University of Edinburgh

Department of Computer Science

HAL 70

by

Hamish Dewar

revised by

Jeffrey Tansley

Internal Report CSR-29-78

James Clerk Maxwell Building
The King's Buildings
Mayfield Road
Edinburgh
EH9 3JZ

October, 1978

```
H   H     A     L       777777     0
H   H    A A    L            7    0 0
H   H   A   A   L           7    0   0
HHHHH   AAAAA   L          7     0   0
H   H   A   A   L         7      0   0
H   H   A   A   L         7       0 0
H   H   A   A   LLLLL     7        0
```

A high level assembly language for Interdata series 70 mini-computers.

by

Hamish Dewar

Revised by

Jeffrey Tansley

Department of Computer Science.
Edinburgh University.

First Edition 1975.
Second Edition 1977.
Third Edition 1978.

The Second Edition of this Manual reflects changes and additions made to the Assembler since 1975. Most of these were done by H.D. The revision of the Manual was done by J.T. The Third Edition reflects comments of users since then.

Contents. Page.

0.0 INTRODUCTION

HAL-70 is a programming language for the Interdata 70 mini-computer range. It is one of a family of high-level assembly languages bearing the family name HAL which have been implemented for a number of different computers. The high-level features and the syntax of the language are essentially the same in all versions, but the machine-level features vary to match the hardware capability of particular machines.

The aim of this hybrid type of language is to provide the kind of program-structuring facilities usually found in full-scale high-level languages -- such as block structure and generalised assignment and conditional statements -- while permitting the direct access to machine resources that is associated with an assembly language. Thus the language does not pre-suppose a particular run-time environment; it makes no fixed assumptions about register usage, storage allocation or routine callir conventions; it permits the use of machine instructions [illegible] arbitrary data formats; and it provides for explicit contr[illegible] the positioning of code and data.

In the case of a multi-register machine like the Interdata 70, the language permits uniform reference to registers and storage locations in the majority of contexts, thus avoiding the need to use different instruction formats for the two cases. However decisions regarding which variables should be allocated to registers and which to storage locations over particular sections of code are left to the programmer and the importanc[illegible]ese decisions for the production of efficient and economical code is stressed.

1.0 FORM OF HAL PROGRAMS

1.1 Structure

HAL is a block-structured language. A complete program is a block and any block may have other blocks nested within it. In the usual way the scope of applicability of a name introduced in a block is confined to that block and all blocks textually contained in it which do not define the same name. This localisation of the effect of definitions (and certain other assembler directives) is the sole purpose of the block structure; it has no significance for storage allocation.

1.2 Format

A HAL program consists of a sequence of statements each of which is terminated by a newline or semi-colon. The ASCII character set is assumed. Apart from null statements, which are accepted and ignored, statements belong to one of the following classes:

- assignment statements
- jump statements
- machine-code statements
- conditional bracketing statements
- loop bracketing statements
- data statements
- assembler directives
- macro-call statements

An assignment statement consists of an assignment instruction optionally followed by a conditional clause. Similarly, a jump statement (machine-code statement, macro-call statement) consists of a jump instruction (machine-code instruction, macro-call) optionally followed by a conditional clause.

Assignment instructions, jump instructions and run-time conditions are translated into a variable number of basic instructions while each machine-code instruction is translated into a single basic instruction (occupying one or two 16-bit words depending on the type of the instruction). Each expression in a data statement is translated into a 16-bit value (8-bit value in byte mode).

Assembler directives do not cause any code to be generated but serve to direct the process of assembly and to provide definitions for the names used in a program. The conditions which occur in conditional assembly directives are tested at assembly time to determine which sections of a program are to be assembled on a particular occasion.

A macro call is expanded in accordance with the corresponding macro definition, ultimately into statements belonging to the other classes.

The syntax of individual statements is described in Sections 2 and 3 with the aid of a form of context-free grammar (BNF). The symbol '*' indicates repetition (zero or more occurrences) and the symbol '?' indicates that a statement component is optional.

1.3.0 Basic statement components

1.3.1 Tags

Tags (or identifiers) are symbolic names used for a variety of purposes in program statements -- for example as instruction mnemonics, register names, labels, and symbolic constants. A tag is a string of up to six letters or digits starting with a letter and with all letters preceding any digits. Any extra trailing letters and digits are ignored. Each tag belongs to a certain type, depending on the way in which it is defined, and there are constraints on the type of tags which may be used in different contexts. In particular, instruction mnemonics and register names belong to different types, distinct from other kinds of tag.

Some tags have a pre-defined significance. These include:

machine instruction mnemonics	eg	BAL, MH, ACH
standard register names	eg	R1, R14
keywords	eg	JUMP, IF, W

Other tags (user tags) may be introduced as required in the program. A user tag is defined by means of a tag definition statement (explicit definition) or by being used as a label (implicit definition).

1.3.2 Register names

The following pre-defined tags are provided to designate the machine registers:

```
      R0,  R1,  R2,  R3,  R4,  R5,  R6,  R7,
      R8,  R9,  R10, R11, R12, R13, R14, R15
```

Normally these denote one of the sixteen 16-bit General registers, but in floating-point machine instructions (even numbered registers only) they denote one of the eight 32-bit Floating-point registers. When used in the context of indexing, reference is to the General registers, irrespective of instruction type.

1.3.3 Literals

The following forms are provided:

decimal numerals	eg	0, 2, 999
hexadecimal numerals	eg	X'1000', X'1C4A'
character constants	eg	'A', 'OK'
packed tags	eg	.LP, .EXP

Literals (and expressions made up of literals) are evaluated as 16-bit absolute values, without regard for sign significance or overflow. Twos-complement representation of negative values is assumed. Hexadecimal numerals with fewer than four digits and character constants consisting of a single character are aligned to the right. A tag of up to three characters preceded by a dot denotes the packed internal value used by the assembler for storing tags. This special form is chiefly of use for system programs.

1.3.4 Operators

The operators available for use in expressions are:

binary		unary	
+	plus	\	logical-not
-	minus	-	minus
&	logical-and		
!	logical-or		
\	exclusive-or		
<<	left-shift		
>>	right-shift		

(Note: there is no multiply)

1.3.5 Expressions

An expression is evaluated from left to right but may be overridden by parenthesis or operator precedence rules. This applies even to assigment operators. Thus R4=1+R4 is translated as R4=1; R4=R4+R4; and sets R4=2 rather than incrementing it.

1.3.6 Relators

The relators available for use in conditions are:

=	equal
#	not equal
<=	less than or equal
>	greater than
>=	greater than or equal
<	less than

1.3.7 Labels

Any statement may be labelled by preceding it by a tag terminated by a colon. More than one label may be placed at the same point in the program if required.

1.3.8 Comments

Any statement may be annotated by following it by a comment. A comment starts with an oblique stroke, which must be preceded by a space unless the comment is at the start of a line, and continues to the end of the line -- that is, a semi-colon does not terminate a comment.

1.3.9 Spaces

In general space symbols may appear freely in statements and are not significant. Exceptions to this rule are:

(a) there must be no spaces within a tag or numeral, nor between a tag used as a label and the following colon, nor between a tag introducing an indexing expression and the following left parenthesis;

(b) at least one space must appear between an instruction mnemonic, macro name, directive name or keyword and a following operand, and between the body of a statement and an attached comment;

(c) spaces are significant within character constants.

2.0 STATEMENTS

2.1 Assignment statements

form

```
Assignment-statement -> term '=' expression ('IF' condition)?
  where term         -> tag,  tag '(' expression ')'
  and   expression   -> exp1,  expression binary-operator exp2
  and   exp1         -> exp2,  unary-operator exp2
  and   exp2         -> item,  '(' expression ')'
  and   item         -> term,  '#' term,  literal
  and   condition    -> cond1, conjunction, disjunction
  and   conjunction  -> cond1 'AND' (cond1, conjunction)
  and   disjunction  -> cond1 'OR' (cond1, disjunction)
  and   cond1        -> expression (relator expression)?
```

examples

```
R2 = R1                     R2 = R1 IF R1 > R2
R4 = R4+NUM                 COUNT = 0 IF COUNT = 1
NUM = MAX-MIN               FLAG = 1 IF SYM < '0' OR SYM > '9'
R4 = #ENTRY                 R3 = -1 IF I = 0 AND J <= K
W(P) = W(P+2)&X'FFF0'       NUM = NUM-1 IF \8
SYM = B(POS)
TYPE = STYPE(K-32)
```

effect

The assignment statement is the principal instruction of the language. Each assignment instruction is translated into a sequence of basic instructions whose effect at run-time is to evaluate the right-hand expression and assign its value to the register or location denoted by the left-hand term. All arithmetic is in terms of 16-bit integers; floating-point operations require the use of machine instructions.

In the case where the assignment instruction has a conditional clause appended to it, code is generated to test the condition and cause a branch over the code for the instruction body in the event that the condition is false. Comparisons in conditions are interpreted algebraically. The form of condition consisting of a single expression indicates an explicit test on the condition-code (as set, for example, by a machine instruction). The value of the expression in this case must be a legitimate mask value (to test true) or the logical complement of a mask value (to test false).

The form of term in which the tag is followed by a parenthesised expression indicates indexing in the IBM 360 sense. The special forms W( ) and B( ) indicate reference to the word (W) or byte (B) addressed by the parenthesised expression. The form of item in which the term is preceded by the symbol '#' denotes the address of the specified storage location.

All operators occurring in expressions have equal priority and the rule of left association applies to expressions containing more than one operator. Parentheses may be used to over-ride this. The implementation of assignment instructions, jump instructions and run-time conditions may require the use of one or more temporary registers; the $TEMP directive (3.4) is used to specify to the assembler which registers are available for use in this way. In order to permit optimisation of temporary register usage over sequences of consecutive instructions, there should be no entry-points in such sequences which are not marked by a label.

2.2 Jump statements

form

```
Jump statement  ->  'JUMP' term ('IF' condition)?
```

examples

```
JUMP L1                      JUMP EOF IF SYM = EOT
JUMP W(SBASE+K)              JUMPS L5 IF IND > MAX
JUMP W(R3)                   JUMP OUT IF 8
                             JUMP OUT IF \3
```

effect

Each jump statement is translated into a sequence of machine instructions whose effect at run-time is to cause control to pass to the location designated by the term, either unconditionally (case without conditional clause) or conditionally (case with conditional clause).

In the case of the Interdata 70 it is advantageous to utilise the short form of machine branch instruction if the destination of a jump is not more than 15 words from the location of the jump. The assembler normally does this automatically for backward- referring jumps, but not for forward jumps (nor for backward jumps occurring in the body of conditions and loops). In the latter case the programmer may indicate that a short jump should be used by writing JUMPS in place of JUMP. A listing option is available which causes the assembler to flag cases where JUMP could be replaced by JUMPS.

2.3 Machine-code statements

form

```
Machine-code-statement  ->  mcode-instruction ('IF' condition)?
    mcode-instruction   ->  inst-mnemonic expression-list
    expression-list     ->  (expression  (',' expression)* )?
```

examples

```
MH R2,COUNT             SVC 13,0 IF CODE = 1
BAL R6,SKIP             DH R2,R4 IF R4 # 0
ACH R2,TAB(R8)          CALL SKIP IF SYM = ' ' OR SYM = NL
AHM R3,W(P-4)           CALL SKIP
CLH R3,WL               READSYM
CLH R3,R4
CLH R3,X'FF00'
THI R3,PMASK\QBIT
LPSW OLDPSW
```

(See also section 3.2 for an explanation of tags as used in the examples.)

effect

Each machine-code instruction is translated into a single basic instruction occupying either one or two 16-bit words according to type. The treatment of conditions is as for assignment statements (2.1).

The number and types of the operand expressions must be consistent with the requirements of the machine instruction. A list of pre-defined instruction mnemonics with associated operand types is given in Appendix B. The method of defining instruction mnemonics with partially specified operands is illustrated in 3.2.

2.4 Conditional bracketing statements

form

```
'IF' condition                      'IF' condition
  [conditional sequence]              [conditional sequence]
'ELSE'                              'FINISH'
  [alternative sequence]
'FINISH'
```

examples

```
IF T4 = BASE+I                      IF CODE > MIN
  T3 = T3+1                           CALL FAULT
ELSE                                  CODE = 0
  T3 = T4                           FINISH
  COUNT = COUNT&(\1)
FINISH
```

effect

An IF clause is translated into a number of instructions whose effect at run-time is to cause a branch over the conditional instruction sequence in the event that the condition is false. The branch is to the start of the alternative instruction sequence (case with ELSE) or the instruction following the associated FINISH (case without ELSE). The effect of the ELSE statement is to cause a branch to be generated over the alternative sequence. The FINISH statement marks the end of the condition structure; no code is generated.

Explicit labels and tag definitions are not permitted within conditional structures.

The following abbreviated forms are also permitted;

```
ELSE
  IF condition                                ELSE IF condition
     [conditional sequence]   may be             [conditional sequence]
  ELSE                        written         ELSE
     [alternative sequence]   as                 [alternative sequence]
  FINISH                                      FINISH
FINISH

ELSE
  IF condition                may be          ELSE IF condition
     [conditional sequence]   written            [conditional sequence]
  FINISH                      as              FINISH
FINISH
```

example

```
IF SYM ='B'
   JUMP BUFFIN
ELSE IF SYM = 'C'
   JUMP CYCIN
ELSE IF SYM = '1'
   JUMP INTRP
ELSE
   JUMP ERR
FINISH
```

2.5 Loop bracketing statements

form

```
'WHILE' condition              'CYCLE'
  [loop sequence]                [loop sequence]
'REPEAT' ('IF' condition)?     'REPEAT' ('IF' condition)?
```

examples

```
WHILE B(P) # NL                CYCLE
  COUNT = COUNT+1                W(I) = 0
  P = P+1                        I = I-1
REPEAT                         REPEAT IF I >= MIN
```

effect

A WHILE or CYCLE statement marks the start of a loop. For the case with WHILE, code is generated to test the condition and branch out of the loop in the event that the condition is false; for the case with CYCLE, no code is generated. The corresponding REPEAT statement marks the end of the loop; a branch to the start of the loop is generated, conditionally in the case with conditional clause.

Explicit labels and tag definitions are not permitted within loop structures. Loop structures and conditional structures must be properly nested within each other.

2.6 Data statements

form

```
Data-statement  ->  expression-list
```

examples

```
0
1, -1, X'1FFF', 1, X'0101'
#ENTRY, #ENTRY+4
#L1-#LBASE, #L2-#LBASE, #L3-#LBASE
I<<12+(J<<6)+K
B 10, 20, 30, 40, 50, 60, 70
B 'FAULT 9'
16 $ 1234
B X'1000'-* $ 0
```

effect

Data statements permit the planting of (initial) data values in line in the object code. All expressions appearing in data statements must be capable of assembly-time evaluation, yielding an index-free -- but possibly relocatable -- result. In an expression operands must be constants, absolute addresses or a single relocatable component. Differences of relocatable components is a possible form, but each must be within the scope of the same relocation directive. Each expression is evaluated as a 16-bit value and, except in byte mode, generates a single 16-bit word. Byte mode is indicated by prefixing a B to the expression-list.

A quoted text string in a data statement is treated as a sequence of single-character constants; for example, 'FAULT 9' is equivalent to 'F', 'A', 'U', 'L', 'T', ' ', '9'. A data value may be repeated by preceding the value with a literal repetition count and a dollar sign.

A data statement may continue onto a following line if it ends with a comma. It still retains the mode of the previous line but note that the inclusion of a comment or label is illegal. For example:

```
  B 1,2,
    3,4
```

continues in byte mode, while

```
  B 1,2,    /BYTES
    3,4
```

and

```
    1,2,
LAB: 3,4
```

are illegal.

2.7 Macro call statements

form

```
Macro-call-statement  ->  macro-call ('IF' condition)?
    macro-call        ->  macro-tag argument-list
```

examples

```
SWOP SYM1,SYM2          SWOP NUM,COUNT IF NUM > COUNT
```

effect

The effect of a macro call is to cause the specified macro body to be assembled with the arguments supplied in the call substituted for the dummy arguments of the macro definition (call by substitution). For the example given in 3.7, SWOP SYM1,SYM2 would expand to R1 = SYM1; SYM1 = SYM2; SYM2 = R1.

If it is required to include spaces, commas or semi-colons in an argument, the argument should be enclosed in square brackets. Outer square brackets enclosing an argument are removed when it is substituted into the macro body.
(See also Section 3.7 page 19 on Macro Definitions).

3.0 ASSEMBLER DIRECTIVES

Assembler directives are introduced by a keyword preceded by a dollar sign. Only the first three letters of the keyword are significant.

3.1 Block structure

form

```
'$BEGIN'   comment text?
  [block-body]
'$END'     comment text?
```

effect

The $BEGIN directive marks the textual start of a block and the corresponding $END directive marks the textual end. New tags introduced within the block body are local to the block and cannot be accessed from outside. The effect of $TEMP directives is also confined to the block in which they occur. Run-time condition structures must be properly nested within a block but assembly-time conditions need not be. Note also that a new page is started if comment text is present on a '$BEGIN'.

3.2 Tag definitions

form

```
Tag-definition        -> '$DEFINE'  definition-list,
                         '$REDEF'   definition-list,
                         '$LABEL'     tag list
   definition         ->  tag '=' expression
```

examples

```
$DEFINE DEL=X'3F', RT=13, NL=10
$DEFINE I=R3, ALIM=ABASE+ASIZE
$REDEF MCOUNT=MCOUNT+1
$DEFINE READSYM=SVC 8,0
$REDEF OUT=W(R4), CALL=BAL R4
$DEFINE BASE=W(R2), WL1=W(R1+2), SYM=B(R1+3)
$DEFINE WL2=W(#WL1+6)
$LABEL ENT1,ENT2
```

effect

The effect of a tag definition is to cause the tag on the left of the equals-sign to be defined -- or re-defined -- to have the type and value of the expression on the right-hand side, evaluated as an assembly-time expression. All tags occurring in the expression to the right of the equals-sign should have been defined prior to the definition statement.

A tag defined in a $DEFINE statement is declared locally to the current block and may not duplicate an existing tag in that block. A tag re-defined in a $REDEF statement must already be defined in the current block or a containing block.

The $LABEL statement permits the declaration of tags textually prior to their appearance as labels. One use of this facility is to make accessible in an outer block labels which identify entry-points in an embedded block.

The case of READSYM and CALL in the examples above illustrates the definition of new instruction mnemonics with partially (CALL) or completely (READSYM) specified operands. Such definitions must always appear as the last definition in a definition-list.

3.3 Location counter specification

form

```
'$LOC'  expression
'$ASSLOC'  expression
```

examples

```
$LOC X'1000'
$LOC *+16
$ASSLOC R4+DISP
```

effect

During the course of assembly, the assembler maintains a location counter to keep track of the (notional) address into which each instruction or data-value is to be loaded at run-time. The initial value of the location counter is relocatable zero and at any time it identifies the byte address for the next item of code. The main significance of the location counter is in relation to the definition of labels.

The $LOC directive both sets the location counter and causes a loader instruction to be generated; the $ASSLOC directive simply sets the location counter. The value of the expression in a $LOC directive may be absolute or relocatable, but must be index-free; the value of the expression in a $ASSLOC directive may additionally have an index-component.

At any point when the type of the location counter is changed by a $LOC or $ASSLOC directive, there must be no outstanding forward references.

The current value of the location counter may be referred to by the pseudo-tag '*'. In the case of an instruction the value is always the value at the start of the statement.

3.4 <u>Temporary register specification</u>

<u>form</u>

'$TEMP' register-name-list

<u>examples</u>

```
$TEMP R1,R3
$TEMP R5
$TEMP   (i.e. no registers)
```

<u>effect</u>

The $TEMP directive defines to the assembler which registers are available for use in generated code to hold intermediate results. The directive applies to all of the program from its occurrence until the end of the containing block or the occurrence of another $TEMP directive.

An error report is generated by the assembler if more registers are required than are available. The initial default is for no temporary registers.

3.5 Listing control

form

```
'$LIST' expression
```

examples

```
$LIST 5
$LIST -1
```

effect

The $LIST directive allows selective control over the program listing produced by the assembler. Various listing options are selected by the bit-pattern specified by the expression as follows:

1	print code overflow lines
2	print macro expansions
4	print unsatisfied conditional assembly sequences
8	flag short jumps

If the value specified is negative, listing is suppressed altogether. The initial setting is $LIST 5. It is usually easier to diagnose assembly errors affecting macro calls if the option to print macro expansions is specified. The assembler starts a new page in the listing on encountering

(1) a $BEGIN with comment text,
(2) a comment introduced by '$/' rather than '/' or
(3) a blank line near the end of a page.

Note also that the current listing value may be referred to by the pseudo-tag '*L'.

3.6 Conditional assembly statements

form

```
'$IF'  condition                   '$IF'  condition
  [conditional sequence]             [conditional sequence]
'$ELSE'                            '$FINISH'
  [alternative sequence]
'$FINISH'
```

examples

```
$IF DISK = 1                   $IF LP # 0
   R3 = DDEV                     $DEF LPCOLS=80
$ELSE                            LPCB: 0; 0; X'62'
   R3 = 0                      $FINISH
$FINISH
```

effect

These statements provide for the conditional assembly of sections of a program, typically to provide for flexible macro expansion or to permit the generation of different versions of a program from a single source file and a number of short definition files.

If the condition in a $IF statement is found to be true, the following statements are assembled, otherwise they are skipped. Expressions occurring in the condition must be capable of assembly-time evaluation. The scope of a $IF statement extends to a matching $ELSE or $FINISH. For the case with $ELSE, the statements of the alternative sequence are skipped in the event that the statements following the $IF were assembled and conversely.

3.7 Macro definitions

form

```
Macro-definition  -> '$MACRO'  tag  dummy-argument-list
  dummy-argument  ->  tag,  tag '[' default-argument ']'
```

example

```
$MACRO SWOP A,B
 R1 = A; A = B; B = R1
$END
```

effect

The $MACRO statement introduces a macro-definition. The tag (SWOP in the example) is declared as the macro name. The following list specifies the names of the formal parameters of the macro (dummy arguments). The $MACRO statement is followed by the macro body which is terminated by the $END statement.

Text enclosed in square brackets following a dummy name is used as a default argument in the case of a partially specified macro call. Square brakets may also be used to substitue text containing non-alphabetic characters. (See section 2.7 Macro Calls).

Macro definitions may not be nested, but multi-level and recursive calls are permitted.

The use of the same termination statement for macro definitions as for blocks prevents a macro from terminating a block or the complete program. Macro definitions may, however, contain $BEGIN directives.

It should be borne in mind when writing macros that the simple tags representing the arguments might be replaced by arbitrarily complex expressions when the macro body is expanded, and the effects on, for example, association of operators should be considered.

To provide for the generation of unique tags when a macro is expanded a tag preceded by a question mark can be used. The tag is then incremented by redefinition. The assembler uses the fact that the internal codes are dense, in the sense that incrementing or decrementing an internal code will always produce another distinct valid internal code. For example:

```
$DEF A=.ABC
 ?A:              /GIVES ABC:
$REDEF A = A+1
 ?A:              /GIVES ABD:
```

3.8 SAVE statements

form

'$SAVE' expression

example

$SAVE 0

effect

In a number of applications it is convenient to be able to access saved registers using the mnemonics defined for the values when held in registers. A $SAVE statement both enables the use of register mnemonics as the fixed elements of indexing expressions as for example, Rl(P) and establishes the interpretation of such occurrences as the value of the literal expression specified plus twice the register number.

3.9 Program end statement

A $END directive which does not match an earlier $BEGIN or $MACRO marks the end of the complete program.

APPENDIX A ERROR REPORTS

Erroneous lines are flagged with an error letter in the assembly listing and are also printed on the report stream.

A	assembly-time evaluation of expression not possible
B	illegal byte reference
C	statement out of context
D	duplicate tag
F	form of statement incorrect
H	error in hexadecimal constant
I	illegal operand tag
J	JUMPS used inappropriately
P	phase error
Q	outstanding forward references at $LOC or $ASSLOC
R	no free register
S	JUMP can be made short (not an error)
T	truncation error in byte data value
U	undefined tag

APPENDIX B MACHINE INSTRUCTION MNEMONICS

tag	formats	tag	formats
?ABL	R,M	*OC	R,R / R,M
ACH	R,R / R,M	> OH	R,R / R,M / R,I/
>?AE	R,R / R,M	*RB	R,R / R,M
> AH	R,R / R,M / R,I / R,L	?RBL	R,M
AHM	R,M	*RD	R,R / R,M
*AI	R,R / R,M	*RH	R,R / R,M
*AL	M	RLL	R,I
?ATL	R,M	RRL	R,I
#BAL	R,M	?RTL	R,M
BFBS	L,L	SCH	R,R / R,M
>#BFC	L,M	>?SE	R,R / R,M
BFFS	L,L	> SH	R,R / R,M / R,I / R,L
BTBS	L,L	*SINT	R,R
>#BTC	L,M	SLA	R,I
BTFS	L,L	SLHA	R,I
BXH	R,M	SLHL	R,I
BXLE	R,M	SLL	R,I
>?CE	R,R / R,M	SLLS	R,L
> CH	R,R / R,M / R,I	SRA	R,I
CLB	R,M	SRHA	R,I
CLH	R,R / R,M / R,I	SRHL	R,I
>?DE	R,R / R,M	SRL	R,I
DH	R,R / R,M	SRLS	R,L
*EPSR	R,R	*SS	R,R / R,M
EXBR	R,R	STB	R,R / R,M
LB	R,R / R,M	>?STE	R,M
>?LE	R,R / R,M	STH	R,M
> LH	R,R / R,M / R,I / R,L	STM	R,M
LM	R,M	SVC	L,I
*LPSW	M	THI	R,I
>?ME	R,R / R,M	*WB	R,R / R,M
MH	R,R / R,M	*WD	R,R / R,M
MHU	R,R / R,M	*WH	R,R / R,M
> NH	R,R / R,M / R,I	> XH	R,R / R,M / R,I

R: register   M: mem-ref   I: immediate   L: short literal

? instruction not available on Interdata 74

* privileged instruction

register form automatically selected when appropriate

> instruction not included for space reasons. (See below)

Because of space limitations in the Assembler a number of the machine mnemonics listed in the above table have been omitted. The following list of the missing mnemonics is in the form of the definitions needed to restore them if required:

```
$DEF BTC=X'7010'_X'0408'
$DEF BFC=X'7010'_X'0303'
$DEF BTB=X'7010'_X'0408'
$DEF BTF=X'7010'_X'0508'
$DEF BFF=X'7010'_X'0608'
$DEF BFB=X'7010'_X'0708'
$DEF NH=X'7000'_X'0407'
$DEF OH=X'7000'_X'0607'
$DEF XH=X'7000'_X'0707'
$DEF LH=X'7000'_X'080F'
$DEF CH=X'7000'_X'0907'
$DEF AH=X'7000'_X'0A0F'
$DEF SH=X'7000'_X'0B0F'
$DEF STE=X'7000'_X'2002'
$DEF LE=X'7000'_X'2803'
$DEF CE=X'7000'_X'2903'
$DEF AE=X'7000'_X'2A03'
$DEF SE=X'7000'_X'2B03'
$DEF ME=X'7000'_X'2C03'
$DEF DE=X'7000'_X'2D03'
```

THE SSVC HALFWORD OP CODE

An additional instruction, SSVC (short SVC) is provided from within HAL for system programs. It takes two constants in the range 0-15 and assembles as the first word of a SVC with the two constants as the two register fields of the SVC instruction. No second half word is generated -- SVC is a 4 byte instruction. The intention is to extend the number of SVC's allowed, since many require no argument.

When the instruction is executed, the hardware processes it as an ordinary SVC, but with a meaningless argument. It is up to the called system procedure to deal correctly with the instruction and generate the appropriate return address. This may be achieved by decrementing the return PSW store location by two bytes.

For example, SSVC 14,10 assembles as X'E1EA'. The code to process this -- pointed at by location X'B8' -- would be something like:

```
SVC14:  STM R0,SAVE           /SAVE USER REGISTERS
        R1=W(X'98')           /GET RETURN ADDRESS
        W(X'98')=R1-2         /AND CORRECT IT
        R2=(W(R1-4)&15)<<1    /SSVC 14,?
        JUMP TAB(R2)          /INDEX ON SECOND CODE
  TAB: #S1400,#S1401,#S1402,#S1404

        ETC---
```

ASSEMBLER SEMANTICS

Reference has been made to the concept of expressions capable of assembly-time evaluation, for example in the context of machine-instructions, data-values and definitions. Such expressions are those which can be evaluated to denote any of the following by application of any of the standard operators and/or indexing:

<u>values</u>

an absolute value
    (16 bits)
a relocatable value
    (an absolute value plus an unknown constant)
an indexed value
    (an absolute value plus the contents of a register)
an indexed relocatable value
    (a relocatable value plus the contents of a register)

<u>store references</u>

a storage location addressed by any of the above types of value

<u>register references</u>

a register

In connection with the distinctions made above it should be noted that the location counter is a value while the operand for jump instructions must be a store reference. Thus the effect of placi g a label like LAB1 is the same as would be achieved by the definition $DEFINE LAB1=W(*), and proper forms of jump instructions are JUMP W(X'200'), JUMP W(*+4), JUMP W(R3) -- not JUMP X'200', JUMP *+4, JUMP R3.

HAL-70 has been implemented on the PDP-15 (in IMP) and on the Interdata 70 (in HAL-70). The object-code output by these assemblers as a binary file is in the standard Interdata loader format (M16/M17 version) and is suitable for loading by the Relocating Loader.

PDP-15

To assemble a HAL-70 program under the IMP15 operating system, give a command of the following form:

```
          .HAL70  <source> / <object> , <listing>
```

For example:

```
          .HAL70 TEST1/TEST1 ABS,LP
          .HAL70 TEST1/LK1(B),N
          .HAL70 TEST1/N,LP
```

At the start of assembly a pre-definition file is read on input stream 2 (default assignment: .HAL70 DEF). The source program is read twice in the course of assembly; the binary output and listing are produced on the second pass. Error reports are sent to the control output stream (normally the on-line console). The following disastrous errors may be reported:

Fault 21 insufficient space for macros and tag definitions
Fault 22 insufficient space for forward references
Fault 23 insuffcent space for block and conditional nesting
Fault 24,25 (internal error)

Interdata

To assemble a HAL-70 program under the ISYS operating system, give a command of the following form:

```
          HAL <source> (,<predef>)? / <object> , <listing>
```

For example:

```
          HAL TEST1/TEST1:B,LP
          HAL PROG,ISYSDEF/N
```

The pre-definition file may be used for standard definitions and macros. Disastrous errors are reported in the same way as in the IMP15 implementation. Error reports are sent to the on-line console.

```
                    /SVCS
                    $DEFINE NEXTSYM=SVC 8,0
                    $DEFINE READSYM=SVC 9,0

                    $MACRO SELIN SLOT          /SELECT INPUT
                    SVC 12,SLOT+SLOT
                    $END

                    $MACRO SELOUT SLOT         /SELECT OUTPUT
                    SVC 12,SLOT+SLOT+1
                    $END

                    /REGISTERS
                    $TEMP1 R1
                    $DEFINE SYM=R1             /*NB ASSEMBLER TEMP*
                    $DEFINE WORK1=R3, WORK2=R4, ENDED=R5
                    $DEFINE CI=R8,CODE=R9,TEXT=R10,NUM=R11
                    $DEFINE TI=R12, CHAIN=R13, TYPE=R14, INSYM=R15

                    $DEFINE STOPPER=-2000, MAXLIN=80, CBSIZE=180
                    $DEFINE CONTROL=0, REPORT=0     /STREAM NUMBERS

                    $MACRO VAR N
                    $DEFINE N=BASE
                    $REDEF BASE=W(#BASE+2)
                    $END
                    $DEFINE BASE=W(R2)
                    /*WORK SPACE LIMITED TO 384 BYTES*

                    VAR LIMIT
                    VAR CLIM
                    $DEFINE CBUFF=BASE         /COMMAND BUFFER START
                    $DEFINE TBUFF=W(#CBUFF+CBSIZE-1)

                    $MACRO ROUTINE NAM,R
                    $DEFINE NAM=BAL R,W(*)
                    $REDEF OUT=R
                    $END
                    $DEFINE OUT=0              /TO PERMIT RE-DEFINITION

                    /COMMAND INPUT ROUTINES

                    STYPE:
                    /       ! #     $%&'     ()*+     ,-./
0000' 0033 3233         X'0033',X'3233',X'A803',X'B233'
0004' A803 B233
                    /       0123     4567     89:;     <=>?
0008' 0000 0000         X'0000',X'0000',X'0031',X'3330'
000C' 0031 3330
                    /       @ABC     DEFG     HIJK     LMNO
0010' 3292 5749         X'3292',X'5749',X'2699',X'9722'
0014' 2699 9722
                    /       PQRS     TUVW     XYZ[     \]‾_
0018' 9296 5562         X'9296',X'5562',X'2223',X'9333'
001C' 2223 9333
```

```
                       ROUTINE RDITEM,WORK1
0020' C9E0 0001            JUMP OUT IF TYPE = 1          /TERMINATOR READ
0024' 0333
0026' 24E1                 TYPE = 1
                           CYCLE
0028' E190 0000              READSYM
002C' C910 0020              JUMP OUT IF SYM < 32        /CONTROL CHAR
0030' 0213
0032' 2235                 REPEAT IF SYM = ' '
0034' C910 0060            SYM = SYM-32 IF SYM >= 96     /MAP LOWER-CASE
0038' 2113
003A' CA10 FFE0
003E' 08F1                 INSYM = SYM
0040' 9011                 SYM = SYM>>1                  /*SETS CC*
0042' D3E1 FFF0'           TYPE = B(#STYPE(SYM-16))      /*PRESERVES CC*
0046' 2182                 TYPE = TYPE>>4 IF \8          /INSYM WAS ODD
0048' 90E4
004A' C4E0 000F            TYPE = TYPE&15
004E' 0233                 JUMP OUT IF TYPE # 0
0050' C8B0 F831            NUM = STOPPER+1
0054' C9F0 003F            JUMP OUT IF INSYM = '?'
0058' 0333
005A' 24B0                 NUM = 0
005C' C9F0 002A            JUMP OUT IF INSYM = '*'
0060' 0333
0062' C8BF FFD0            NUM = INSYM-'0'
                           CYCLE
0066' E180 0000              NEXTSYM
006A' C910 0030              JUMP OUT IF SYM < '0'
006E' 0213
0070' C910 0039              JUMP OUT IF SYM > '9'
0074' 0223
0076' 081B                   NUM = NUM<<2+NUM<<1         /NUM*10
0078' 9112
007A' 0AB1
007C' 91B1
007E' E190 0000              READSYM
0082' CAB1 FFD0              NUM = NUM+(SYM-'0')
0086' 4300 0066'           REPEAT

                       ROUTINE READCO,R6
008A' E1C0 0000            SELIN CONTROL
008E' E1C0 0001            SELOUT REPORT
0092' C882 0004        RC1:CI = #CBUFF; TI = #TBUFF
0096' C8C2 00B7
009A' 24D0                 CHAIN = 0; TYPE = 0
009C' 24E0
009E' 4130 0020'           RDITEM
00A2' 08EE                 IF TYPE = 0 AND CLIM # 0
00A4' 213D
00A6' 4812 0002
00AA' 233A
00AC' 40B1 FFFE              W(CLIM-2) = NUM
00B0' 4130 0020'             RDITEM
00B4' C9E0 0001              JUMP ER2 IF TYPE # 1
00B8' 4230 00BE'
00BC' 0306                   JUMP OUT
                           FINISH
                         . . . . . .
```