Technical Report No. 79-88

STRUCT79:

A Macro Package to Simulate High-Level Control Structures in MACRO-11

by

Dr. Michael Levison

Department of Computing & Information Science
Queen's University, Kingston, Ontario, Canada

© 1979 M. Levison

Introduction

STRUCT is a set of MACRO-11 macros which give this assembly language the appearance of possessing high-level control structures, including conditional, iterative and case statements, declarations, procedures and interrupt procedures. These have been used extensively both in large and small programs to simplify programming and to improve structure and readability. The first version of STRUCT was described in Technical Report 77-53; the version described here represents both a considerable extension and a reimplementation of existing features. There are a few additional reserved words, but otherwise, existing programs should be unaffected unless they make use of implementation details.

This report is self-contained and incorporates all relevant material from the earlier one.

Simple Conditions

Conditions take one of the forms:

```
<op> <rel> <op>
<op> <rel>
<rel>
```

where

<op> is any MACRO-11 operand

and <rel> is .EQ. | .LT. | .HIS. | ....

(i.e. any similar operator derivable from a PDP-11 conditional branch.)

The first of these is analogous to a compare and a branch, the second to a test and a branch, the third to a branch only.

Examples:

```
%1  .GT.  #4          [ R1 > 4      ] ?
X(R2)  .LT.  Y        [ X(R2) < Y   ] ?
(R6)+  .NE.           [ (R6)+ ^= 0  ] ?
.EQ.                  [ Z bit set   ] ?
```

Conditions are used within while, repeat and conditional statements. The construction of more complex conditions, including those linked with Boolean operators, is discussed in a later section.

While Statements

While statements take the form:

```
WHILE <condition>
    <code>
ENDWHILE
```

where

<code> is any MACRO-11 code.

The statement allows <code> to be executed repeatedly while the condition is true. If the condition is omitted, an infinite loop is formed (and presumably there must be some other form of exit).

WHILE may be replaced by any of WHILB, WHILJ, WHILBJ. WHILB and WHILBJ cause CMPB and TSTB instructions to be generated instead of CMP and TST. WHILJ and WHILBJ cause JMP instructions to be generated between distant parts of the statement instead of branches. WHILEB and WHILEJ are synonyms for WHILB and WHILJ respectively.

In all cases, the operands are evaluated only once per loop, so that any MACRO-11 operands, including auto-increments, may be used in the conditions.

There is a DO macro which may be placed on the line preceding the body of the loop. It is normally omitted, but must be present if the loop contains an 'inverting' complex condition (see below). The DO macro may have another valid MACRO-11 instruction (including one of these structure macros) on the same line, if desired.

Repeat Statements

Repeat statements take the form:

```
REPEAT
  <code>
ENDREPEAT
```

and cause <code> to be executed repeatedly. They normally contain either an UNTIL or a COND clause, giving rise to repeat-until or repeat-while statements. If no such exit is present, an infinite loop is formed.

No jump variant has been implemented in this case. The terminating macro itself determines whether to assemble a branch or jump (but conditional clauses too far from the ENDREPEAT will cause errors).

The REPEAT macro may have another valid MACRO-11 statement on the same line, if desired.

Conditional Statements

Conditional statements take the form:

```
IF <condition>
    THEN
        <code>
    ELSE
        <code>
ENDIF
```

The ELSE part may be omitted, and IF may be replaced by any of IFB, IFJ, IFBJ. The interpretation is analogous to that of the WHILE statement.

The THEN and ELSE macros may have another valid MACRO-11 statement following on the same line, if desired.

For Statements

For statements take the form:

```
FOR <op1> FROM <op2> TO <op3> BY <op4>
    <code>
ENDFOR
```

and cause the <code> to be executed for each value of <op1>, starting with <op2>, and going UP in increments of <op4>, as long as <op1> does not exceed <op3>. All or any of the parts FROM <op2>, TO <op3>, and BY <op4> may be omitted. In default, the iteration will begin at the current value of <op1>, the loop end test will be omitted, and the increment will be taken to be 1, respectively. Note that the increment must be positive. For negative increments, FOR is replaced by FORDN. FORJ and FORDNJ provide JMP versions of these macros. <op3> and <op4> are referenced only once per cycle, but <op4> is referenced one additional time during loop

initialization. <op1> is referenced several times per cycle, so that auto-increment operands should not be used. <op2> is called only once for the entire statement.

An alternative iterative statement takes the form:

```
DECR <reg> FROM <op>
  <code>
ENDDECR
```

where <reg> is the name of a register. This causes <code> to be obeyed while the register takes each value from <op> down to 1. Its purpose is to allow the programmer access to the SOB instruction on those PDP-11s which have it. FROM <op> may be omitted, in which case the iteration starts at the current value of the register. Note that <code> must not be too large (that is, it must conform to the limit imposed by the SOB instruction), and that the loop test follows loop body (so that the register is left at 0).

Complex Conditions

Complex relational expressions, combinations of several conditions, loop exits, repeat-until statements, and so on, can be put together using COND (synonymously AND), OR and UNTIL macros, which take the form:

```
COND <condition>
```

These macros may be used in any of the (vertical) combinations:

```
AND     UNTIL     OR      UNTIL     UNTIL
AND     UNTIL     OR      OR        AND
AND     UNTIL     OR      OR        AND
...     ...       ...     ...       ...
AND     UNTIL     OR      OR        AND
```

the first four of which may appear within while, for, repeat or if statements; the last in repeat statements only.

The sequence of AND macros causes a branch to the exit (or, in the case of an if statement, to the ELSE part where one exists) when a false condition is found.

The sequence of UNTIL macros causes a branch to the loop exit when a true condition is found. (Note that this

combination does not read well in if statements.) The UNTIL-OR sequence is synonymous.

The sequence of OR macros used in while, if or for statements causes a branch to the body of the loop (or, in the case of an if statement, to the THEN part) if any of the conditions, including the one on the WHILE, IF or FOR, is true. This sequence is one of two referred to as 'inverting' (it inverts the meaning of the preceding WHILE, and causes the interchange of certain subsequent labels). It imposes the following special requirements: the first OR macro must follow immediately after the WHILE, IF or FOR; the body of the loop (or the THEN part) must be preceded by a DO (or THEN) macro; and the DO (or THEN) must immediately follow the final OR.

The sequence of OR macros used in a repeat statement is not 'inverting', and is synonymous with the sequence of UNTILs.

The UNTIL-AND sequence (used in a repeat statement only) causes a branch to the loop body if any of the conditions is false. This too is an 'inverting' sequence. The first AND must follow immediately after the UNTIL, and the ENDREPEAT immediately after the final AND (no DO is necessary; the ENDREPEAT embodies one).

Except in the cases mentioned, the individual ANDs, ORs and UNTILs of a sequence may be separated by sections of code. Applications are given below.

As the examples show, the sequences specified produce 'obvious' effects. Other combinations, notably those which mix AND and OR, do not, and should be avoided. Nor should more than one of these sequences be included in the same structure.

If a byte condition is to be tested, CONDB, ORB and UNTILB are substituted for COND, OR and UNTIL. AND and ANDB are synonyms for COND and CONDB. There are no 'jump' forms of the macros, the compilation of branches or jumps being determined by the statement in which a macro occurs.

EXAMPLES:

(1) To perform a loop while (p+q)=(r+s)

```
WHILE
  mov p,x
  add q,x ; x:=p+q
  mov r,y
  add s,y ; y:=r+s
  COND  x .EQ. y

    <loop body>

ENDWHILE
```

(2) To perform a piece of code if two conditions p>q and r>s hold:

```
IF  p .GT. q
AND r .GT. s
  THEN

    <code>

ENDIF
```

As many ANDs as desired may be included, each optionally preceded by some code which assists in computing the condition. The 'assisting' code may contain other structures nested within it (including other IF and WHILE statements which contain CONDs, etc.)

(3)
```
IF x .LT. #3
OR x .GE. #20
  THEN

    <code>

ENDIF
```

(4)
```
WHILE y .EQ.
    OR    z .EQ.
      DO

        <code>

    ENDWHILE
```

Again, as many ORs as desired may be included, but the first may have no 'assisting' code. Note too that the THEN/DO abuts the final OR.

(5)
```
REPEAT

  <code>

UNTIL p .LT. q
ENDREPEAT
```

(6)
```
REPEAT UNTIL x .HIS. y

    <code>

    ENDREPEAT
```

(7)
```
FOR I FROM #1
UNTIL r1 .GT. r2

  <code>

ENDFOR
```

(8)
```
REPEAT

    <code>

UNTILB char .EQ. #040
ORB    char .EQ. #'A
ENDREPEAT
```

NOTE

The user should remember that conditions containing operands cause the assembly of CMP or TST instructions which alter the PDP-11 condition codes. Hence in multiple conditions, those without operands should be placed first; e.g.

```
IF .EQ.
AND p .GT. q
```

determines whether the Z-bit is set and whether p>q, but the reverse does not because the comparison of p and q alters the Z-bit.

Case Statements

Case statements take the form:

```
CASE <op>
    STMNT <int1>,<int2>,...
        <code1>
    STMNT <int3>,...
        <code2>
    ...
    ELSE
        <codeN>
    ...
ENDCASE
```

and may be either one- or two-dimensional.

If the case statement is one-dimensional, <op> is a word operand (evaluated only once), and <int1>, <int2>, ... are integers; if it is two-dimensional, <op> is a pair of word operands (each evaluated only once), and <int1>, <int2>, ... are each pairs of integers. Each pair of integers and operands is enclosed in angle-brackets.

The effect of a case statement is to evaluate <op>, locate its value among the integers (or pairs) which follow the STMNT macros, and then execute whichever one of <code1>, <code2>, ... follows that STMNT.

Each STMNT may be followed several integers (or pairs), a STMNT without a parameter being ignored.

The optional ELSE part provides a target for all cases within 'jump table range' which do not appear in STMNT macros. If the ELSE part is omitted, these cases simply cause a jump past the case statement. All such cases are flagged with a warning message during assembly. Note that the execution of a case outside 'jump table range' is not detected, and its effect is potentially disastrous. ELSE may, as usual, have another MACRO-11 statement on the same line.

'Jump table range' is defined as follows:

(one-dimensional)
if n is the largest integer which appears in a STMNT macro, the range is 0 to n;

(two-dimensional)
for every pair <r,c> which occurs in a STMNT macro, the range includes <r,0>, <r,1>, ... , <r,c> (all earlier columns of the same row), and also <0,0>, <1,0>,..., <r-1,0> (the zero column of each earlier row).

Thus, the appearance of <0,2>, <3,3> and <5,2> in STMNTs causes

```
<0,0> <0,1> <0,2>
<1,0>
<2,0>
<3,0> <3,1> <3,2> <3,3>
<4,0>
<5,0> <5,1> <5,2>
```

to be in jump table range.

For small integer byte operand(s), CASE is replaced by CASEB. In the two-dimensional case, both operands are either bytes or words (i.e. not mixed). If CASE, CASEB are replaced by CASEJ, CASEBJ, JMP instructions will be generated instead of branches.

WARNINGS

(1) In estimating the size of code, it should be noted noted that a jump table is generated at the position of the ENDCASE. The table contains one element for every case number (or pair) in jump table range, and an additional element for every 'row' in the two-dimensional case.

(2) Note that

```
STMNT 1,2
<code1>
```

is NOT equivalent to

```
STMNT 1
STMNT 2
<code1>
```

The second causes the null statement to be executed for value 1 of the operand.

(3) A jump to a label placed on a STMNT macro, will cause control to pass to the end of the case statement, not to the code following the STMNT.

(4) The ELSE part may be positioned anywhere in the case statement after the first STMNT.

Procedures

Procedure declarations take the form:

```
PROCEDURE <name> <reg> GLOBAL
    <code>
ENDPROC
```

where

<name> is the procedure name,

and the register <reg> and parameter GLOBAL are optional.

Procedure calls take the form:

```
<name>  <op1>,<op2>,...
```

where

<op1>,... are optional MACRO-11 word operands.

The effect of the call depends on the register parameter of the declaration. If the register parameter is absent or is R7, the effect is to place the operands on the stack (in reverse order) and call the procedure. Thus, inside the procedure, <op1> will be found in 2(R6), <op2> in 4(R6), and so on, so long as the procedure itself does not alter the stack. Encountering ENDPROC causes the stack to be cleared and control to be returned to the calling program.

If the register parameter is some other register, then this register is used as the linkage register in the JSR instruction (and in the corresponding RTS), and the parameters are placed in line after the call. On entry to the procedure, the linkage register will be found to point at the first parameter, so that <op1> is in @Rx, <op2> in 2(Rx), and so on. (Note the discrepancy between this and the stack form of procedure, caused by the presence of the return address on the stack.)

In this latter case, it is the responsibility of the procedure body to ensure that the linkage register is pointing at the word following the final parameter before the ENDPROC is encountered. Recall that the use of a register as linkage register does not affect its global contents, since the JSR and RTS instructions preserve and restore it.

EXAMPLES

(1) A procedure with R7 as register parameter

```
PROCEDURE FRED
  ; Suppose the procedure expects five parameters,
  ; they will be in 2(R6), 4(R6), 6(R6), ... resp.

  <code>

ENDPROC

...

FRED  R2, #45., PQR, 2(R3), @R4
```

This call actually generates

```
MOV @R4,-(R6)
MOV 2(R3),-(R6)
...
JSR R7,FRED
ADD #5+5,R6 ; clean up stack
```

(2) A procedure with R3 as register parameter

```
PROCEDURE FN R3
  ; Suppose the procedure expects five parameters,
  ; they will be in @R3, 2(R3), 4(R3), ...   resp.

  <code>

ENDPROC

...

FN   XYZ,#A,#5,R2,4(R4)
```

This call actually generates

```
        MOV XYZ,p1
        MOV R2,p4
        MOV 4(R4),p5
        JSR R3,FN
    p1: .WORD  ; space to receive contents of XYZ
    p2: .WORD A
    p3: .WORD 5
    p4: .WORD  ; space to receive contents of R2
    p5: .WORD  ; space to receive contents of 4(R4)
```

where p1,p2,p3,p4,p5 denote local addresses computed by the calling macro. On entry, R3 contains p1; before exit, the procedure body must ensure that R3 contains the address of the word following p5.

If the parameter GLOBAL is present, the procedure will be available to be called from other modules. In each such other module, there must be an 'external' declaration of the form:

PROCEDURE <name> <reg> EXTERNAL

A procedure must be declared before it is called. To allow a call to occur earlier in the module than the procedure body, a forward declaration is provided in the form

PROCEDURE <name> <reg> FORWARD

In these two cases, there is of course no code, and no ENDPROC.

WARNINGS

(1) With either of the two call types, only WORD operands are permitted, failing which an addressing error may occur.

(2) There must be exactly one ENDPROC per procedure; further returns can, if necessary, be effected by RTS instructions with the appropriate linkage register (R7 if absent).

(3) The system will not detect the discrepancy if the user specifies different linkage registers in the body declaration and the EXTERNAL and/or FORWARD declarations of the same procedure.

PUSH and POP

The macro calls

PUSH <op1>,<op2>,<op3>,...

and

POP ...,<op3>,<op2>,<op1>

cause the specified MACRO-11 operands to be placed on, and removed from the stack respectively. PUSHB and POPB variants are also available for the manipulation of byte variables. If PUSH occurs without a parameter, a cleared word is pushed onto the stack. If POP occurs without a parameter, stack is popped, and the top item lost.

Interrupts and Traps

The declaration

```
INTERRUPT <name> <intvect> <contr> GLOBAL
  <code>
ENDINT
```

may be used to declare an interrupt routine, where

<name> is the routine name (it will also be used to name the associated device),

<intvect> is the interrupt vector address of the associated device, and

<contr> is the control/status register addr of the associated device.

The parameter <contr> is optional (traps and some devices, such as the PDP-11/03 line-clock, have no control register) as is the parameter GLOBAL. The parameters <intvect>, <contr> and GLOBAL may occur in any order.

<intvect> and <contr>, unlike other parameters in STRUCT, must be integers, or symbols defined as integers.

Once such a declaration has occurred, the user may call

```
INSTALL <name> <ps>
```

or synonymously

```
<name> <ps>
```

to load the address of the routine and the parameter <ps> into the interrupt vector position named in the routine declaration. If <ps> is absent, the processor-status part of the interrupt vector is cleared.

The calls DISABLE <name> and ENABLE <name> respectively disable and enable interrupts from the device. These are, of course, meaningful only if there is a control register.

Example:

```
INTERRUPT KEYBD 177560,60
                ; these are  the addresses of the
                ; console keyboard

  <code>        ; does  whatever  is desired with
                ; character typed on console keys

ENDINT

...

INSTALL KEYBD ; sets up interrupt vector (ps zero)

ENABLE KEYBD  ; enables keyboard interrupts
```

If the parameter GLOBAL is present, the interrupt routine can be INSTALLed, and interrupts ENABLEd or DISABLEd from other modules. In each such other module, there must be an 'external' declaration of the form:

```
INTERRUPT <name> EXTERNAL
```

The routine declaration must precede in the module any INSTALL, ENABLE or DISABLE which refers to it. To allow any of these operations to occur earlier in the module than the routine body, a forward declaration is provided in the form

```
INTERRUPT <name> FORWARD
```

In these two cases, there is of course no code, and no ENDINT.

Declarations

Declarations take one of the forms:

```
WORD <id>,<id>,...
```

or

```
BYTE <id>,<id>,...
```

where each <id> is either a MACRO-11 identifier or else a doublet or triplet enclosed in angle brackets; for example:

```
WORD X,Y,<M,777>,Z,<A,6,101>
```

The first item of each doublet or triplet is a MACRO-11 identifier, the others are integers (or assembly-time variables).

The effect of the given declaration is to reserve a word for each of X,Y,M,Z, initializing M to 777, and an array of 6 words for A, initializing each to 101. After each complete BYTE line, .EVEN is generated to ensure that the subsequent line begins on a word boundary.

Warning:

In the event that variables are used as the second or third items of doublets or triplets, it must be remembered that it is the assembly-time value of the variable which is used in the declaration. Thus WORD <X,22>,<M,X> will initialize M to the address of X, not to its run-time contents,22. The usual MACRO-11 restrictions as to whether variables must be defined before use apply here.

Other forms of declaration are obtained using conventional MACRO-11 facilities (i.e. .ASCII, .BLKW, .WORD, and so on).

BEGIN and END

Following the definitions of the macros which implement the above constructions, the program must be preceded by the macro call

BEGIN <macro name>,<macro name>,...

where the optional <macro name>s are macros in the system library file SYSMAC.SML . BEGIN initializes certain parameters used in the code generation and generates .MCALLs to the macros in the list. The program may also be terminated by a macro call of the form:

END <label>

This replaces the usual MACRO-11 .END directive; its purpose is to generate the latter, and also to check that all the earlier constructs have been properly terminated.

Nesting and Restrictions

The above constructions may be nested within one another to a depth of 64, but any beginning (IF,WHILE,...) must be matched by the corresponding ending (ENDIF, ENDWHILE, ...). Mismatches will be detected and reported by the assembler.

The number of one-dimensional case statements may not exceed 56, and the case numbers must be less than 512; the number of two-dimensional case statements may not exceed 8, and each part of the case 'pair' must be less than 64.

Parameter lists are in all cases restricted to 16 in length.

The following are macro names, and are thus 'reserved':

IF	IFJ	IFB	IFBJ	ENDIF
	THEN	ELSE		
.EQ.	.LT.	.HIS.	etc.	
WHILE	WHILJ	WHILB	WHILBJ	ENDWHILE
	WHILEJ	WHILEB		
	DO			
REPEAT				ENDREPEAT
	UNTIL	UNTILB		
COND	CONDB			
AND	ANDB			
OR	ORB			
FOR	FORJ	FORDN	FORDNJ	ENDFOR
DECR				ENDDECR
CASE	CASEJ	CASEB	CASEBJ	ENDCASE
	STMNT			
PROCEDURE				ENDPROC
INTERRUPT				ENDINT
	INSTALL	ENABLE	DISABLE	
PUSH	PUSHB	POP	POPB	
WORD	BYTE			
BEGIN	END			

The names GLOBAL, EXTERNAL, FORWARD, FROM, TO and BY are variables, and are also 'reserved'.

Note that, for cosmetic reasons, names longer than six characters have been used, but that only the first six are distinguished by the assembler.

The names of all subsidiary macros, and of all labels generated by the macros, contain one of the symbols . or $. Thus names containing these symbols should be avoided.

Implementation and Use

The implementation comprises about eighty macros, approximately 750 lines of code. These are available as a file which can be assembled at the start of each MACRO-11 program. Locally, they have been included in an expanded version of the system macro library SYSMAC.SML, and an additional macro STRUCT has been written containing a .MCALL to all of the others. In this case, it is simply necessary to start each program with

```
.MCALL STRUCT
STRUCT
```

Essentially, the initial macro of each structure (IF, IFB,WHILE,...) occurring in a program is assigned an integer, which is used in the generation of all labels required for that structure. This integer, together with certain other information (whether it is a branch or jump macro, etc.), is placed on a stack for use when the terminal (ENDIF,ENDWHILE, ...) or intermediate (ELSE,STMNT) macros of the structure occur. Although MACRO-11 does not explicitly provide assembly-time stacks, the effect can be achieved indirectly, because it is possible to assign an integer value to an assembly-time variable, and use the value to construct further variable names. For example, the variables $LEV1, $LEV2,... , constructed by concatenating the string $LEV with the string corresponding to the present value of the variable $V, constitute such a stack.

The macro call PROCEDURE X generates not only a label X on the code which follows, but also a further macro definition for a macro named X whose body is a subroutine jump to label X. Happily there is no clash between label X and macro X in MACRO-11. (In this case, it was not possible to use a label generated from the structure's integer because it was required to be able to generate a jump to a global procedure from a separately assembled module, where this integer would not be known).

In other regards, the macros are fairly straightforward. Further details are given in the appendix to this report.

Error Messages

In addition to the usual error messages which arise from the assembler, the macros generate a few of their own. Each takes the form

level p:message

where p indicates static nesting level of the structure, and message is one of

```
mismatched END
misplaced register parameter
missing coordinate                  (two-dimens case)
too many double CASE statements     (i.e. two-dimens)
```

A further message, prefixed by WARNING, is

missing STMNT x

This is not a syntax error, as the ELSE part will be executed if the missing case is attempted. The warning is issued in case the omission was unintentional.

The present implementation does not check the 'number' restrictions mentioned in an earlier section. If they are violated, the result will usually be a label or variable name with more than six characters, which will be truncated by the assembler and thus clash with some other name. Parameter lists with more than sixteen entries are merely truncated.

Branches (rather than jumps) over too long a piece of code give rise to the usual MACRO-11 error (****** A). The user should realize that the line flagged with this error is not necessarily the initial line of a construct; for example, the ELSE macro of an if statement, or some of the STMNTs of a case statement, will be flagged, not the IF or CASE itself.

Failure to declare a procedure before using it causes a macro to be called before it is defined, and this usually leads to dozens of 'phase' errors (****** P).

Caveat User

When the macros are used in the manner described above, they appear to generate correct and (for the most part) irredundant code. Experience, however, has suggested the following hints which the user should bear in mind:

(1) In the matter of deciding whether to use the branch or jump variants, one should always use branches (i.e. IF, IFB rather than IFJ, IFBJ). In a well structured well procedured

program, even one a few thousand lines long, there are usually only two or three instances where jumps are necessary (these almost always involve case statements with many simple cases), and it is very easy to change these after assembly reveals an error.

(2) The error checking and recovery procedures in the present implementation are not elaborate. Some mistakes in the program lead to MACRO-11 errors in the code generated by the macros, and this code may not be immediately recognizable to the programmer. Recovery from a mismatched END sometimes causes a cascade of further errors.

(3) Since the structure macros themselves call internal macros to a depth of several levels, it is very unwise to include the directive .LIST ME in the program. Doing so will cause a very lengthy listing of material not recognizable to the user. A more satisfactory listing of the code produced by the macros can be obtained by using the directive .LIST MEB , but this will omit the generated labels (as these occur on lines by themselves), and will include some 'immediate conditionals' of the form .IIF ... , with the generated code on the end of the line.

(4) Most of the macros generate non-local labels. Thus it is not usually possible to branch to a local label across one of the macros. (Actually, the programmer using these macros should not be branching explicitly anyway!!).

(5) The user should beware of ascribing to the 'language' features which it does not possess. It does not, for example, have 'block structure'. Furthermore, if the user chooses to declare a procedure at some inner point of the program, he must remember to supply a branch over the body of the procedure.

Conclusion

The macros described here have been used extensively by Computer Science students at Queen's, and also in the development of several very large programs. Although they increase, perhaps twofold, the time needed for an assembly, this is more than compensated by the time saved in programming and by the readability of the resulting code.

Acknowledgement

The earlier version of STRUCT from which the present version is adapted was implemented by R. L. Stevens.

Appendix

To assist in locating any problems which might arise, we outline here the code generated by the individual macros.

Notation

In the following we denote by

```
<bfalse $Kxxxx>
```

the sequence

```
<compare>
<branch $Kxxxx>
```

where <compare> is one of CMP,CMPB,TST,TSTB or is null, <branch> is of the form

```
BNE $Kxxxx
```

or

```
BEQ .+6
JMP $Kxxxx
```

(when the condition being tested contains .EQ.), xxxx is the integer assigned to the present structure, expressed in octal with four or fewer digits, and K is some letter of the alphabet.

We denote by

```
<btrue $Kxxxx>
```

the analogous sequence with the conditional branches inverted, and by

```
<uncond $Kxxxx>
```

an unconditional branch or jump to the label $Kxxxx.

IF,WHILE,...

```
$Sxxxx:
  <bfalse $Gxxxx>
```

```
REPEAT

                $Sxxxx:

UNTIL,...

                  <btrue  $Gxxxx>

AND,...

                  <bfalse $Fxxxx>     (normally)


                  <uncond $Fxxxx>     (if this is the first AND
                $Gxxxx:                of an 'inverting' sequence)
                  <bfalse $Fxxxx>


OR,...

                  <btrue  $Fxxxx>     (normally)


                  <uncond $Fxxxx>     (if this is the first OR
                $Gxxxx:                of an 'inverting' sequence)
                  <btrue  $Fxxxx>

THEN,DO

(normal)        $Uxxxx:


('inverting')     <uncond $Uxxxx>
                $Fxxxx:
```

Actually the label $Uxxxx is not referenced in the normal case and can be omitted, as indeed can the macro itself. The author has usually included THEN for cosmetic purposes, but omitted DO. (The suggestion in the earlier report of placing the cosmetic THEN as the fourth parameter of an IF macro --where it would be ignored-- does not of course apply to the THEN required by the 'inverting' sequence.)

```
ELSE

(normal)          <uncond $Txxxx>
                $Fxxxx:
                $Gxxxx:
```

```
('inverting')     <uncond $Txxxx>
                  $Uxxxx:
```

ENDIF

```
(normal)          $Fxxxx:                 (if there was
                  $Gxxxx:                                 no ELSE)

                  $Txxxx:

('inverting')     $Uxxxx:                 (if there was no ELSE)

                  $Txxxx:
```

ENDWHILE,ENDFOR

```
(normal)            <uncond $Sxxxx>
                  $Fxxxx:
                  $Gxxxx:

('inverting')       <uncond $Sxxxx>
                  $Uxxxx:
```

ENDWHILE and ENDFOR determine whether to generate a branch or jump by computing the distance back to the S label (WHILJ, WHILBJ, ... are still necessary, however, because of the forward jumps).

ENDREPEAT

This is essentially the same as

```
DO
ENDWHILE
```

FOR,...

```
  MOV <op2>,<op1>     (if FROM <op2> is present)
  SUB <op4>,<op1>     (if  BY  <op4> is present;
                          otherwise DEC <op1>  )
$Sxxxx:
  ADD <op4>,<op1>     (if  BY  <op4> is present;
                          otherwise INC <op1>  )

  CMP <op1>,<op3>     (if TO <op3>  is
  BGT $Fxxxx                                present)
```

The conditional branch is replaced by

```
BLE .+6
JMP $Fxxxx
```

for the jump variant; FORDN,FORDNJ are analogous.

DECR

```
  MOV <op>,<reg>      (if <op> present)
$Sxxxx:
```

ENDDECR

```
  SOB <reg>,$Sxxxx
```

(For those PDP-11s missing this instruction, this structure would have to be simulated by the corresponding FORDNJ structure)

CASE,...

```
$Sxxxx:
  MOV <op1>,-(%6)
  ASL @%6
  ADD #$Hxxxx,@%6

  MOV @(%6),@%6      |  in
  MOV <op2>,-(%6)    |    the
  ASL @%6            |      two-dim
  ADD (%6)+,@%6      |        case

  MOV @(%6)+,%7
```

Both lines MOV <op>,-(%6) are replaced in the byte variants by

```
CLR -(%6)
MOVB <op>,@%6
```

STMNT

Note that case statements are each given individual numbers in addition to the regular structure numbers; these run from 0 to 7 for two-dimensional case statements, 10 to 77 (octal) for one-dimensional.

(one-dimensional)

```
        <uncond $Txxxx>
$nnppp:
$nnqqq:
```

where nn is the number of the case statement (exactly two octal digits), and ppp,qqq,... are the case numbers (three or fewer octal digits).

(two-dimensional)

```
        <uncond $Txxxx>
.nrrcc:
.nssdd:
```

where n is the number of the case statement (exactly one octal digit), rr is the 'row' of the case pair (exactly two octal digits, a period being used as leading 'zero' if required), and cc is the 'column' number (two or fewer octal digits).

The unconditional branch on the first STMNT of each CASE statement is not generated.

ENDCASE

```
        <uncond $Txxxx>

$Hxxxx:
        <jump table>

$Fxxxx:                 (if there was
$Gxxxx:                                 no ELSE)

$Txxxx:
```

where <jump table> is as follows:

(one-dimensional)

```
        .WORD $nn0
        .WORD $nn1
           ...
        .WORD $nnqqq
```

where nn is the number of the case statement, and qqq is the highest case number which occurred.

(two-dimensional)

```
          .WORD .n0
          .WORD .n1
             ...
          .WORD .nrr
        .n0:
          .WORD .n.00
          .WORD .n.01
             ...
          .WORD .n.0cc
        .n1:
          .WORD .n.10
          .WORD .n.11
             ...
          .WORD .n.1dd
             ...
        .nrr:
          .WORD .nrr0
          .WORD .nrr1
             ...
          .WORD .nrree
```

where n is the number of the case statement, rr is the highest 'row' which occurs, cc the highest 'column' in row 0 and so on.

In both one- and two-dimensional cases, if one of the $nnppp or .nrrcc labels has not been declared, .WORD $Fxxxx is generated instead of .WORD $nnppp or .WORD .nrrcc, and the corresponding case will cause a jump to the ELSE part (or past the table if there is no ELSE).

PUSH,PUSHB

For each parameter:

```
MOV <param>,-(%6)     or MOVB <param>,-(%6)
```

POP,POPB

For each parameter:

```
MOV (%6)+,<param>     or MOVB (%6)+,<param>
```

If no parameter is present:

```
CLR  -(%6)    is generated for PUSH, PUSHB
TST  (%6)+    is generated for POP, POPB
```

PROCEDURE

```
        .GLOBL <name>          (if GLOBAL or EXTERNAL)

      <name>:              (if not EXTERNAL or FORWARD)
          .MACRO <name>
             <macrobody to generate
              code specified below>
          .ENDM
```

INTERRUPT

```
        .GLOBL <name>          (if GLOBAL or EXTERNAL)
```

And if not EXTERNAL or FORWARD:

```
         .WORD <controlreg>                (if present)
         .WORD <intvec>
       <name>:
         MOV <name>-2,-(%6)
         ADD #2,@%6
         MOV 4(%6),@(%6)+
         MOV #$.y,@<name>-2
         RTS %7
       $.y:
          .MACRO <name>
             <macrobody to generate
              code specified below>
          .ENDM
```

y being an integer unique to this interrupt routine. Extra labels are generated after label <name>. The code may seem more complex than necessary due to the desire to be able to call ENABLE, ... in others modules.

<name>

Case 1 <name> 'declared' in a PROCEDURE macro

(a) If <reg> is %7, or is not present in PROCEDURE macro

```
        PUSH ...,<param1>  (if there are params)
        JSR %7,<name>
        ADD #n+n,%6        (where n is the number
                                of params)
```

The last line is omitted if n = 0; and replaced by TST (%6)+ or CMP (%6)+,(%6)+ for n = 1 or 2 resp.

(b) For other linkage registers, the code is shown in the body of the report.

Case 2 <name> 'declared' in an INTERRUPT macro

```
MOV <ps>,-(%6)    (CLR -(%6) if <ps> absent)
JSR %7,<name>
TST (%6)+
```

INSTALL

INSTALL <name> is synonymous with <name>.

ENDPROC

```
RTS <reg>          ( <reg> is %7 if param
                   not present in PROCEDURE)
```

ENDINT

```
RTI
```

ENABLE

```
BIS #100,@<name>-4
```

DISABLE

```
BIC #100,@<name>-4
```

WORD,BYTE

These generate the obvious .WORD and .BYTE constructions. Each complete BYTE declaration is followed by .EVEN .

BEGIN

This generates .MCALL if necessary. More importantly, however, it initializes certain assembly-time variables used by the other macros. It is therefore the one macro which MUST be called in the program.

END

```
.END <label>
```

It will be seen that some macros generate unnecessary labels, which shortens the implementation, without increasing the object code. The code generated is irredundant, except in the case that the programmer includes his own jumps or branches out of conditional or case statements.