</td>
<td>.31149.</td>
<td></td>
</tr>
<tr>
<td>DOCASE_INFO_SAVE</td>
<td>31073</td>
<td>.31261.</td>
<td></td>
</tr>
<tr>
<td>DYNAMIC_SAVEDAREA</td>
<td>a30321</td>
<td>83090</td>
<td>83264</td>
</tr>
<tr>
<td>ELSE</td>
<td>*11001</td>
<td>11024</td>
<td>11123</td>
</tr>
<tr>
<td>ELSE_SEQ_BLOCK</td>
<td>u13008</td>
<td>13042</td>
<td>13044</td>
</tr>
<tr>
<td>END_LABEL_VALUE</td>
<td>11025</td>
<td>11030</td>
<td>11042</td>
</tr>
<tr>
<td>EQUAL_TEST_OUTSTANDING</td>
<td>a33022</td>
<td>33194</td>
<td>33205</td>
</tr>
<tr>
<td>ERROR_OCCURRED</td>
<td>n104</td>
<td>11130</td>
<td>11149</td>
</tr>
<tr>
<td>EXIT</td>
<td>*11001</td>
<td>11024</td>
<td>11123</td>
</tr>
<tr>
<td>EXIT_SEQ_LABEL</td>
<td>a11016</td>
<td>11138</td>
<td>11211</td>
</tr>
<tr>
<td>EXIT_SEQ_LABEL_REQ</td>
<td>n206</td>
<td>11046</td>
<td>11058</td>
</tr>
<tr>
<td>EXIT_SEVERITY</td>
<td>a160</td>
<td>10113</td>
<td>10149</td>
</tr>
<tr>
<td>EXIT_SERVERS</td>
<td>*11001</td>
<td>11024</td>
<td>11123</td>
</tr>
<tr>
<td>FALLTHRU_CONDITION</td>
<td>09400</td>
<td>09409</td>
<td>09530</td>
</tr>
<tr>
<td>FALLTHRU_LABEL_USED</td>
<td>a256</td>
<td>.11153</td>
<td>.11155</td>
</tr>
<tr>
<td>FCO_1</td>
<td>a3502</td>
<td>.97247</td>
<td>.97247</td>
</tr>
<tr>
<td>FCO</td>
<td>11072</td>
<td>.11561</td>
<td>.1158</td>
</tr>
<tr>
<td>FCO_LABEL_REQ</td>
<td>a23014</td>
<td>.20353</td>
<td>.20367</td>
</tr>
<tr>
<td>FCO_INDEX</td>
<td>a25072</td>
<td>20702</td>
<td>20703</td>
</tr>
<tr>
<td>FCO_INDEX2</td>
<td>a81027</td>
<td>81424</td>
<td>81465</td>
</tr>
<tr>
<td>FCO_INDEX3</td>
<td>*a81500</td>
<td>81504</td>
<td>81781</td>
</tr>
<tr>
<td>FCO_INDEX4</td>
<td>a25069</td>
<td>.21634</td>
<td>.21651</td>
</tr>
<tr>
<td>FAST_INDEX</td>
<td>a247</td>
<td>.11173</td>
<td>.12656</td>
</tr>
<tr>
<td>FAST_INDEX1</td>
<td>a81010</td>
<td>.83183</td>
<td>.83208</td>
</tr>
<tr>
<td>FAST_INDEX2</td>
<td>a38033</td>
<td>.83152</td>
<td>.83168</td>
</tr>
<tr>
<td>FAST_INDEX3</td>
<td>a83026</td>
<td>.83195</td>
<td>.83314</td>
</tr>
<tr>
<td>FAST_INDEX4</td>
<td>a83013</td>
<td>.83178</td>
<td>.83180</td>
</tr>
<tr>
<td>FAST_INDEX5</td>
<td>a81012</td>
<td>.81462</td>
<td>.81467</td>
</tr>
<tr>
<td>FAST_INDEX6</td>
<td>a83024</td>
<td>.83165</td>
<td>.83176</td>
</tr>
<tr>
<td>FWD_1</td>
<td>a81836</td>
<td>.81851</td>
<td>.81855</td>
</tr>
<tr>
<td>FWD_2</td>
<td>80466</td>
<td>.82377</td>
<td>.82479</td>
</tr>
<tr>
<td>FWD_3</td>
<td>.8243</td>
<td>.8243</td>
<td>.8243</td>
</tr>
<tr>
<td>GCA_INDEX</td>
<td>a233</td>
<td>.31201</td>
<td>.31201</td>
</tr>
<tr>
<td>GCA_INDEX1</td>
<td>31221</td>
<td>33089</td>
<td>33129</td>
</tr>
<tr>
<td>GCA_INDEX2</td>
<td>33244</td>
<td>33042</td>
<td>33075</td>
</tr>
<tr>
<td>GCA_INDEX3</td>
<td>a236</td>
<td>31202</td>
<td>31222</td>
</tr>
<tr>
<td>GCA_INDEX4</td>
<td>a81173</td>
<td>.81182</td>
<td>.81199</td>
</tr>
<tr>
<td>GCA_INDEX5</td>
<td>a83032</td>
<td>.83156</td>
<td>.83161</td>
</tr>
<tr>
<td>GCA_INDEX6</td>
<td>a3340</td>
<td>23442</td>
<td>.83444</td>
</tr>
<tr>
<td>GCA_INDEX7</td>
<td>33455</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GCA_INDEX8</td>
<td>a162</td>
<td>99006</td>
<td>99010</td>
</tr>
<tr>
<td>GCA_INDEX9</td>
<td>*99013</td>
<td>99013</td>
<td>99013</td>
</tr>
<tr>
<td>GCA_INDEX10</td>
<td>a141</td>
<td>.33581</td>
<td>.81155</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Modified</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>99016</td>
</tr>
</tbody>
</table>

C-121
### CROSS-REFERENCE LISTING

<table>
<thead>
<tr>
<th>NAME</th>
<th>MODIFIED</th>
<th>ORIGINAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>N eclectic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T eclectic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N eclectic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T eclectic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N eclectic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T eclectic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N eclectic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T eclectic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N eclectic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T eclectic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N eclectic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T eclectic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N eclectic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T eclectic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N eclectic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T eclectic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N eclectic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T eclectic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N eclectic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T eclectic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N eclectic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T eclectic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N eclectic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T eclectic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N eclectic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T eclectic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N eclectic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T eclectic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N eclectic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T eclectic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N eclectic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T eclectic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N eclectic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T eclectic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N eclectic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T eclectic</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
CROSS-REFERENCE LISTING

97253.

IDL_FAILTHRU_OCCURS
n 146. 25055. **98045. **98046. **98063.

IDL_GEMMED

TERMINEATE_GO_LOOP
23049. 25054. 27033. **98001.

THIS_CONDITIONAL_LOOP

TRACE_Printer

TRACE_VECTOR_GEMMED
n 153. 81873. **81894.

ULTIMATE_BRANCH_LABEL

ULTIMATE_FALING_CONDITION

ULTIMATE_FALLTHRU_LABEL

UNEXPECTED_OPERANDS_FOUND

UNIQUE_LABEL_ID

UNTILL_COND_TEST

UNTILL_END_INDEX

UNTILL_INDEX

UNTILL_PRESENT

USER_NAME

USING13

VALID_EXIT

VERIFY_END

WHILE_COND_TEST

WHILE_END_INDEX

WHILE_INDEX

WHILE_PRESENT

WORK

WORKREG

WORKREG_USED

X

XHEX
33562. **81156. 099001.
The messages generated by the STRCMACS are described below. Each message has an identifying number.

<table>
<thead>
<tr>
<th>Severity</th>
<th>Message-Number</th>
<th>Message-Text</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>STRC1101</td>
<td>EXIT= IGNORED WITH ASYNCH</td>
<td>Both the EXIT= and ASYNCH operands were specified, but are mutually exclusive; the EXIT= operand has been ignored.</td>
</tr>
<tr>
<td>8</td>
<td>STRC1102</td>
<td>REL= OR MASK= NOT IN PARENTHESES—IGNORED</td>
<td>The REL= and MASK= operands must be part of a simple conditional and thus must be inside parentheses. The keyword has been ignored.</td>
</tr>
<tr>
<td>8</td>
<td>STRC1103</td>
<td>EXIT= IGNORED WITH ELSE=</td>
<td>The EXIT= and ELSE= operands were both specified but are mutually exclusive. The EXIT= operand has been ignored.</td>
</tr>
<tr>
<td>8</td>
<td>STRC1301</td>
<td>ELSE=namel SPECIFIED ON IF BLOCK name2</td>
<td>The current IF block (whose name is name2) included ELSE=namel as an operand, but a different (or no) name appears in the label field of this ELSE macro. The discrepancy is ignored.</td>
</tr>
<tr>
<td>8</td>
<td>STRC1302</td>
<td>ELSE HAS ALREADY BEEN GENERATED FOR CURRENT IF</td>
<td>An ELSE macro has already occurred in the current IF block. The macro is ignored.</td>
</tr>
<tr>
<td>Severity</td>
<td>Message-Number</td>
<td>Message-Text</td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td>----------------</td>
<td>------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>STRC1501</td>
<td>ELSE BLOCK elsename NOT FOUND</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>The operand ELSE=elsename was coded on the IF macro, but the FI has occurred before the ELSE occurred.</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>STRC2101</td>
<td>OPERANDS AFTER SECOND &quot;WHILE&quot; IGNORED</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>The keyword &quot;WHILE&quot; appears more than once in the DO's operands.</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>STRC2102</td>
<td>OPERANDS AFTER SECOND &quot;UNTIL&quot; IGNORED</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>The keyword &quot;UNTIL&quot; appears more than once in the DO's operands.</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>STRC2103</td>
<td>WARNING—&quot;WHILE,(BCT,...&quot; WILL LOOP ONE LESS TIME THAN VALUE IN REGISTER</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>The looping branch BCT was coded in the WHILE looping group. Since the BCT is executed before the loop, the loop will occur one time fewer than the initial value in the register.</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>STRC2104</td>
<td>WARNING—LOOPING BRANCH MAY NOT BE EXECUTED ON EVERY ITERATION</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>A looping branch is present in the WHILE looping group and the UNTIL looping group is also present. The two looping groups are connected by &quot;OR&quot;. If loop execution is to be continued on the basis of the UNTIL group, the WHILE group will not be executed. Hence the indexing register of the looping branch will not be bumped in such cases.</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>STRC2105</td>
<td>TWO LOOPING BRANCHES INVALID IN &quot;DO&quot;—&quot;WHILE&quot; IGNORED</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Both the WHILE and UNTIL looping groups contain looping branches (BCTs, BXHs, or BXLE,); the WHILE looping group has been ignored.</td>
<td></td>
</tr>
<tr>
<td>Severity</td>
<td>Message-Number</td>
<td>Message-Text</td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td>----------------</td>
<td>--------------</td>
<td></td>
</tr>
</tbody>
</table>
| 8        | STRC2106       | INVALID NUMBER OF OPERANDS FOR \textit{opcode}  
The looping branch \textit{opcode} has the wrong number of operands; BCT should have one, BXLE or BXH should have two. |
| 8        | STRC2107       | \textit{xxx} INVALID AFTER LOOPING BRANCH—"AND" INSERTED  
The operand following the looping branch must be "AND" or "OR"; \textit{xxx} was found. |
| 8        | STRC2108       | FIRST OPERAND MUST BE "WHILE", "UNTIL", "FOREVER", OR OMITTED  
The first operand of the DO macro is invalid. Either WHILE or UNTIL has been inserted, depending on the remaining operands. |
| 8        | STRC2109       | WHILE TEST IS VOID—IGNORED  
No looping group follows the keyword "WHILE"; the keyword has been discarded. |
| 8        | STRC2110       | LOGIC OPERATOR BETWEEN "WHILE" AND "UNTIL" OMITTED—"AND" ASSUMED  
No logic operator occurs between the WHILE and UNTIL looping groups. An "AND" has been inserted. |
| 8        | STRC2111       | UNTIL TEST IS VOID—IGNORED  
No looping group follows the keyword "UNTIL"; the keyword has been discarded. |
| 8        | STRC2112       | PARENTHESES OMITTED AROUND \textit{opcode}  
The looping branch \textit{opcode} was not specified as a sublist. |
<table>
<thead>
<tr>
<th>Severity</th>
<th>Message-Number</th>
<th>Message-Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>STRC2113</td>
<td>REL= or MASK= not in parentheses—ignored&lt;br&gt;The REL= and MASK= operands must be a part of a simple conditional and thus must be inside parentheses. The keyword has been ignored.</td>
</tr>
<tr>
<td>8</td>
<td>STRC2114</td>
<td>Superfluous looping group ignored&lt;br&gt;Both WHILE and UNTIL are present, but other operands precede both. Such operands have been ignored.</td>
</tr>
<tr>
<td>8</td>
<td>STRC2301</td>
<td>More than one &quot;ATEND&quot; in block&lt;br&gt;More than one ATEND macro has been found for the same DO. Only the first is processed.</td>
</tr>
<tr>
<td>8</td>
<td>STRC2501</td>
<td>More than one &quot;ONEXIT&quot; in block&lt;br&gt;More than one ONEXIT macro has been found for the same DO. Only the first is processed.</td>
</tr>
<tr>
<td>8</td>
<td>STRC2502</td>
<td>No exit for this &quot;DO&quot;&lt;br&gt;No EXIT has occurred specifying the DO block for which an ONEXIT is being generated. The segment is dead code.</td>
</tr>
<tr>
<td>4</td>
<td>STRC3101</td>
<td>Warning—xxx assumed as index: use &quot;DOCASE,xxx&quot; for range spec&lt;br&gt;xxx is either &quot;IFANY&quot; or &quot;ONLY&quot; and appears as the first operand. As such, it is assumed to be the address of the DOCASE index. If the range specification is intended, xxx must be the second or third operand.</td>
</tr>
<tr>
<td>8</td>
<td>STRC3102</td>
<td>xxx invalid second operand—ignored&lt;br&gt;The second operand of DOCASE may only be &quot;SPARSE&quot;, &quot;SIMPLE&quot;, &quot;IFANY&quot;, &quot;ONLY&quot;, or omitted. xxx was found.</td>
</tr>
<tr>
<td>Severity</td>
<td>Message-Number</td>
<td>Message-Text</td>
</tr>
<tr>
<td>---------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>8</td>
<td>STRC3103</td>
<td>INVALID THIRD OPERAND—IGNORED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The third operand of DOCASE may only be &quot;IFANY&quot;, &quot;ONLY&quot;, or omitted. xxx was found.</td>
</tr>
<tr>
<td>12</td>
<td>STRC3104</td>
<td>GENERAL/SPARSE/CHARCOMP DOCASE NESTING LEVEL nestlev EXCEEDS MAXIMUM OF maxlev—MACROS MUST BE MODIFIED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The number of nesting levels of DOCASE macros (other than SIMPLE or conditional test type DOCASEs) is nestlev; but the internal stack limits such nesting to maxlev. Either the program or the macros must be modified.</td>
</tr>
<tr>
<td>8</td>
<td>STRC3301</td>
<td>&quot;CASE&quot; NOT IMMEDIATE DAUGHTER OF &quot;DOCASE&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The CASE macro is not immediately surrounded by a DOCASE macro. If one or two BLENDS are required, they will be inserted and message 3302 or 3303 will be issued; otherwise, message 3304 will follow.</td>
</tr>
<tr>
<td>8</td>
<td>STRC3302</td>
<td>ASSUMING &quot;BLEND&quot; OMITTED—INSERTED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Preceded by message 3301. Since the second containing block is a DOCASE, one BLEND is inserted to get to it.</td>
</tr>
<tr>
<td>8</td>
<td>STRC3303</td>
<td>ASSUMING TWO &quot;BLENDS&quot; OMITTED—INSERTED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Preceded by message 3301. Since the third containing block is a DOCASE, two BLENDS are inserted to get to it.</td>
</tr>
<tr>
<td>8</td>
<td>STRC3304</td>
<td>&quot;CASE&quot; TREATED AS &quot;BLOCK&quot; MACRO</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Preceded by message 3301. No fix-up was possible. The CASE is converted to a BLOCK.</td>
</tr>
<tr>
<td>Severity</td>
<td>Message-Number</td>
<td>Message-Text</td>
</tr>
<tr>
<td>----------</td>
<td>----------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>8</td>
<td>STRC3305</td>
<td>xxx INVALID—yyy ASSUMED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>xxx appears as the operand of a CASE macro, but the format is invalid.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Usually this occurs when a range is coded whose second value is less than</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the first (such as &quot;(15,10)&quot;). The operand yyy replaces xxx.</td>
</tr>
<tr>
<td>4</td>
<td>STRC3306</td>
<td>EARLIER UNEXPECTED OPERAND IMPLIES THIS TO BE CASE xxx</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CASE macro has no operands but earlier CASEs for the same DOCASE contained</td>
</tr>
<tr>
<td></td>
<td></td>
<td>operands other than their ordinal position numbers. The operand xxx has</td>
</tr>
<tr>
<td></td>
<td></td>
<td>been assumed.</td>
</tr>
<tr>
<td>8</td>
<td>STRC3307</td>
<td>OPERAND INVALID VALUE ON SIMPLE CASE xxx</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The ordinal position number of this CASE is xxx, but SIMPLE was coded on</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the DOCASE and an operand other than xxx on the CASE. The operand is</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ignored.</td>
</tr>
<tr>
<td>8</td>
<td>STRC3308</td>
<td>&quot;DOCASE...,ONLY&quot; INVALID WITH MISC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A CASE MISC has been found in a DOCASE with the ONLY range option. Since</td>
</tr>
<tr>
<td></td>
<td></td>
<td>these are mutually exclusive, the ONLY has been ignored.</td>
</tr>
<tr>
<td>8</td>
<td>STRC3309</td>
<td>OPERAND MUST BE SELF-DEFINING TERM OR OMITTED ON SIMPLE CASE xxx</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The ordinal position number of this CASE is xxx and SIMPLE was coded on</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the DOCASE, but an operand has been specified which is not a self-defining</td>
</tr>
<tr>
<td></td>
<td></td>
<td>term. It has been ignored.</td>
</tr>
<tr>
<td>8</td>
<td>STRC3310</td>
<td>REL= OR MASK= NOT IN PARENTHESES—IGNORED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The REL= and MASK= operands must be a part of a simple conditional and thus</td>
</tr>
<tr>
<td></td>
<td></td>
<td>must be inside parentheses. The keyword has been ignored.</td>
</tr>
<tr>
<td>Severity</td>
<td>Message-Number</td>
<td>Message-Text</td>
</tr>
<tr>
<td>----------</td>
<td>----------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>8</td>
<td>STRC3311</td>
<td>MULTIPLE MISC CASES IN THIS DOCASE—THIS BLOCK IS DEAD CODE</td>
</tr>
<tr>
<td></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'h'</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'h' 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>

D-7
<table>
<thead>
<tr>
<th>Severity</th>
<th>Message-Number</th>
<th>Message-Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>STRC8101</td>
<td>LINKAGE=xxx INVALID—&quot;OS&quot; ASSUMED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The first suboperand of the LINKAGE= keyword is</td>
</tr>
<tr>
<td></td>
<td></td>
<td>invalid; &quot;OS&quot; has been substituted.</td>
</tr>
<tr>
<td>8</td>
<td>STRC8102</td>
<td>SECOND LINKAGE OPERAND IGNORED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The second suboperand of the LINKAGE= keyword</td>
</tr>
<tr>
<td></td>
<td></td>
<td>must be either &quot;CSECT&quot; or omitted. It is invalid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>and has been ignored.</td>
</tr>
<tr>
<td>4</td>
<td>STRC8103</td>
<td>WARNING—SAVETRACE REQUIRES &quot;FINAL&quot; MACRO</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The SAVETRACE debug option has been specified</td>
</tr>
<tr>
<td></td>
<td></td>
<td>on the first PROC; warning is printed to indicate</td>
</tr>
<tr>
<td></td>
<td></td>
<td>need for FINAL macro.</td>
</tr>
<tr>
<td>8</td>
<td>STRC8104</td>
<td>DEBUG=xxx INVALID—IGNORED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>An invalid debug specification is present; the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>first eight characters of the invalid operand</td>
</tr>
<tr>
<td></td>
<td></td>
<td>are listed as xxx. That option is ignored.</td>
</tr>
<tr>
<td>8</td>
<td>STRC8105</td>
<td>SAVETRACE MUST BE SPECIFIED ON FIRST PROC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The SAVETRACE debug option must be enabled on</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the first PROC. The operand has been ignored.</td>
</tr>
<tr>
<td>8</td>
<td>STRC8106</td>
<td>SAVETRACE REQUIRES FIRST PROC TO BE LINKAGE=OS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The SAVETRACE debug option is valid only if the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>first PROC includes the LINKAGE=OS specification.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The operand has been ignored.</td>
</tr>
<tr>
<td>4</td>
<td>STRC8107</td>
<td>REG 1 MUST BE AMONG THOSE SAVED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Register 1 was destroyed during the GETMAIN for</td>
</tr>
<tr>
<td></td>
<td></td>
<td>a dynamic save area and registers 14 through 12</td>
</tr>
<tr>
<td></td>
<td></td>
<td>were not specified (or defaulted) as being saved</td>
</tr>
<tr>
<td></td>
<td></td>
<td>and WORK=NONE was specified. No further check is</td>
</tr>
<tr>
<td></td>
<td></td>
<td>made to assure register one was among those saved,</td>
</tr>
</tbody>
</table>
but the restore has been issued. If register 1 was not among those saved, its value will be undefined.

PROC proc-name, DEBUG ID=X'hh'

The save-trace, proc-trace, or proc-count debug option has been specified for this proc. The hex id byte hh will be used to identify this proc in the labels and dumps. This message is generated so that the user will know the proc id number even if "PRINT NOGEN" has been specified.

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.

RESTORE= was coded on the CORP macro, but SAVE=None was coded on the PROC. The operand is ignored.

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.

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.
<table>
<thead>
<tr>
<th>Severity</th>
<th>Message-Number</th>
<th>Message-Text</th>
</tr>
</thead>
<tbody>
<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 \textit{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 \textit{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 \textit{opcode} and a bc-spec.</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>STRC9402</td>
<td>SUPERFLUOUS 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 more than 5 items.</td>
</tr>
<tr>
<td>8</td>
<td>STRC9403</td>
<td>NO CONDITION SPECIFIED—&quot;MASK=0&quot; ASSUMED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A void simple conditional has been specified.</td>
</tr>
<tr>
<td>8</td>
<td>STRC9601</td>
<td>ONE BLEND ASSUMED TO GET TO &quot;type&quot; BLOCK</td>
</tr>
<tr>
<td></td>
<td></td>
<td>In a block terminating macro (such as FI), the current block was not of the corresponding type (such as IF), but the surrounding block is of the proper type. One BLEND has been inserted.</td>
</tr>
<tr>
<td>8</td>
<td>STRC9602</td>
<td>TWO BLENDS ASSUMED TO GET TO &quot;type&quot; BLOCK</td>
</tr>
<tr>
<td></td>
<td></td>
<td>In a block terminating macro (such as FI), the current block was not of the corresponding type (such as IF), but the second surrounding block is of the proper type. Two BLENDS have been inserted.</td>
</tr>
<tr>
<td>8</td>
<td>STRC9603</td>
<td>CURRENT BLOCK IS NOT &quot;type&quot; BLOCK—MACRO IGNORED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>In a block terminating macro (such as FI), the current block was not of the corresponding type (such as IF).</td>
</tr>
<tr>
<td>8</td>
<td>STRC9604</td>
<td>NO ACTIVE BLOCK NAMED &quot;blname&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The request to terminate block blname has been ignored because no block named blname is in the nest.</td>
</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>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>8</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>8</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>8</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>8</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;) are present.</td>
</tr>
<tr>
<td>8</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>8</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>8</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>8</td>
<td>STRC9706</td>
<td>SYNTAX ERROR—LOOKING FOR SIMPLE CONDITIONAL, FOUND &quot;xxx&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The operand xxx was found in a conditional expression where a simple conditional sublist was expected.</td>
</tr>
<tr>
<td>*</td>
<td>STRC9902</td>
<td>START OF BLOCK nn (blname) AT DEPTH level</td>
</tr>
<tr>
<td></td>
<td></td>
<td>In response to the debug option LISTBLOCKS, the message indicates the start of the block whose sequential number is nn. The block name is blname; if no name was specified on the block initiation macro, blname is an internal name of the form BLKnn. The current static nest level is level.</td>
</tr>
<tr>
<td>*</td>
<td>STRC9903</td>
<td>END OF BLOCK nn (blname) AT DEPTH level</td>
</tr>
<tr>
<td></td>
<td></td>
<td>In response to the debug option LISTBLOCKS, the message indicates the end of the block whose sequential number is nn. The block name is blname; if no name was specified on the block initiation macro, blname is an internal name of the form BLKnn. The current static nest level is level.</td>
</tr>
</tbody>
</table>
APPENDIX E

INSTALLING THE STRCMACS

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

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

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

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

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

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

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

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

Any comments, suggestions, or criticisms of the macros will be greatly appreciated.
<table>
<thead>
<tr>
<th>Term</th>
<th>Page</th>
<th>Term</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>abnormal block exit</td>
<td>27</td>
<td>CORP</td>
<td>31, A-10</td>
</tr>
<tr>
<td>abstract programming language</td>
<td>8</td>
<td>CSECT</td>
<td>35, A-30</td>
</tr>
<tr>
<td>addressability</td>
<td>42</td>
<td>current block</td>
<td>17</td>
</tr>
<tr>
<td>angle brackets</td>
<td>19, A-2</td>
<td>current nest level</td>
<td>17</td>
</tr>
<tr>
<td>asynchronous IF block</td>
<td>20</td>
<td>debug counts</td>
<td>40</td>
</tr>
<tr>
<td>ATEND</td>
<td>29, A-3</td>
<td>debugging aids</td>
<td>38, A-33</td>
</tr>
<tr>
<td>ATEXIT</td>
<td>30, A-4</td>
<td>decision making</td>
<td>18, 23</td>
</tr>
<tr>
<td>base register</td>
<td>34, A-32</td>
<td>defining modules</td>
<td>30</td>
</tr>
<tr>
<td>bc-spec</td>
<td>A-1</td>
<td>diagnostic messages</td>
<td>D-1</td>
</tr>
<tr>
<td>BCT</td>
<td>21</td>
<td>distribution medium for</td>
<td></td>
</tr>
<tr>
<td>BLEND</td>
<td>16, 19, 23, 27, 38, A-5</td>
<td>STRCMACS</td>
<td>E-1</td>
</tr>
<tr>
<td>BLISS</td>
<td>6, 7, 10</td>
<td>DO decision table</td>
<td>B-7</td>
</tr>
<tr>
<td>BLOCK</td>
<td>16, A-6</td>
<td>DO operand format</td>
<td>B-7</td>
</tr>
<tr>
<td>block counts</td>
<td>40</td>
<td>DOCASE</td>
<td></td>
</tr>
<tr>
<td>block exit</td>
<td>27</td>
<td>character compare</td>
<td>26, A-7, A-15</td>
</tr>
<tr>
<td>blocks</td>
<td>16</td>
<td>type</td>
<td></td>
</tr>
<tr>
<td>BXH</td>
<td>21</td>
<td>conditional test type</td>
<td>26, A-7, A-15</td>
</tr>
<tr>
<td>BXLE</td>
<td>21</td>
<td>simple type</td>
<td>25, A-15</td>
</tr>
<tr>
<td>CASE</td>
<td>7, 23, A-7</td>
<td>DOCASEEND</td>
<td>27, A-16</td>
</tr>
<tr>
<td>case ranges</td>
<td>24</td>
<td>ELSE</td>
<td>20, A-18</td>
</tr>
<tr>
<td>CASEEND</td>
<td>27, A-9</td>
<td>DROP</td>
<td>35</td>
</tr>
<tr>
<td>COBOL</td>
<td>10</td>
<td>efficiency</td>
<td>14</td>
</tr>
<tr>
<td>conditional expression</td>
<td>19, 21, 26, A-1</td>
<td>error messages</td>
<td>D-1</td>
</tr>
<tr>
<td>conditional relations</td>
<td>19</td>
<td>ESAC</td>
<td>23, A-19</td>
</tr>
</tbody>
</table>

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

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

E-5