MPL COMPILER
REFERENCE MANUAL

Microdata 3200

MPL COMPILER REFERENCE MANUAL

August 1, 1975

Microdata Corporation
17481 Red Hill Avenue, Irvine, California 92714
Post Office Box 19501, Irvine, California 92713
Telephone: 714/540-6730 · TWX: 910-595-1764

OUTLINE

LIST OF FIGURES

LIST OF APPENDICIES

SPECIFICATIONS

● Language Implemented:	MPL
● Implementation Language:	MPL
● Host (Compile-Time) Machine:	Microdata 32/S
● Target (Run-Time) Machine:	Microdata 32/S
● Compile-Time Support:	Loader, I/O System
● Run-Time Support:	Loader, I/O System
● Translation Strategy:	One pass, top-down parse, in memory operation, hash addressing of symbols
● Memory:	64K Bytes
● Symbol Table (6 character names, 4 references per symbol, 8000 bytes total)	
Hash Table Size:	401 Symbols
Symbol Count, No References:	363 Symbols
Symbol Count, With References:	210 Symbols
● Speed:	1200 Records per minute (typical) when not I/O bound
● I/O Structure:	Source file, object file, listing file, console
● Translation Control:	24 toggles including listing selection, format control, and error control
● Diagnostics:	100 discursive diagnostics, 35 consistency checks, memory checksums, stack overflow protection

1.0 INTRODUCTION

The MPL 64K Self Compiler is a modern, high-performance, user-oriented language translator. Emphasis is placed on simplicity of operation and complete diagnosis of error conditions. This compiler is a *self* compiler in two senses: it is written in the same language, MPL, it translates, and it runs on the same machine, Microdata 32/S, for which it generates code.

The present reference manual describes a language translator for MPL: Microdata Programming Language. The MPL language is described in a companion language reference manual. The present MPL language is an outgrowth of an earlier MPL that is a subset of the current MPL. A cross compiler exists that implements the earlier MPL; the cross compiler runs on any machine that has an XPL implementation, in particular an IBM System/370. A complete bibliography of documents related to the MPL language and translators is shown in Appendix A.

This compiler reference manual is organized in 6 chapters and several appendicies. Chapter 1 is an introduction to the compiler and compiler reference manual. Chapter 2 discusses compiler operation. Chapter 3 describes compiler formats and is mainly useful for interpreting the listings. Chapter 4 discusses diagnostics and is useful when the on-line messages do not suffice. Chapter 5 specifies the translation algorithms used to convert source to object. Chapter 6 describes internal compiler operation and organization and is useful for compiler maintenance or enhancement. The appendicies provide a quick reference guide to the flags and toggles.

The term *compiler* is used to refer specifically to the subject of this present document: the MPL 64K Byte Self Compiler; the term *translator* is used generically as appropriate. The term *program* is used colloquially for an external procedure. Likewise, a *card* is a colloquialism for a source record. A program as input to the compiler is a *source* program, as output from the compiler (and used as input to the loader) an *object* program. A routine name of the compiler is written in capital italics (*ROUTINE*), a variable of the compiler in small italics (*variable*). In programming examples, keywords are written in upper case (KEYWORD) and variables in lower case (variable). The graphic 'b̸' is used to represent a blank.

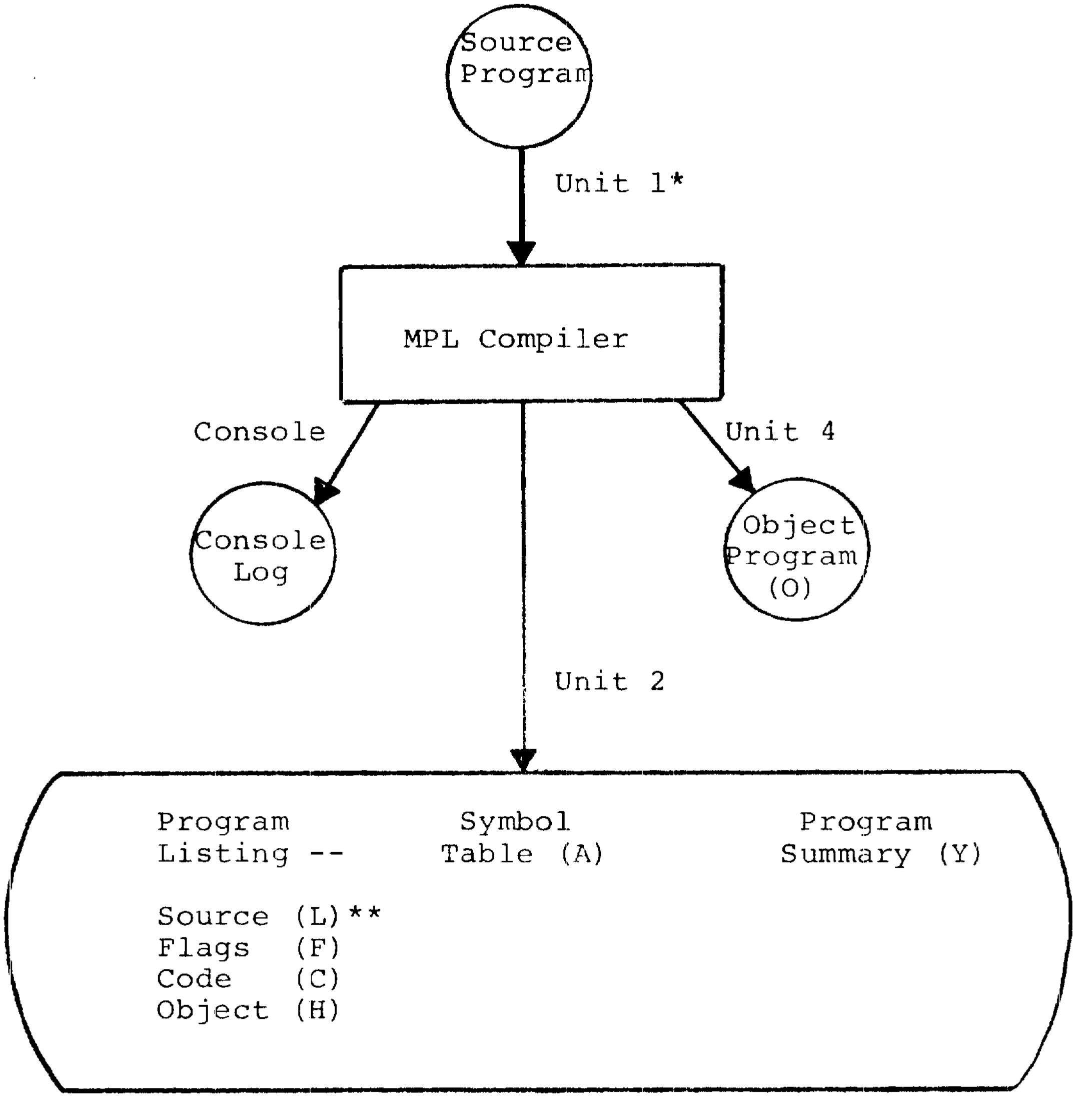
2.0 COMPILER OPERATION

This chapter discusses the various aspects of compiler operation. The compiler is controlled by means of toggles. A console log is maintained of compiler activity. Large or unusual programs may require attention to compiler memory use.

Operation of the MPL compiler is designed for convenience and simplicity. The compiler requires only one pass and operates in memory. No scratch peripherals are required, nor program overlays. In addition to a console device, up to 3 peripherals are required for source, object, and listing as shown in Figure 2.0a; the object and listing devices are necessary only if used

The compiler is invoked as appropriate to the operating system at hand. It may be loaded as any other program or invoked directly as a memory image.

Compilation speed is typically 1200 records per minute when not limited by I/O. Compilation rythmn is steady except for brief pauses to compute checksums at compiler start and program end, to assign addresses after a declaration set, and to sort the symbol table before it is listed.



* I/O system logical unit number
** Controlling toggle

Figure 2.0a: I/O File Structure

2.1 CONTROL TOGGLES

Control over compiler operation is through the mechanism of toggles. Toggles are internal to the source program; there is no external operating system mechanism for controlling compiler operation. Each toggle controls a single aspect of compiler operation and has a binary value: on or off.

The complete set of 24 toggles is listed in Appendix E. with their default values. A detailed explanation of the operation of each toggle is given in Appendix F.

All toggles assume a default value when the compiler is invoked. Toggles change during program compilation as specified by the user, or occasionally are altered by the compiler. At the start of subsequent programs toggles reassume their default values unless frozen by the P-toggle. The state of all toggles at the end of a program is displayed in the program summary (see Section 3.3.8).

Toggles may be altered by the user in any comment. Three characters are used as toggle control operators:

```
$ (dollar sign)  - If on set off, if off set on
& (ampersand)    - Set on
% (percent sign) - Set off
```

The character that immediately follows the toggle operator is interpreted as the toggle to be altered. Should this character not be an implemented toggle, compiler operation is not affected. Toggles that are sometimes altered by the compiler are the H-toggle, O-toggle, U-toggle, and X-toggle as described in Appendix F.

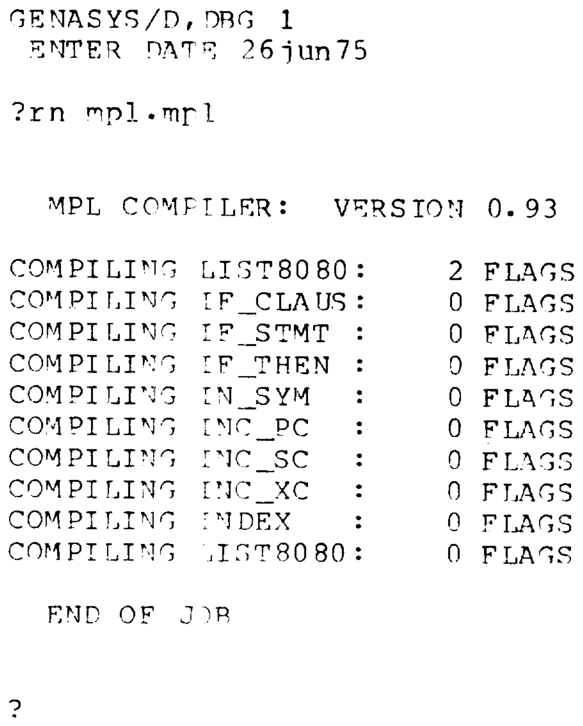
Toggle changes are honored by the compiler as soon as they occur. Since a listing line is buffered, toggles that affect the listing format become effective before the line in which they appear is listed. Since the compiler parse is top-down, toggles that affect object generation should appear before the external procedure entry. It is common practice to collect all toggles together as a preamble to the program proper.

2.2 CONSOLE LOG

During the operation of the compiler a console log is maintained on the console device to track compiler progress and report major errors. Figure 2.2a shows a typical console log.

The content and format of the console log are fixed and may not be suppressed or altered. The first entry on the log is a preamble that contains the version identification; the appearance of the preamble assures that the compiler has been invoked successfully. The last entry on the log is a postamble that is issued just before exit to the operating system; the appearance of the postamble assures that the compiler has terminated successfully.

The body of the console log consists of program entries, compiler aborts, and operating system messages. For each program the program name and flag count is shown; the name is listed as soon as encountered, the flag count is appended when the program ends. A compiler abort is listed on the log; all other flags are not. If the abort is suppressed, this is noted and compilation continues, otherwise end-of-job activity occurs. Operating system messages are also included in the console log as they occur.



```
GENASYS/D,DBG 1
 ENTER DATE 26jun75

?rn mpl.mpl


  MPL COMPILER:  VERSION 0.93

COMPILING LIST8080:     2 FLAGS
COMPILING IF_CLAUS:     0 FLAGS
COMPILING IF_STMT :     0 FLAGS
COMPILING IF_THEN :     0 FLAGS
COMPILING IN_SYM  :     0 FLAGS
COMPILING INC_PC  :     0 FLAGS
COMPILING INC_SC  :     0 FLAGS
COMPILING INC_XC  :     0 FLAGS
COMPILING INDEX   :     0 FLAGS
COMPILING LIST8080:     0 FLAGS

  END OF JOB


?
```

Figure 2.2a: Sample Console Log

2.3 MEMORY USE

Compiler memory use is of concern only with very large or unusual programs. The compiler is designed to handle ordinary programs of modest size without strain. A large program may exceed the available symbol table space, a pathological program the available stack space. Stack and symbol table use is monitored and overflow causes a compiler abort.

A fixed portion of memory is devoted to the operating system, the program space of the compiler, and the compiler's stack. The remainder of memory is used for the symbol table. The use of memory and the memory cost of various compiler features is shown in Figure 2.3a. Where feasible, memory is shared and reclaimed so that, in ordinary programs, memory is exhausted only by symbol table overflow.

Should the stack overflow, Figure 2.3a can be used to judge how to reorganize the program to fit. Usually the overflow is precipated by an excessively complicated expression and can be corrected by breaking the offending expression into simpler components.

Should the symbol table overflow, references may be suppressed with the R-toggle. This usually doubles the effective symbol table size. Ultimately, the symbol table is completely full and the only remedy possible is to break the offending program into smaller segments. It is considered good programming practice to separate a large program into several smaller functional modules.

The compiler and operating system require 64K bytes.

Type	Item	Controlling Variable	Limit	Unit Cost
Stack Pool		#stacksize	6000 bytes	
	Procedure block nesting		(about 5000 is overhead)	48 bytes
	Begin block nesting			72
	Do nesting			58
	Expression nesting			106
	If statement nesting			20
	If expression nesting			30
Symbol Table Pool		None: Uses remainder of available memory	8000 bytes (typical)	
	Attributes			16 bytes
	Name text			1 per character
	Symbol reference			4
Forward Pool		#fwdmax	160 items	
	Label			5 bytes
	Entry			5
	Initialized variable			5
Individual				
	Literally nesting	#maxlit	10 items	8 bytes
	Block nesting	#maxlex	15	6
	Do nesting	#domax	20	2
	Do cases	#casemax	300	2
	Hash table	#hashsize	401	2

Figure 2.3a: Compiler Limits and Memory Costs

3.0 COMPILER FORMATS

The source, object, and listing formats are described in this chapter. Figure 3.0a shows the I/O record structure.

Function	Logical Unit	Record Format	Maximum Record Length	Block Size
Source	1	Variable	80	SYSIN
Listing	2	Variable	120	SYSOUT
Object	4	Fixed	80	80

Figure 3.0a: I/O Record Structure

3.1 SOURCE FORMAT

The source program compiled is read from logical unit 1. This logical unit is used by the operating system as SYSIN and is typically blocked, double buffered, and open. Figure 3.1a shows a sample source program.

Input is streamed: that is record boundaries have no syntactic significance. However, each record is listed on a separate line in the program listing. Input records may be up to 80 bytes in length and may be of variable or fixed length. If variable length records are used, trailing blanks are appended by the compiler to produce an 80 byte record.

Source may also come from the symbol table via literallys. Literallys may contain arbitrary text, may be nested, and follow the same scoping rules as other symbols.

When a program ends any text remaining in a active literally or a partially used source record is discarded. The next program begins with the next source record.

An end-of-file in the source stream terminates compilation. All output buffers are flushed and exit is made to the operating system.

```
LIST8080;  MAIN PROC;   DCL               /***   SYSIN TO SYSOUT 80/80 LIST     ***/

        SYSGET          EXT PROC WORD,
        SYSPUT          EXT PROC,
        K               WORD,
        BUF(80)         BYTE,
        CRLF            WORD INITIAL ("0D0A");
                                              /* READ SYSIN UNTIL EOF         */
        DO WHILE  SYSGET(@BUF(1), 80) > 0;
             DO  K=80  TO 1  BY -1;           /* TRIM TRAILING BLANKS         */
                  IF  BUF(K) ↑= ' '
                       THEN GO TO WRITE;
             END;
WRITE:;                                       /* WRITE SYSOUT, APPEND CRLF */
             CALL SYSPUT(@BUF(1), K);
             CALL SYSPUT(@CRLF, 2);
        END;
END LIST8080;
```

Figure 3.1a: Sample Source Program

Fig 3.1a

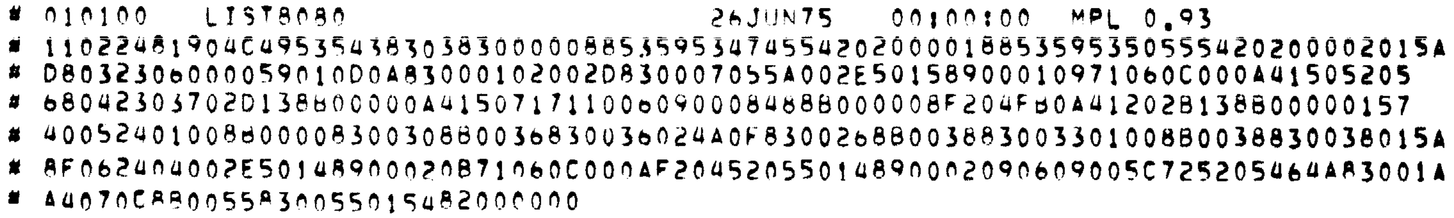
3.2 OBJECT FORMAT

The object program is written on logical unit 4. This logical unit has no special significance to the operating system. It is double buffered by the compiler and is unblocked. Object is written in response to the O-toggle.

The object record format uses the hexadecimal format defined in Section 5.9. A sample object program is shown in Figure 3.2a.

The first object record is used as a title card. Reading from left to right the title card contains:

- Object Record Header: This will always be '#b̸010100'.
- Program Name: The first 25 characters of the program name. Only the first 8 are passed in the body of the object program.
- Date/Time: The date and time at the start of the compilation of the current program.
- Version: The version identification of the compiler.



```
# 010100   LIST8080                        26JUN75   00:00:00   MPL 0.93
# 11022481904C4953543830383000008853595347455420200000188535953505554202000002015A
# D8032306000059010D0A83000102002D830007055A002E50158900010971060C000A41505205
# 68042303702D138B00000A415071711006090008488B000008F204FB0A41202B138B00000157
# 4005240100880000830030880036830036024A0F830026880038830033010088003883003 8015A
# 8F062404002E50148900020B71060C000AF204520550148900020906090 05C725205464A83001A
# A4070C880055830055015482000000
```

Figure 3.2a: Sample Object Program

Fig 3.2a

3.3 LISTING FORMATS

All listings are written on logical unit 2. This logical unit is used by the operating system as SYSOUT and is typically blocked, double buffered, and open.

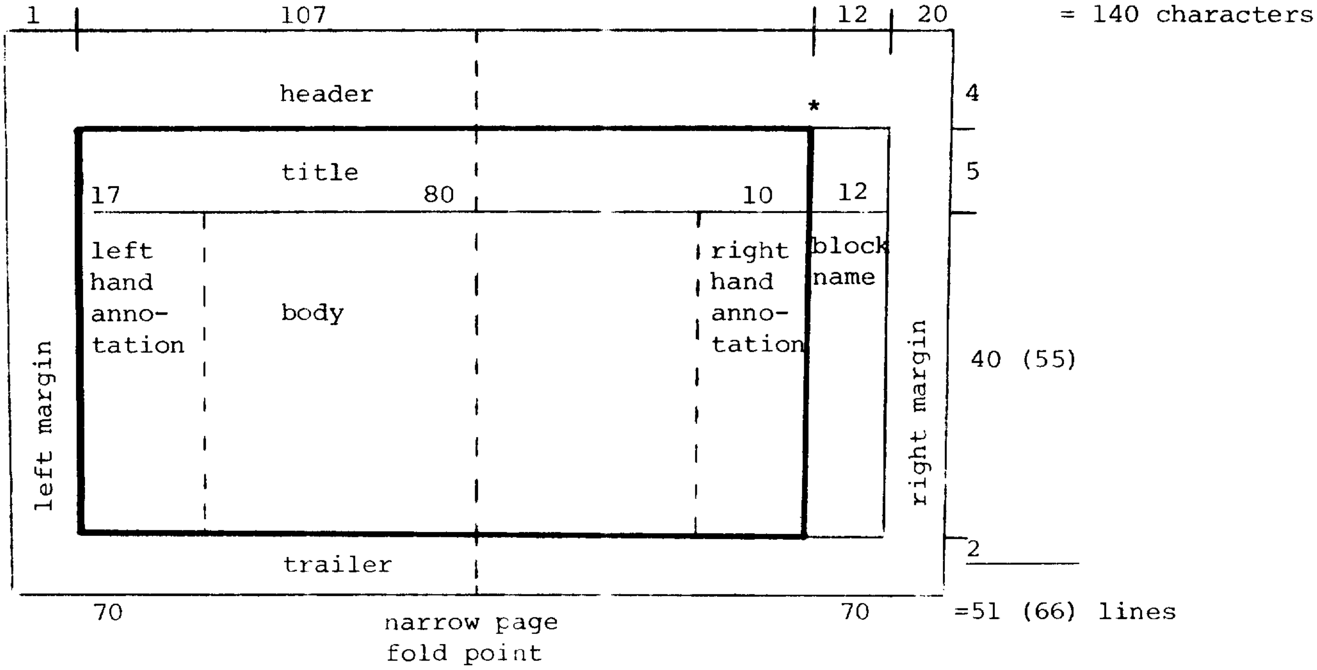
Listing output is streamed: that is line boundaries have no relationship to record boundaries. Line boundaries are specified by explicit carriage return/line feed sequences; page boundaries are specified by explicit form feeds or carriage return/line feeds, as selected by the E-toggle. Listing output may be spooled to a secondary storage device for later printing.

The listings consist of 3 major components: program, symbol table, and summary. The program listing consists of 4 subcomponents: source, flags, object, and code. All listings share a common page organization and title structure.

3.3.1 Page Organization

A listing page may be formatted for 11x12" sheets, 8½x12" sheets or 7" rolls. Figure 3.3.1a shows the listing page layout and the controlling toggles. Note that in the short page, full width format the listing may be copied directly to 8½x11" paper losing only the block name.

Fig 3.3.1a



Toggle	Description	State	Function

N	Format for
	narrow page
		Off
		On
			Width = 120 characters (12" @10 per inch)
			Width = 70 characters (7" @10 per inch)

Q	Chop program listing	Off On	List full line Suppress right hand annotation and blockname

S	Format for
	short page
		Off
		On
			Length = 66 lines (11" @6 per inch)
			Length = 51 lines (8½" @6 per inch)

*Area shown inside heavy lines is 8½ x 11"

Figure 3.3.1a: Listing Page Layout

3.3.2 Listing Titles

Each listing page contains a title. This title is common to all listing components. Each page also contains a subtitle peculiar to the listing components.

Reading from left to right the listing title consists of:

- Program Name: The first 8 characters of the program name. Until the program name is encountered, this field is blank. To ensure that the program name appears on the first title, the program name must appear on the first line.

- Data/Time: The date and time at the start of the compilation of the current program.

- Version: The version identification of the compiler.

- Listing Component: The component to which the page belongs: Program Listing, Symbol Table, or Program Summary.

- Page: The page number. The page number starts at 1 at the start of the current program and is common to all listing components.

3.3.3 Program Listing

The program listing consists primarily of an annotated source listing. Program listing is enabled by the L-toggle. If desired, flags, object, and code are interspersed as they are generated. Figure 3.3.3a shows a sample program listing with object and code suppressed, Figure 3.3.3b a sample program listing with flags, object, and code in evidence.

Where feasible, the annotation to the left of a source line reflects the state of the translation before the line is translated, the annotation to the right, the state afterwards. Exceptions to this rule are noted below.

Actual listing of a line is buffered one line. This buffering permits toggles to be processed before a line is listed. However, if a flag is generated, or code or object listed, the buffer is first flushed so that the source line always preceeds any derived listing; in such a case the righthand annotation reflects the translation state before line translation.

Reading from left to right the fields of the annotated source listing are:

- DEC: Program counter in decimal. In some cases the location listed points to an SSP instruction that precedes the statement proper.

- HEX: Program counter in hexidecimal.

- LINE: Source record number from start of current program.

- SOURCE: The source line padded on the right with blanks to 80 characters. Nonprinting characters (a character whose code is not in the range 32 to 126 inclusive) are listed as a period ('.').

- DL: Do level. The do level is defined as the nesting depth of all active blocks and groups. A block is a procedure block or begin block. A group is a repeat or any type of do.) The external procedure name is level 0. The do level is useful in reconciling mismatched ends, and in clarifying block and group structure.

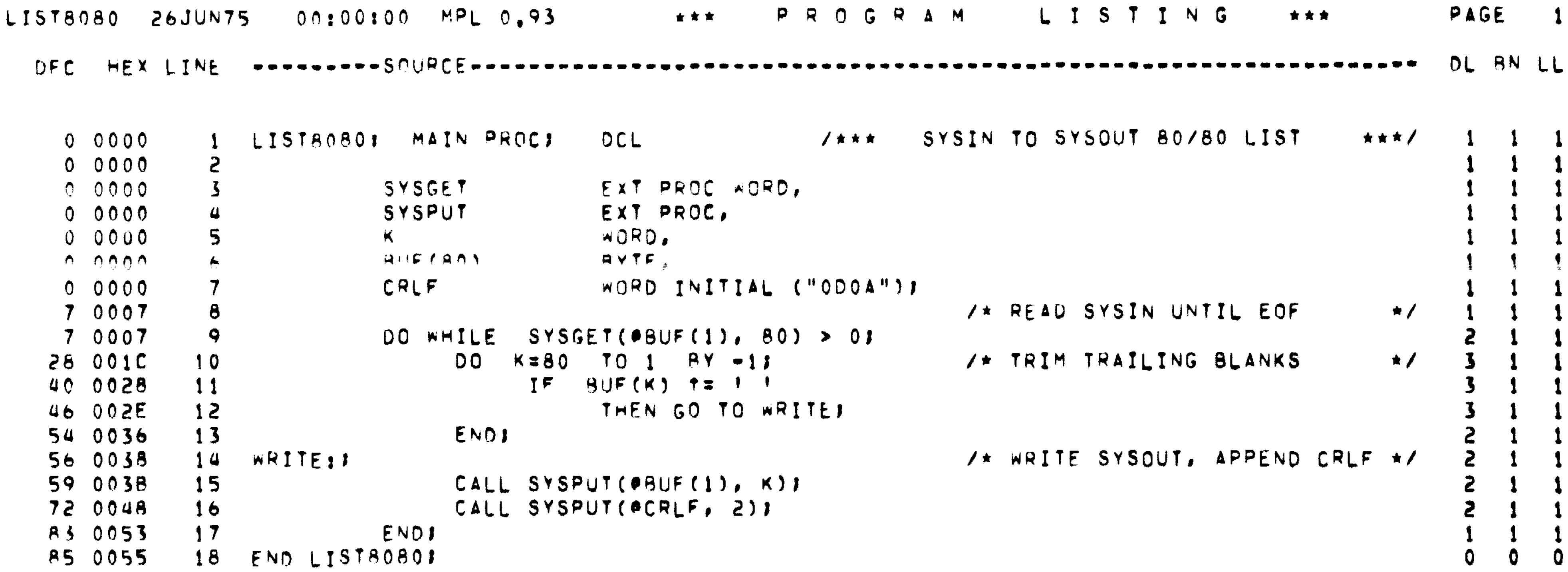
  If a do case is active the do level field is used to specify the do case item and do case nest level. Item and level are given for the translation state before line translation and are separated by a hyphen ('-').

- BN, Block Number: The block number is defined as the ordinal count of the most recent block entry. The external procedure name is block 0. The block number is useful in identifying block boundaries.

- LL, Lex Level: The lex level is defined as the nesting depth of all active blocks. The external procedure name is lex level 0. The lex level is useful in analyzing scoping restrictions.

- BLOCK: The block field gives the first 10 characters of the innermost active procedure block. A begin block or group does not alter the block field.

Fig 3.3.3a

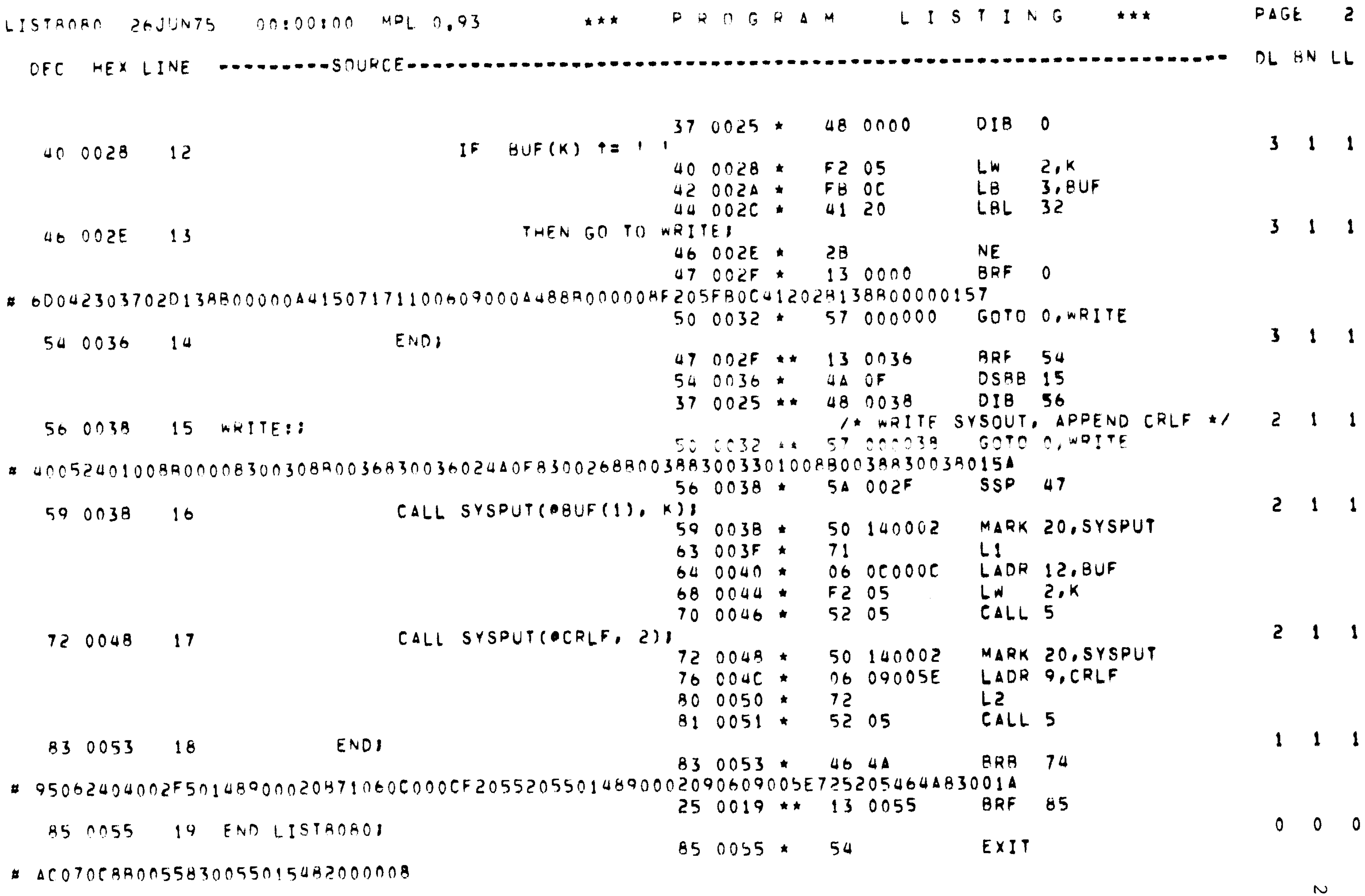


```
LIST8080  26JUN75   00:00:00  MPL 0.93         ***    P R O G R A M    L I S T I N G    ***        PAGE   1

 DEC  HEX LINE  ---------SOURCE-------------------------------------------------------------------  DL BN LL

   0 0000    1  LIST8080:  MAIN PROC;    DCL             /***   SYSIN TO SYSOUT 80/80 LIST    ***/   1  1  1
   0 0000    2                                                                                      1  1  1
   0 0000    3           SYSGET          EXT PROC WORD,                                             1  1  1
   0 0000    4           SYSPUT          EXT PROC,                                                  1  1  1
   0 0000    5           K               WORD,                                                      1  1  1
   0 0000    6           BUF(80)         BYTE,                                                      1  1  1
   0 0000    7           CRLF            WORD INITIAL ("0D0A");                                     1  1  1
   7 0007    8                                                  /* READ SYSIN UNTIL EOF       */    1  1  1
   7 0007    9           DO WHILE  SYSGET(@BUF(1), 80) > 0;                                         2  1  1
  28 001C   10                DO  K=80  TO 1  BY -1;             /* TRIM TRAILING BLANKS       */    3  1  1
  40 0028   11                     IF  BUF(K) ¬= ' '                                                3  1  1
  46 002E   12                          THEN GO TO WRITE;                                           3  1  1
  54 0036   13                END;                                                                  2  1  1
  56 0038   14  WRITE:;                                          /* WRITE SYSOUT, APPEND CRLF */    2  1  1
  59 003B   15                CALL SYSPUT(@BUF(1), K);                                              2  1  1
  72 0048   16                CALL SYSPUT(@CRLF, 2);                                                2  1  1
  83 0053   17           END;                                                                       1  1  1
  85 0055   18  END LIST8080;                                                                       0  0  0
```

Figure 3.3.3a: Sample Program Listing -- Listing Options Off

```
LIST8080  26JUN75   00:00:00  MPL 0,93        ***    P R O G R A M    L I S T I N G    ***        PAGE   1

  DEC  HEX LINE  ---------SOURCE-------------------------------------------------------------------  DL BN LL


   0 0000    1  LIST8080:  MAIN PROC;   DCL             /***   SYSIN TO SYSOUT 80/80 LIST    ***/   0  0  0
* 010100   LIST8080                   26JUN75   00:00:00  MPL 0,93
   0 0000    2                                                                                        1  1  1
   0 0000    3           SYSGET        EXT PROC WORD,                                                 1  1  1
   0 0000    4           SYSPUT        EXT PROC,                                                      1  1  1
   0 0000    5           JUNK          JUNK,                                                          1  1  1
                                          $
***** (0)  ERROR 67         BAD SIZE, WORD USED


                                          $
***** (5)  ERROR 59         JUNK IN DCL STMT, TEXT SKIPPED TO NEXT ',' OR ';'


   0 0000    6           K             WORD,                                                          1  1  1
   0 0000    7           BUF(80)       BYTE,                                                          1  1  1
   0 0000    8           CRLF          WORD INITIAL ("0D0A");                                         1  1  1
* 11022481904C4953543830383000008853595347455420200001885359535055420200002015A
                                                   0 0000 *   5A 0000     SSP  0
                                                   3 0003 *   59 01       FILL 1
                                                   5 0005 *      0D0A     WORD 3338
   7 0007    9                                                /* READ SYSIN UNTIL EOF      */   1  1  1
   7 0007   10           DO WHILE  SYSGET(@BUF(1), 80) > 0;                                           1  1  1
                                                   0 0000 **  5A 002E     SSP  46
                                                   7 0007 *   5A 002F     SSP  47
                                                  10 000A *   50 150001   MARK 21,SYSGET
                                                  14 000E *   71          L1
                                                  15 000F *   06 0C000C   LADR 12,BUF
                                                  19 0013 *   41 50       LBL  80
                                                  21 0015 *   52 05       CALL 5
* DC032306000059010D0A83000102002E830007055A002F5015890001097106 0C000C41505205
                                                  23 0017 *   70          L0
                                                  24 0018 *   2D          GT
                                                  25 0019 *   13 0000     BRF  0
  28 001C   11               DO  K=80  TO 1  BY -1;           /* TRIM TRAILING BLANKS      */   3  1  1
                                                  28 001C *   41 50       LBL  80
                                                  30 001E *   71          L1
                                                  31 001F *   71          L1
                                                  32 0020 *   10          NEG
                                                  33 0021 *   06 09000A   LADR 9,K
```

Fig 3.3.3b



```
LIST8080  26JUN75   00:00:00  MPL 0.93         ***    P R O G R A M     L I S T I N G    ***          PAGE   2

  DEC  HEX LINE  ---------SOURCE--------------------------------------------------------------------  DL BN LL

                                                           37 0025 *    48 0000      DIB  0
   40 0028    12                    IF  BUF(K) ↑= ' '                                                   3  1  1
                                                           40 0028 *    F2 05        LW   2,K
                                                           42 002A *    FB 0C        LB   3,BUF
                                                           44 002C *    41 20        LBL  32
   46 002E    13                        THEN GO TO WRITE;                                               3  1  1
                                                           46 002E *    2B           NE
                                                           47 002F *    13 0000      BRF  0
# 6D042303702D138B00000A415071711006090004488B000008F205FB0C41202B138B00000157
                                                           50 0032 *    57 000000    GOTO 0,WRITE
   54 0036    14                END;                                                                3  1  1
                                                           47 002F **   13 0036      BRF  54
                                                           54 0036 *    4A 0F        DSBB 15
                                                           37 0025 **   48 0038      DIB  56
   56 0038    15  WRITE:;                                               /* WRITE SYSOUT, APPEND CRLF */   2  1  1
                                                           50 0032 **   57 000038    GOTO 0,WRITE
# 40052401008B00008300308B0036830036024A0F8300268B003883003301008B0038830038015A
                                                           56 0038 *    5A 002F      SSP  47
   59 003B    16                CALL SYSPUT(@BUF(1), K);                                            2  1  1
                                                           59 003B *    50 140002    MARK 20,SYSPUT
                                                           63 003F *    71           L1
                                                           64 0040 *    06 0C000C    LADR 12,BUF
                                                           68 0044 *    F2 05        LW   2,K
                                                           70 0046 *    52 05        CALL 5
   72 0048    17                CALL SYSPUT(@CRLF, 2);                                              2  1  1
                                                           72 0048 *    50 140002    MARK 20,SYSPUT
                                                           76 004C *    06 09005E    LADR 9,CRLF
                                                           80 0050 *    72           L2
                                                           81 0051 *    52 05        CALL 5
   83 0053    18            END;                                                                    1  1  1
                                                           83 0053 *    46 4A        BRB  74
# 95062404002F50148900020B71060C000CF205520550148900020609005E725205464A83001A
                                                           25 0019 **   13 0055      BRF  85
   85 0055    19  END LIST8080;                                                                     0  0  0
                                                           85 0055 *    54           EXIT
# AC070C8B00558300550154B2000008
```

Figure 3.3.3b: Sample Program Listing -- Listing Options On

3.3.4 Flag Listing

Flags are interspersed in the program listing as they are detected. Flag listing is enabled by the F-toggle. If the F-toggle is on and the L-toggle (list source program) off the current source line is also listed. This feature is useful in scanning for errors. Figure 3.3.3b includes some flags.

Each flag generates 4 lines of listing: a cursor line, a description line, and 2 blank lines. The cursor indicates the point in the source line at which the flag was generated. Reading from left to right the description line consists of:

- Last Line: The last flagged line. If the current flag is the first flag this field is 0. The last line field simplifies flag location in a large listing.

- Severity: The severity of the flag: warning, error, blunder, or abort.

- Number: The code number of the flag. This code is an index to the flag descriptions in Appendix D.

- Description: A brief description of the cause for the flag and the compiler response. Cause and response are separated by a comma (',').

3.3.5 Object Listing

The object program is interspersed in the program listing as it is generated. Object listing is enabled by the H-toggle and is exactly the same as the object program written on logical unit 4 (see Section 3.2 Object Format). This listing is rarely activated. Figure 3.3.3b includes an object listing.

3.3.6 Code Listing

The translated 32/S instructions are interspersed in the program listing as they are generated. Code listing is enabled by the C-toggle. This listing is useful in low level debugging and for clarifying translation algorithms. For better readability, the code listing is indented 50 spaces in response to the I-toggle. Figure 3.3.3b includes a code listing.

Reading form left to right the code listing consists of:

- DEC: Program counter in decimal.
- HEX: Program counter in hexadecimal.
- Program Counter Type: This field distinguishes the 3 uses of the DEC/HEX program counter fields:

  blank - source
  * - generated in-line instruction
  ** - generated fixup instruction

- Numeric Operation Code: The 32/S instruction operation code in hexidecimal. In the case of data this field is blank.
- Numeric Operand: Up to 4 bytes of operand or data in hexidecimal.
- Symbolic Operation Code: The 32/S instruction operation mnemonic or pseudo-operation. The instruction mnemonics used are defined in Reference 6.
- Numeric Operand: The operation code dependent operand in decimal. This field is not always present.
- Symbolic Operand: The operation code dependent operand as a symbol. This field is not always present; if present it is separated from the numeric operand by a comma (',').

3.3.7 Symbol Table

A symbol table listing follows the program listing and is enabled by the A-toggle. Figure 3.3.7a shows a sample symbol table.

The symbol table has many uses. For new programs it is an aid to finding and eliminating flags. For old programs it aids in maintenance and documentation. In addition, the symbol table clarifies attribution and scoping.

Each symbol defined or referenced in the program is listed in the symbol table in alphabetical order. Literallys and builtins qualify as symbols, but keywords do not.

Each symbol is represented by a single symbol table entry with the occasional exception of forward labels. If a label is first encountered in a goto statement and is potentially satisfied in more than one block, then a separate entry is made for each block in which the label is potentially defined. The entry at the lowest lex level is the ultimate definition of the label and the references are scattered among the multiple entries. The generated code is always correct. For example, the following program results in two entries in the symbol table for the label x of which one will remain undefined:

```
p: PROC;
   GO x;
   q: PROC;
      GO x;
   END q;
x:
END p;
```

Reading from left to right the symbol table listing consists of:

- NAME: The first 12 characters of the symbol name. If the name has more than 12 characters, a plus sign ('+') is appended to the name.

- DEF: The source program record number on which the symbol is defined. If the symbol is never defined, 4 stars ('****') are displayed; except for built-ins this is an error. For procedures that acquire some of their attributes from a declare statement and some from a procedure statement, the declare statement is used to derive the definition line. This field may be viewed as the first element on the list of references.

- BN: The block number of the block in which the symbol is defined.

- LL: The lex level at which the symbol is defined.

- ST: The definition state of the symbol. If a symbol is completely defined this field is blank, otherwise 2 stars ('**') are inserted, except for forward labels that are potentially defined in more than one block, this is an error. Undefined symbols may be spotted by scanning this field for non-blank entries.

- DEC/HEX: The decimal and hexadecimal values of the symbol. The meaning of the value is conditioned by symbol class as described below.

- CLASS: The compiler division of symbol types. The possible classes are:

Class	Description	Use of Value Field
AUTO	Automatic variable	Stack location, EP relative
STATIC	Static variable	Internal: Static location External: External data sequence number
PARAMETER	Parameter	Stack location, EP relative
CONSTANT	Constant	Program location, PB relative
CBASE	Constant based variable	Base value, absolute
VBASE	Variable based variable	Symbol table address of base
PROCEDURE	Ordinary procedure	Internal: Program location, PB relative External: External procedure sequence number or program location of 0 if external procedure head
MAIN	Main procedure	
INTERRUPT	Interrupt procedure	
MICRO	Micro coded procedure	Microstore entry location
BUILTIN	Builtin procedure	Builtin code
LABEL	Ordinary label	Program location, PB relative
DOLABEL	Label on a group head	Program location, PB relative
BEGIN	Label on a begin block head	Program location, PB relative
LITERALLY	Literally	Length of text
??	Unclassified symbol (Appears only if job is aborted)	--

- SCOPE: The scope of a symbol:

Scope	Description
I	Internal
EXT	External

When not explicitly supplied, this attribute is always defined by the compiler according to the rules of the language.

- SIZE: The size of a symbol (specified only where meaningful):

Size	Description
BYTE	Byte
WORD	Word
DOUBLE	Double
PTR B	Pointer to byte
PTR W	Pointer to word
PTR D	Pointer to double
BIT 1	Bit (1)
BIT 2	Bit (2)
BIT 4	Bit (4)

- SET: Indicates if the symbol is preset before the procedure is executed:

Set	Description
YES	Symbol preset
Blank	Not preset or not meaningful

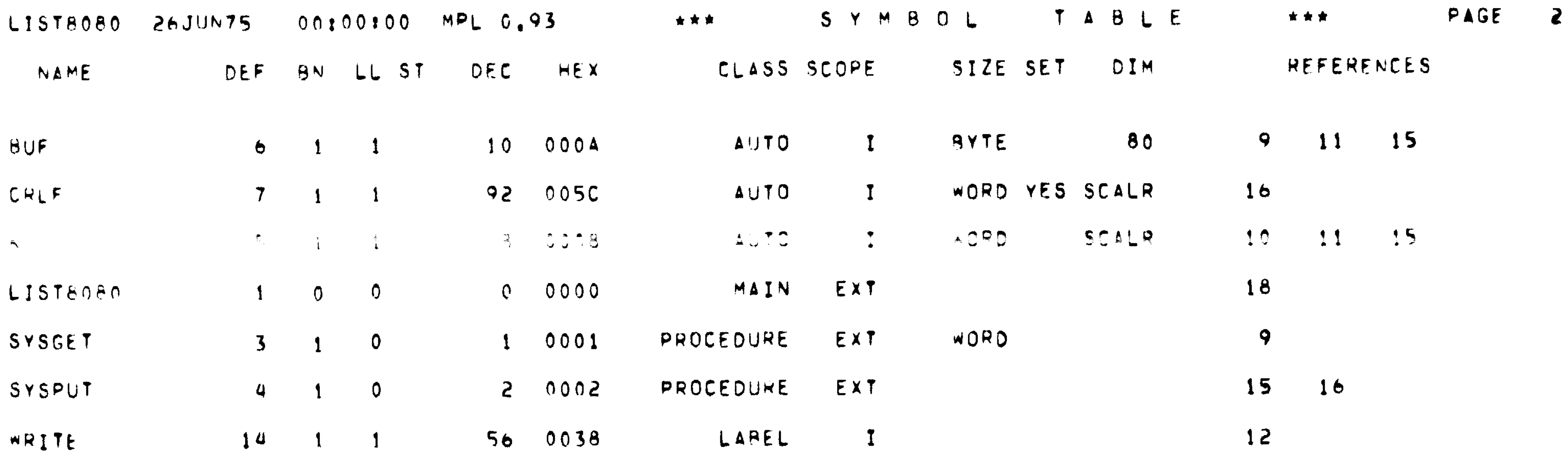
- DIM: The dimension of a symbol:

Dimension	Description
SCALR	Scalar variable
ARRAY	Array variable whose dimension was not retained
number	Array variable of specified dimension
blank	Not meaningful

- REFERENCES: The source program record numbers on which the symbol is referenced. The definition field may be viewed as the first item on this list. Possible entries are:

Reference	Description
number	Source program record number of reference
- NONE -	No references
SUPPRESSED	Some references may have been suppressed by the R-toggle

Fig 3.3.7a



```
LIST8080  26JUN75   00:00:00  MPL 0.93        ***       S Y M B O L    T A B L E       ***        PAGE   2

  NAME          DEF  BN  LL ST    DEC    HEX          CLASS SCOPE     SIZE SET   DIM          REFERENCES

BUF               6   1   1        10   000A           AUTO      I    BYTE        80       9    11    15

CRLF              7   1   1        92   005C           AUTO      I    WORD YES SCALR      16

K                 5   1   1         8   0008           AUTO      I    WORD     SCALR      10    11    15

LIST8080          1   0   0         0   0000           MAIN    EXT                        18

SYSGET            3   1   0         1   0001      PROCEDURE    EXT    WORD                 9

SYSPUT            4   1   0         2   0002      PROCEDURE    EXT                        15    16

WRITE            14   1   1        56   0038          LABEL      I                        12
```

Figure 3.3.7a: Sample Symbol Table

3.3.8 Program Summary

A program summary follows the symbol table listing and is enabled by the Y-toggle. Figure 3.3.8a shows a sample program summary.

The summary is useful in checking for program flags, summarizing the source and object programs, and optimizing compiler memory use.

The program summary consists of 7 parts: flags, source program, object program, symbol table, compiler stack, toggles, and flag link.

Flags --

- ABORTS: The number of compiler aborts. This number includes any aborts suppressed by the X-toggle and, hence, may be more than 1.
- BLUNDERS: The number of program blunders. If there are any blunders the object program (O-toggle) and object listing (H-toggle) will be turned off, unless suppressed by the D-toggle. There is no way to suppress blunders.
- ERRORS: The number of program errors. There is no way to suppress errors.
- WARNINGS: The number of program warnings. This number includes any warnings suppressed with the W-toggle.

Source Program --

- LINES: The number of lines (records) in the source program.
- STATEMENTS: The number of statements (delimiting semicolons (';')) in the source program.
- BLOCKS: The number of blocks in the source program.
- LEXDEPTH: The maximum lex depth acheived.

Object Program --

- BYTES PROGRAM: The number of bytes in the object program exclusive of stack requirements.
- BYTES STATIC: The number of static bytes required by the object program. This value is always rounded up to a word boundary and is the value passed to the loader.

- BYTES STACK(1): The maximum stack depth achieved by the outer procedure block. It is the sum of the automatic data allocation and the scratch required for instruction execution.

- OBJECT RECORDS: The number of object records even if object is suppressed by the O-toggle.

Symbol Table --

- BYTES USED: The number of bytes in the symbol table used.

- BYTES SPARE: The number of bytes in the symbol table unused.

- SYMBOLS: The number of symbols of all types.

- REFERENCES: The number of references to all symbols even if references are suppressed with the R-toggle.

  The sum of the BYTES USED and the BYTES SPARE is a constant. If symbol table space is a problem the symbol table summary can be used to determine corrective action. See Figure 2.3a for symbol table costs.

Compiler Stack --

- BYTES USED: The number of bytes in the compiler stack used.

- BYTES SPARE: The number of bytes in the compiler stack unused.

- ACCESSES: The number of primary accesses to the hash table (used to address the symbol table).

- COLLISIONS: The number of secondary accesses to the hash table due to hashing collisions.

  The sum of BYTES USED and BYTES SPARE is a constant. If stack space is a problem the compiler stack summary can be used to monitor stack use. See Figure 2.3a for stack costs. Hash table use can be monitored with the hash table summary. So long as symbol names are not highly pathological and the hash table is less than 90% full (Warning 39 has not been generated) the number of collisions should remain, at worst, a few times the number of accesses. If the number of collisions exceeds 10 times the number of accesses and there are no extenuating circumstances, compiler performance will start to degrade and the situation should be brought to the attention of Microdata.

Toggle Summary --

- TOGGLES OFF: All toggles that are off when the program ends, listed in alphabetical order.

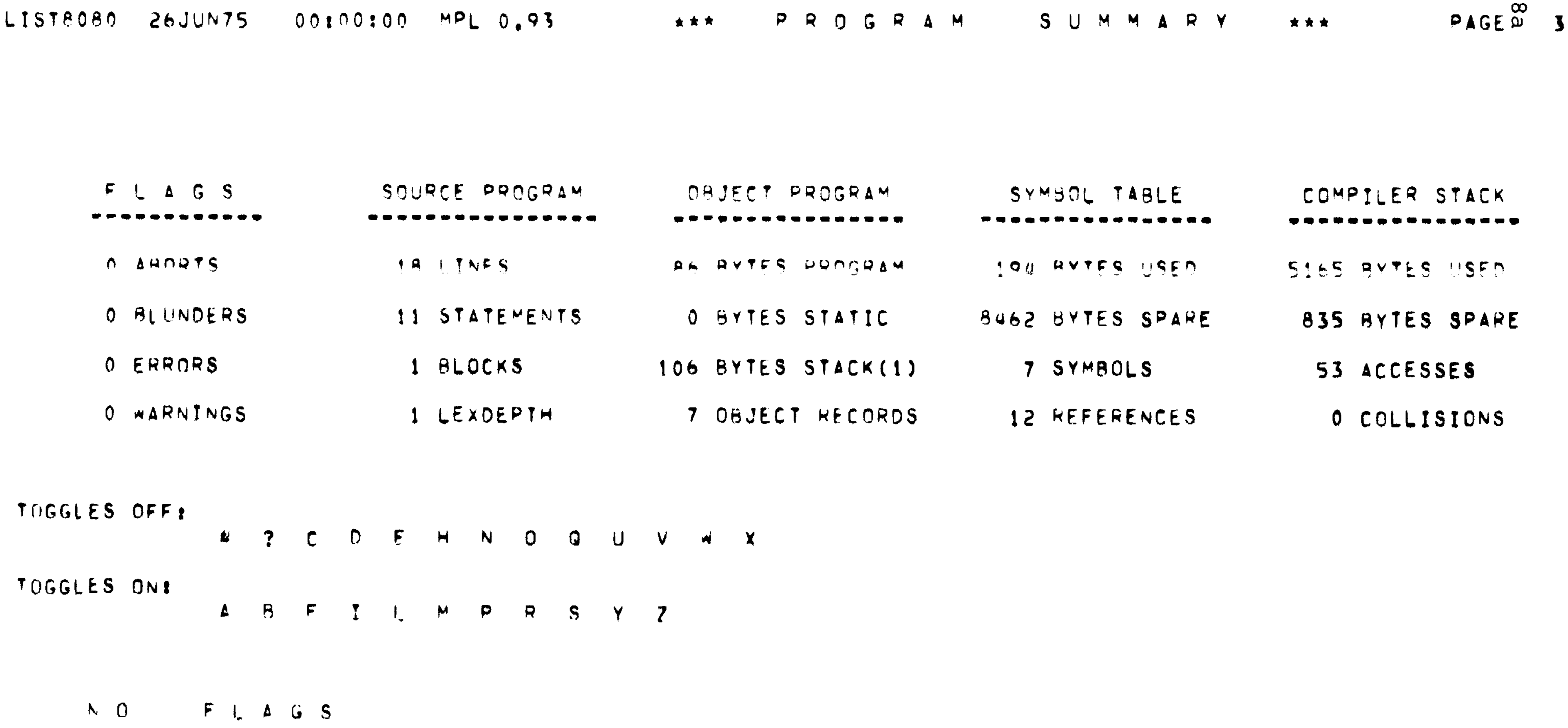
- TOGGLES ON: All toggles that are on when the program ends, listed in alphabetical order.

Flag Link --

The final item in the program summary is a link to the line number of the last flag, excluding suppressed warnings. If there are no such flags the message given is:

```
        N O   F L A G S
```

Fig 3.3.8a



```
LIST8080  26JUN75   00:00:00  MPL 0.93         ***    P R O G R A M    S U M M A R Y    ***         PAGE  3



      F L A G S             SOURCE PROGRAM        OBJECT PROGRAM         SYMBOL TABLE         COMPILER STACK
     -----------           ---------------       ---------------       ---------------       ---------------

      0 ABORTS               18 LINES            86 BYTES PROGRAM       190 BYTES USED       5165 BYTES USED

      0 BLUNDERS             11 STATEMENTS        0 BYTES STATIC       8462 BYTES SPARE       835 BYTES SPARE

      0 ERRORS                1 BLOCKS          106 BYTES STACK(1)        7 SYMBOLS            53 ACCESSES

      0 WARNINGS              1 LEXDEPTH          7 OBJECT RECORDS       12 REFERENCES          0 COLLISIONS


 TOGGLES OFF:
                @  ?  C  D  E  H  N  O  Q  U  V  W  X

 TOGGLES ON:
                A  B  F  I  L  M  P  R  S  Y  Z


    N O    F L A G S
```

Figure 3.3.8a: Sample Program Summary

4.0 DIAGNOSTICS

Error conditions that occur while the compiler is running fall in three areas: operating system errors, compiler errors, and hardware errors.

Operating system errors depend on the operating system in use and are discussed in detail in the appropriate operating system reference manual.

Compiler errors are either detected or undetected; detected errors are called flags which are in turn classified as warnings, errors, blunders, and aborts. Internal compiler operation is monitored for consistency and any fault reported as a foul. On occasion, some hardware problems may not be completely diagnosed.

4.1 FLAGS: DETECTED ERRORS

Whenever feasible, compilation errors are detected and an explicit diagnostic is issued in discursive form. The generic term flag is used to refer to detected errors which are classified in four levels according to their severity. The four flag classes, in order of increasing severity are: warnings, errors, blunders, and aborts. The attributes of the flag classes are shown in Figure 4.1a. The highest severity flag that occurs is noted in the object program for later job control by the operating system.

Warnings may be suppressed by means of the W-toggle to improve listing appearance in cases where the warnings are anticipated. The routine suppression of warnings is a dangerous practice.

An error results in a suspect object program, a blunder in a faulty object program. However, the program may still be executable depending on the circumstances. Once again, the routine use of object programs that contain errors or blunders is a dangerous practice. Object program generation is suppressed after a blunder unless object suppression is disabled by use of the D-toggle.

Compilation is usually discontinued after an abort; end-of-program and end-of-job activity is attempted and a return is made to the operating system. However, aborts may be suppressed (although the diagnostic is still issued) by use of the X-toggle; the compiler response to a suppressed abort is given in the description of each abort in Appendix D.

Flags are inserted in-line in the program listing as they occur. In addition, aborts are listed on the console. The flag listing format is described in Section 3.3.4. Flags are listed by number in Appendix B, by severity in Appendix C, and are described in detail in Appendix D.

Flag Class	Loader Code	Compilation Continuation	Source Program	Object Program	Compiler Integrity	Related Toggles
Warning	4	Yes	Suspect	OK	OK	W
Error	8	Yes	Bad	Suspect	OK	--
Blunder	12	Yes	Bad	Bad	OK	D,H,O
Abort	--	No	Suspect	Suspect	Suspect	X

Figure 4.1a: Flag Classes

4.2 UNDETECTED ERRORS

Some program errors go undetected by the compiler either because detection is not feasible or is not possible. Good programming practice will reduce the occurrence of undetected errors. The major undetected errors are listed below in no particular order.

- Do Case Range: There is no compiler or hardware protection against a do case index exceeding the range of the supplied cases. It is good coding practice to protect against range violation.

- Subscript Range: Likewise, there is no compiler or hardware protection against a subscript exceeding the declared dimension of an array. Indeed, it is common practice to declare external arrays to have dimension 0 in all but a single routine. Still, it is good coding practice to guard against subscript range violation.

- Dangling Else: The language associates an else clause with the innermost unmatched then-clause. It is good coding practice to physically format the source program to ensure that this association is correct.

- Object Cards in Source: A source record with a pound sign ('#') in the first byte is passed to the object program directly. This feature is activated by the #-toggle and it is good practice to deactivate it when not in use lest legitimate source be unadvertantly missed.

- Non-Distinct Symbol References: As a block structured language, MPL allows a symbol defined in an outer block to be used in an inner block. Indeed, this is the prominent feature of block structuring. However, in the case of scratch variables this feature is generally a liability as the use of the scratch variables gets inadvertantly multiplexed. It is good coding practice to define scratch variables separately for each block.

- Forward Labels: There is one circumstance where the current compiler, that makes only a single pass over the source, fails to detect a scoping violation.

Consider the program:

```
p: PROC;
   DCL x WORD;
   q: PROC;
      x = 1;
x:    GO x;
   END q;
END p;
```

The multiple use of x as a variable and label goes undetected. This circumstance is rare and, inasmuch as the correct code is generated, is only of academic interest.

- Redefined Literallys: The compiler performs literally text substitution before symbol processing. Thus, in spite of the fact that literally symbols follow the ordinary scoping rules and may be redefined in an inner block, in effect redefinition is not possible. The following example illustrates this problem:

```
p: PROC;
   DCL x LITERALLY '$*%#';
   q: PROC (x);
      DCL x WORD;
   END q;
END p;
```

The legal redefinition of x is never honored.

- Initial Value Range: Care should be taken to assure that the interpretation of constant precision is appropriate to the use at hand; lest constant truncation go undetected. See Subsection 5.2.4.

- Distribution of Initial Lists Over Namelists: When an initial list is applied to a namelist the length of the initial list is validated against the aggregate length of the namelist. The proper association of each name to each part of the initial list cannot be validated by the compiler.

- Interative Do Index: The index of an interative do (or repeat) is a signed word variable. Thus, the use of positive indicies over 32,767 requires the use of a do-while construct.

- Operand Size Alteration: In evaluating an expression the size of an operand may be automatically altered according to the translation algorithms of Section 5.5. Care must be taken to ensure that such size alteration preserves the sense of the desired operation. For example, the logical comparisons (LEQ, etc.) convert both operands to word size and any decision based on the high-order part of a double size operand will be erroneous.

4.3 FOULS: CONSISTENCY CHECKS

The compiler contains many internal consistency checks, called fouls, to facilitate compiler checkout and maintenance. For example, fouls are used to guard against faulty symbol table use. A foul will never occur if the compiler and hardware are functioning properly. Should a foul persist, contact Microdata.

A list of fouls is given in Appendix G. A foul results in a message being issued on both the console and program listing in the form:

```
    'XXXX' FOUL
```

where 'XXXX' represents the type of foul.

4.4 HARDWARE CONSIDERATIONS

There are certain hardware characteristics of the 32/S processor that may lead to undiagnosed errors, or unreported operational problems. This section is a discussion of these hardware aspects, indicating how they are exhibited and how they may be minimized.

- Memory Validity: On a 32/S not equipped with memory parity, failing memory may be undetected. The symptoms of failing memory are any unexplained compiler behavior. On a machine equipped with parity, the compiler may be used to validate memory.

  The compiler is capable of checking memory validity in the program space it occupies: from PB (program base) to PB+PL (program length). The default toggle settings check memory at the end of each program. By use of the V-toggle, memory validity may be checked each source record. (Memory checking is too time consuming to do this as a matter of course.)

  Should bad memory be a problem, place the suspect memories between PB and PB+PL and ensure that the M-toggle and V-toggles are on, and hardware parity is disabled. A faulty memory is indicated by a console and listing message:

  ```
  BAD MEMORY IN MODULE mm, ROW rr, BIT bb, BITS xx
  ```

  where (mm,rr,bb) jointly specify a faulty memory chip. If a multiple bit error has occurred, (xx) will have more than one bit on and the rightmost will be decoded as (mm,rr,bb). Bad memory also results in Abort 30.

- Stack Overflow: On a 32/S not equipped with stack overflow protection, stack overflow will go undiagnosed until program end. The stack overflows into the symbol table; thus the symptoms of stack overflow are unexpected symbol table behavior, usually a foul or a loop in a reference chain.

  The compiler checks for stack overflow at program end, and if the V-toggle is on, each source record. Stack overflow results in Abort 36.

  The compiler stack size has been chosen so that any ordinary program is accommodated. See Section 2.3 for a discussion of compiler stack use.

- Misread Source Data: Source devices occasionally misread data. The entropy of MPL is low enough so that most instances of misread source data cause computer flags.

- Miswritten Object Data: Object devices occasionally misread data. Object records are checksummed and sequenced so that all instances of miswritten object data are detected later by the loader.

5.0 TRANSLATION ALGORITHMS

The translation of source program to object program is specified by a set of translation algorithms that are the subject of the present chapter. These algorithms define the language as actually implemented. Translation algorithms are useful whenever knowledge of the generated code is required. Should these algorithms be found unclear or incomplete, the generated code may be displayed by use of the C-toggle, and the generated loader directives displayed by use of the H-toggle.

An analytic (top-down) approach is taken in presenting the translation algorithms: blocks are discussed first and operands last. Some algorithms show the direct translation of MPL statement to loader directives. Other algorithms show the translation of MPL statements to 32/S instructions. The translation of 32/S instructions to loader directives, and the loader directives themselves, are discussed in the final sections of this chapter.

Source text may be passed directly to the object program, bypassing all translation activity, by use of the #-toggle as discussed in Appendix F. This is the only mechanism available to generate arbitrary instruction sequences. There is no mechanism in the language to generate unimplemented or unused instructions, or instruction sequences that are not the product of an algorithm in this chapter. The location counters maintained by the compiler during translation are:

Mnemonic	Name	Number	Initial Value
pc	Program Counter	1	0
xc	Static Counter	1	0
sc	Stack Counter	1 per block	8

The program and static counters are common to all blocks within an external procedure, start at 0, and increase monotonically as translation progresses. The stack counter is unique to each block and starts at 8 to leave room for the stack mark. The stack counter moves up and down as translation progresses, returning to its initial state at the end of each block; each 32/S instruction has an associated stack increment (or decrement).

In the code sequences that follow, a box in the place of an instruction (□) represents the generated code for the entity enclosed in the box

5.1 BLOCKS

The block translation algorithms are shown in Figure 5.1a.

Blocks are of 4 types: main, procedure, interrupt, and begin. A block may be internal or external. External blocks, that is programs, are bounded by the begin and end loader directives. Internal begin blocks are entered with a BENT instruction; other internal blocks are skipped over with a BRA instruction. A block is exited with one of the 3 exit instructions as determined by the block type.

Block Type	Scope	Entry Loader Directives	Exit Loader Directive	Entry 32/S Instruction	Exit 32/S Instruction
Main	External	"81', "90"	"82"	--	EXIT
Procedure	External Internal	"81", "85" --	"82" --	-- BRA <end>	EXIT EXIT
Interrupt	External Internal	"81", "86" --	"82" --	-- BRA <end>	IXIT IXIT
Begin	Internal	--	--	BENT	BXIT

Figure 5.1a: Block Translation Algorithms

5.2 DECLARATIONS

Symbol declarations are contained in procedure heads, labels, and declaration statements. The attributes of a symbol are stored in the symbol table. The declaration process may be viewed as the translation of declarations into attributes in the symbol table. (The classification of symbols is discussed in conjunction with the symbol table listing in Section 3.3.7.)

The only declarations that result in loader directives are those for external symbols. The only declarations that result in 32/S instructions are those for labels and initial values.

In the case of a symbol that is a label, parameter, or procedure, the complete declaration may be fragmented in two parts. In such cases the symbol is marked as undefined until all attributes are in hand. A symbol that is undefined when used (which is illegal) results in automatic declaration appropriate to the context in which the symbol is used.

A procedure head declaration defines the entry name. In the case of a forward procedure, the procedure head is the second half of a fragmented declaration, and the size attribute must agree with that previously declared. The generated code and loader directives issued for a procedure head are discussed in conjunction with block translation in Section 5.1.

A label declaration defines the label name. In the case of a forward label, the label declaration is the second half of a fragmented declaration. A label is reclassified as a do label or begin label if it is subsequently found to appear on a group or begin. This subclassification of labels is used to mechanize labeled-end checking. A label generates an SSP instruction to purge any iteration context from the stack should the label be invoked from within an iterative group:

```
        SSP (sc-2)/2
```

The declaration-statement declarations form the bulk of the declarations. In the case of parameters, the declaration statement is the second half of a fragmented declaration; in the case of forward procedures, the first half. Each name is entered in the symbol table; the symbol class and all attributes, except value, are filled in from the attributes appearing in the declaration statement. As each external procedure is encountered an "88" loader directive is issued; as each literally is encountered the literally text is saved in the symbol table.

5.2.1 Value Allocation

The value attribute is determined when all declaration statements are in hand, that is, when the first non-declaration statement is encountered. (It is necessary to delay address allocation to this time to accommodate parameters which must be allocated first, but need not be declared first.) Figure 5.2a shows the use and value of the value attribute. Before the value is assigned it is rounded up to a word boundary, if appropriate, as indicated in the word alignment column. After the value is assigned it is incremented by the value increment column.

At the same time values are assigned, external variables are delivered to the loader with a series of "8F" directives. The extent field of this directive uses the value increment of Figure 5.2.la.

Following a declaration set, an SSP instruction is issued to skip over the stack mark, initial values, variables, and parameters:

SSP (sc-1)/2

and, finally the stack counter, sc, rounded up to a word boundary in preparation for future stack activity.

Class	Scope	Size	Value	First Value	Word Alignment	Value Increment
Procedure	External	--	External procedure sequence number	1	--	1
	Internal	--	pc	0	--	--
Static	External	--	External variable sequence number	1	--	1
	Internal	--	xc	0	Except byte	See automatic
Constant	--	--	pc	0	Except byte	See automatic
Parameter	--	Double	sc	8	Yes	4
		Other				2
Automatic	--	Byte	sc	8	Only if first of namelist and initialized	1*D
		Word			Yes	2*D
		Double			Yes	4*D
		Bit(1)			Yes	(1*D+7)/8
		Bit(2)			Yes	(2*D+7)/8
		Bit(4)			Yes	(4*D+7)/8
		Pointer			Yes	2*D

pc - program counter
xc - static counter
sc - stack counter
D - dimension+1

Figure 5.2.1a: Value Attribute for Symbols

5.2.2 Preset Variables

There are two types of preset variables: constant variables (constants) and initialized variables (initialized automatic variables). Constants require less execution time and space but may not be addressed as freely as initialized variables. There are two ways of specifying the preset values: with an initial string or with an initial list, as discussed in the next two subsections. These sections contain examples of preset variable translation.

Constant variable text appears only in program space and never in the stack. The text is inserted in program space and skipped over with a branch instruction. A new branch is issued if the constant set is broken by an initial variable. A BYTE pseudo-instruction is issued, if necessary, to align a constant of word or double size on a word boundary.

Initialized variable text appears in program space and is moved to the stack each time the procedure is entered. Data is moved from program space to stack space with FILL instructions. A SSP instruction is also generated for each name list to route the FILL data to the proper stack location:

```
        SSP (sc-2)/2
```

where sc is stack address of the first item of the name list.

5.2.3 Initial Strings

Initial text that appears as a string is converted to an infix format. (A string appearing as an element in an initial value list is used without alteration.)

The infix string format is:

length^1	t^1	e^1	x^1	t^1	0^1

The <length> indicates the number of characters in the <text>. Note that a string is restricted to 255 bytes. A 0 is appended to the string, if necessary, to pad out the text for a FILL instruction to a full word. The total length of the converted string is:

Preset Type	Length Byte	Padded	Total Length
Constant	--	No	<length>+1
Initial	Odd Even	No Yes	<length>+1 <length>+2

If the first or only name of the name list is not an array the variable is changed to an array and the <length> inserted as the dimension; if the variable is an array, the dimension is validated against the string length: the length must exactly match for a constant variable and may not be longer for an initialized variable. The size attribute must be BYTE or a flag is generated.

Should an initial string be applied to a name list the translation depends on whether the first name is dimensioned. If it is not, the string will exactly match the created dimension of the first name and subsequent names will not be initialized. If a dimension is supplied and is more than the string length, the tail of the first variable, and all subsequent variables are not initialized; a flag is generated for a constant variable. Finally, if a dimension is supplied and is less than the string length, the string text will spill into the trailing variables; a flag is always generated and the length field for subsequent variables contains a <text> character.

The code generated for an initial string is a series of DBLE pseudo-instructions. If the total length is not a multiple of 4, the WORD and BYTE pseudo-instructions are used for the final bytes. For an initial variable, a FILL instruction is inserted every 16 words; the final FILL may have a count of less than 16.

The example below illustrates initial string translation:

```
P: PROC;  DCL       /* &C */
  A         BYTE CONSTANT  'ABC',
                              0 0000 *   47 0000     BRA  0
                              3 0003 *      03414243 DBLE 54608451
  B(1)      BYTE CONSTANT  '''',
                              7 0007 *      0127     WORD 295
  C         BYTE INITIAL   '',
                              0 0000 **  47 0009     BRA  9
                              9 0009 *   5A 0000     SSP  0
                             12 000C *   59 01       FILL 1
                             14 000E *      0000     WORD 0
  D(99)     BYTE INITIAL   '123456789A123456789B123456789C12';
                             16 0010 *   5A 0000     SSP  0
                             19 0013 *   59 10       FILL 16
                             21 0015 *      20313233 DBLE 540095027
                             25 0019 *      34353637 DBLE 875902519
                             29 001D *      38394131 DBLE 943276337
                             33 0021 *      32333435 DBLE 842216501
                             37 0025 *      36373839 DBLE 909588537
                             41 0029 *      42313233 DBLE 1110520371
                             45 002D *      34353637 DBLE 875902519
                             49 0031 *      38394331 DBLE 943276849
                             53 0035 *   59 01       FILL 1
                             55 0037 *      3200     WORD 12800
END P;
                              9 0009 **  5A 0003     SSP  3
                             16 0010 **  5A 0004     SSP  4
                             57 0039 *   5A 0036     SSP  54
                             60 003C *   54          EXIT
```

5.2.4 Initial Lists

Initial text that appears in a list is packed into a buffer whose elements correspond to the declared size: double, word, byte, bit(4), bit(2), or bit(1)· Pointers are treated as word; after packing, bits are treated as byte. The buffer is padded with 0's to the next word boundary. There is no limit to the length of an initial list.

If an initial list is applied to a name list, the first name list element corresponds to the first initial list element, the second to the second, and so forth. Should the name list contain arrays, each array element qualifies as a name list element.

For a constant variable name list, the aggregate number of elements in the name list must exactly match the number of elements in the initial list. For an initialized variable name list, the initial list may not be longer.

Each constant of the initial list is inserted in the element buffer as follows. The constant magnitude is first evaluated in 32 bit precision. A preliminary overflow flag is generated if a bit string has more than 32 bits (leading 0's are significant), a character string has more than 4 bytes, or a decimal number has more than 31 magnitude bits (leading 0's are not significant). The constant sign, if any, is applied using 2's complement arithmetic. The 32 bit result is then inserted in the element buffer truncating high-order bits. An overflow flag is generated if all the truncated bits do not have the same sign.

This interpretation of precision treats a constant as a magnitude number whole value may be represented with a signed number. Since decimal constants in expressions are strictly 2's complement (see Section 5.7), care must be exercised as shown in the following examples of constant translation in initial lists:

External Value	Declared Variable Size	Internal Value	Flagged	Implicit Expression Size
4	Bit(2)	"0"	Yes	Word
-1	Bit(4)	"F"	No	Word
60000	WORD	"EA60"	No(!)	Double

The code generated for an initial list is a series of DBLE, WORD, or BYTE pseudo-instruction. For an initial variable, a FILL instruction is inserted every 16 words; the final FILL may have a count of less than 16 and may be padded with an extra byte of 0 to fill out the final word.

The example below illustrates initial list translation:

```
P: PROC;  DCL      /* &C */
   A         BYTE CONSTANT  (1),
                                    0 0000 *   47 0000     BRA  0
                                    3 0003 *      01       BYTE 1
   B(1)      WORD INITIAL   (2, 3),
                                    0 0000 **  47 0004     BRA  4
                                    4 0004 *   5A 0000     SSP  0
                                    7 0007 *   59 02       FILL 2
                                    9 0009 *      0002     WORD 2
                                   11 000B *      0003     WORD 3
   (CR, LF)  BYTE INITIAL   ("0D", "0A"),
                                   13 000D *   5A 0000     SSP  0
                                   16 0010 *   59 01       FILL 1
                                   18 0012 *      0D       BYTE 13
                                   19 0013 *      0A       BYTE 10
   D(2)      BIT(2) CONSTANT(0, 1, 2),
                                   20 0014 *   47 0000     BRA  0
                                   23 0017 *      00       BYTE 0
                                   24 0018 *      18       BYTE 24
   E(99)     BYTE INITIAL   ('A', 'B', 'C');
                                   20 0014 **  47 0019     BRA  25
                                   25 0019 *   5A 0000     SSP  0
                                   28 001C *   59 02       FILL 2
                                   30 001E *      41       BYTE 65
                                   31 001F *      42       BYTE 66
                                   32 0020 *      43       BYTE 67
                                   33 0021 *      00       BYTE 0
END P;
                                    4 0004 **  5A 0003     SSP  3
                                   13 000D **  5A 0005     SSP  5
                                   25 0019 **  5A 0006     SSP  6
                                   34 0022 *   5A 0038     SSP  56
                                   37 0025 *   54          EXIT
```

5.3 GROUPS

There are 6 grouping statements: simple do, repeat, do forever, do while, do iterative, and do case. The simple do generates no code. All others generate some code for the group head and some code for the matching group end. The do case also genereates code for each enclosed case (block sentence).

In the algorithms in this section it is understood that all expressions that result in double size are converted to word size with a SNGL instruction. Further, should the distance of a backward branch exceed 254 bytes, the long form is substituted for the short form shown: DBL for DBB, BRA for BRB, and DSBL for DSBB.

- REPEAT exp TIMES:

```
        exp
        BRA     y
x       body
y       DBB     x
```

- DO FOREVER:

```
x       body
        BRB     x
```

- DO WHILE exp:

```
x       exp
        BRF     y
        body
        BRB     x

y
```

- DO ref = expl TO exp2 BY exp3:

```
        exp1
        exp2
        exp3
        LADR     ref

        DIB      y

  x     body

        DSBB     x

  y
```

If the increment is missing, a L1 instruction is substituted for the evaluation of <exp3>.

- DO CASE exp:

```
        exp

        CASE     x

  c0    case 0

        BTOS

  cl    case 1

        BTOS
        .
        .
        .
  cn    case n

        BTOS

        NOP

        ADDR     cn
        .
        .
        .
        ADDR     cl
  x     ADDR     c0
```

The NOP instruction at the head of the case branch table is generated only if required to force the table to a word boundary.

5.4 STATEMENTS

There are only 6 true statements in MPL if blocks, groups, and declarations are considered separately. These are treated in this section in alphabetical order: assignment, call, go-to, if, null, and return.

Assignment Statement --

There are 4 flavors of the assignment statement: regular, multiple, embedded, and increment. The last 3 require an expression of size word (and a storage reference of size word). If the expression results in size double, a SNGL instruction, not shown below, is generated; if the storage reference is not of size word, a flag is generated.

- Regular ref = exp:

  $\boxed{\text{exp}}$

  $\boxed{\text{store}}$

The <ref> store is in detail in Section 5.6.3.

- Multiple, ref0 := ref1 ... := refn := exp:

  $\boxed{\text{exp}}$

```
STWN    refn
.
.
.
STWN    ref1
STW     ref0
```

- Embedded, ref := exp:

  $\boxed{\text{exp}}$

```
STWN    exp
```

- Increment, ref += exp:

  $\boxed{\text{exp}}$

```
AWM     ref
```

Call Statement --

The call statement is discussed in conjunction with procedure references in Section 5.7.

Go-to Statement --

A go-to statement generates a GOTO instruction.

If Statement --

An if statement has slightly different translation depending on the presence of an else unit:

- If exp THEN unit:

```
          exp
          BRF      x
          unit
  x
```

- If exp THEN unit1 ELSE unit2:

```
          exp
          BRF      x
          unit1
          BRA      y
  x       unit2
  y
```

Null Statement --

The null statement generates no code.

Return Statement --

The return statement generates an EXIT instruction unless the containing procedure block is an interrupt procedure in which case an IXIT is used. A return statment in a begin block behaves as if the return statement were in the containing procedure block.

Should the return statement contain an expression, the expression is evaluated and converted to the declared size of the current procedure. The conversion instructions generated are:

		Procedure Size	
Expression Size	Byte	Word	Double
Word	LBL "FF" AND	--	DBL1
Double	SNGL LBL "FF" AND	SNGL	--

The following program illustrates statement translation:

```
P: PROC BYTE;       /* &C */
X: BEGIN;  DCL (A,B,C) WORD;
                                      0 0000 *    5A 0003      SSP  3
                                      3 0003 *    5A 0003      SSP  3
                                      6 0006 *    53           BENT
   A = 1;
                                      7 0007 *    5A 0006      SSP  6
                                     10 000A *    71           L1
                                     11 000B *    B2 04        STW  2,A
   A := B := C := 2;
                                     13 000D *    72           L2
                                     14 000E *    AA 06        STWN 2,C
                                     16 0010 *    AA 05        STWN 2,B
                                     18 0012 *    B2 04        STW  2,A
   A = (B := 3);
                                     20 0014 *    73           L3
                                     21 0015 *    AA 05        STWN 2,B
                                     23 0017 *    B2 04        STW  2,A
   A += 4;
                                     25 0019 *    74           L4
                                     26 001A *    A2 04        AWM  2,A
   GO TO X;
                                     28 001C *    57 010003    GOTO 1,X
   IF 1<2  THEN A=5;
                                     32 0020 *    71           L1
                                     33 0021 *    72           L2
                                     34 0022 *    28           LT
                                     35 0023 *    13 0000      BRF  0
                                     38 0026 *    75           L5
                                     39 0027 *    B2 04        STW  2,A
   IF 1=2  THEN A=6;  ELSE A=7;
                                     35 0023 **   13 0029      BRF  41
                                     41 0029 *    71           L1
                                     42 002A *    72           L2
                                     43 002B *    2A           EQ
                                     44 002C *    13 0000      BRF  0
                                     47 002F *    76           L6
                                     48 0030 *    B2 04        STW  2,A
                                     50 0032 *    47 0000      BRA  0
                                     44 002C **   13 0035      BRF  53
                                     53 0035 *    77           L7
                                     54 0036 *    B2 04        STW  2,A
                                     50 0032 **   47 0038      BRA  56
   ;
   RETURN "10000";
                                     56 0038 *    42 00010000 LDL  65536
                                     61 003D *    37           SNGL
                                     62 003E *    41 FF        LBL  255
                                     64 0040 *    25           AND
                                     65 0041 *    54           EXIT
END X;
                                     66 0042 *    58           BXIT
END P;
                                     67 0043 *    54           EXIT
```

5.5 OPERATORS

This section discusses the translation of operators. For the present purposes, an expression is composed of a series of operators applied to operands. (The language allows for operator precedence, subexpressions, and conditional expressions.) Operand fetching, discussed in the next section, places the operand in the stack; the result of any operation is also in the stack. Each operand and the result of an operation are either of size word or double. Operand translation concerns the determination of the result size, the conversion of the operands to a compatible size, and the selection of the 32/S instruction to execute the operation.

Operators are of 2 types: unary (1 operand) and binary (2 operands).

Unary Operators --

There are 3 unary operators as shown in Figure 5.5a. The unary addition operator ('+') generates no code, the others a single instruction. The size of the result is always the same as the size of the operand.

Binary Operators --

There are 26 binary operators as shown in Figure 5.5d. There are 5 classes of binary operators from the point of view of the translation algorithms: other, shift, logical, DIVD, and MULD. Figure 5.5b shows the result size by class, Figure 5.5c the operand preparation by class, and Figure 5.5d the code generation by individual operation and also the class definitions.

- Other: The other class consists of the bulk of the arithmetic operators. It distinguishes the word-word case from the others. The word-word case is done exclusively in word size, the other cases convert word to double, use double operations, and produce a double result.

- Shift: The shift class consists of the 4 shifts. It operates in, and produces a result of, the first operand size, but requires the second operand, the shift count, to be word.

- Logical: The logical class consists of the 6 logical comparisons. It works exclusively in word size.

- DIVD: The DIVD class consists only of DIVD. The first operand (the dividend) must be double, the second operand (the divisor) must be word, and the result is always word.

- MULD: The MULD class consists only of MULD. Both operands must be word and the result is always double.

Unary Operator	Operand Size	
	Word	Double
+	--	--
-	NEG	DNEG
↑	NOT	DNOT

INSTRUCTIONS GENERATED

Unary Operator	Operand Size	
	Word	Double
+	Word	Double
-	Word	Double
↑	Word	Double

RESULT SIZE

Figure 5.5a: Unary Operator Translation

Fig 5.5b

Binary Operator Class	Operand Sizes			
	Word-Word	Double-Double	Word-Double	Double-Word
Other	Word	Double	Double	Double
Shift	Word	Double	Word	Double
Logical	Word	Word	Word	Word
DIVD	Word	Word	Word	Word
MULD	Double	Double	Double	Double

Figure 5.5b: Binary Operator Result Size

Binary Operator Class	Operand Sizes			
	Word-Word	Double-Double	Word-Double	Double-Word
Other	op1 op2	op1 op2	op1 op2 DBL2	op1 op2 DBL1
Shift	op1 op2	op1 op2 SNGL	op1 op2 SNGL	op1 op2
Logical	op1 op2	op1 SNGL op2 SNGL	op1 op2 SNGL	op1 SNGL op2
DIVD	op1 DBL1 op2	op1 op2 SNGL	op1 DBL1 op2 SNGL	op1 op2
MULD	op1 op2	op1 SNGL op2 SNGL	op1 op2 SNGL	op1 SNGL op2

op1 = code for first operand fetch
op2 = code for second operand fetch

Figure 5.5c: Binary Operator Operand Preparation

Fig 5.5d

		Operand Sizes			
Binary Operator Class	Operator	Word-Word	Double-Double	Word-Double	Double-Word
Other	+	ADD	DADD		
	-	SUB	DSUB		
	*	MUL	DMUL		
	/	DIV	DDIV		
	&	AND	DAND		
	|	OR	DOR		
	XOR	XOR	DXOR		
	MOD	MOD	DMOD		
	<=	LE	DLE		
	<	LT	DLT		
	=	EQ	DEQ		
	↑=	NE	DNE		
	>	GT	DGT		
	>=	GE	DGE		
Shift	SLC	SLC	DSLC	SLC	DSLC
	SLL	SLL	DSLL	SLL	DSLL
	SRA	SRA	DSRA	SRA	DSRA
	SRL	SRL	DSRL	SRL	DSRL
Logical	LEQ	LEQ			
	LGE	LGE			
	LGT	LGT			
	LLE	LLE			
	LLT	LLT			
	LNE	LNE			
DIVD	DIVD	DIVD			
MULD	MULD	MULD			

Figure 5.5d: Binary Operator Code Generation

5.6 OPERANDS

This section discusses expression operands. There are 5 types: procedure references, subexpressions, conditional expressions, literals, and memory references. Procedure references are discussed in Section 5.7. Subexpressions are a linguistic artifact and have no bearing on the translation algorithms.

5.6.1 Conditional Expressions

A conditional expression operand has the form:

IF exp1 THEN exp2 ELSE exp3

The result size of a conditional expression is double if either <exp2> or <exp3> is double; the result size is single if both <exp2> and <exp3> are word. The code generated is:

Size2 = Size3			Size2 = Double Size3 = Word			Size2 = Word Size3 = Double		
	$\boxed{\text{exp1}}$			$\boxed{\text{exp1}}$			$\boxed{\text{exp1}}$	
	BRF	x		BRF	x		BRF	x
	$\boxed{\text{exp2}}$			$\boxed{\text{exp2}}$			$\boxed{\text{exp2}}$	
	BRA	y		BRA	y		BRA	y
x	$\boxed{\text{exp3}}$		x	$\boxed{\text{exp3}}$		x	$\boxed{\text{exp3}}$	
y				DBL1			BRA	z
			y			y	DBL1	
						z		

5.6.2 Literals

Literal operands may be written in 3 forms: bit strings, character strings, and decimal numbers. A literal operand does not include a sign. The literal is converted to a 32 bit internal value with an associated precision and a load literal instruction generated. An overflow flag is generated if the internal representation requires more than 32 bits. Unused bit positions are filled with zeroes.

- Bit Strings: A bit string is regarded as representing a binary magnitude number. It has the bit precision of the number of bits written. Leading zeroes are significant. If overflow occurs, the 32 low-order bits are retained.

- Character Strings: A character string is regarded as representing a binary magnitude number, as with a bit string. It has the bit precision of the number of characters written times 8. If overflow occurs the high-order characters are retained.

- Decimal Numbers: A decimal number is regarded as representing a signed binary number. It has the precision of the number of magnitude bits present in the converted binary representation plus a sign bit (which is always 0). Leading zeroes are not significant. If overflow occurs, the 32 low-order bits are retained. This interpretation of decimal precision differs from that used in initial lists (see Section 5.2.4).

The load literal instruction generated is a function of the precision:

Precision (bits)	Opcode	Operand
0-4	Lx	none
5-8	LBL	1 byte literal
9-16	LWL	2 byte literal
17-32	LDL	4 byte literal

5.6.3 Memory References

Memory reference operands occur in 2 symetric forms: loads and stores. This subsection deals exclusively with loads but applies equally to stores with the following alterations: all load operations become store operations; memory reference preparation (index evaluation, etc.) is separated from expression size conversion and the actual store, by expression evaluation; and, finally, constant variables may be loaded from but not stored into. (Parameters may be stored into, but since they are mechanized with a call-by-value scheme, they disappear when the procedure is terminated.)

The memory reference translation algorithms are shown in Figures 5.6.3a (non-automatic variables) and 5.6.3b (automatic variables). For non-automatic variables there is no difference in the algorithms for the different sizes, except of course, for the actual store. For automatic variables the address for byte size is a byte address, the address for word and double size is a word address. In all cases the size field for a LADR instruction is conditioned by the variable size. Figure 5.6.3c shows the correspondence between hardware address mode and language variable class.

Memory reference translation for automatic variables is a function of locality and span. A variable is local if it is referenced in the same block it is defined; a variable is remote if it is referenced in a block that is nested in the block in which it is defined. The span of a variable is its EP byte address for a byte variable, and its EP word address (byte address divided by 2) for a word or double variable; the span is short if less then 255, long if 256 or over.

The pointer size attribute may be applied to a variable of any class. The code consists of a load of the pointer followed by a load of the actual data. The bit size attribute and field select specification may be applied to any storage class; however, the size attribute must be word for field select and is effectively word for bit, as the final LF (load field) instruction assumes word. The algorithms shown assume that the load can be accomplished in a single LF instruction. Should an index or other supporting operand be required, the necessary code is inserted before the final LF per the algorithm of the parent symbol class.

The 2 argument form of a field select specification generates a field descriptor using a GFD instruction based on the supplied length and position; the 1 argument form assumes the single argument is a prefabricated descriptor.

Data when allocated by the translator is always left justified. A bit scalar occupies the leftmost bits of the full word required to hold it leaving the rightmost bits unused; a byte variable, allocated on an even byte, followed by a word variable, leaves the preceding byte (the rightmost of the preceding word) unused. (This byte would have been used if the byte variable were followed by another byte variable.)

Data when loaded by a load instruction is always right justified. For bytes, this justification discrepancy can be resolved when addresses are assigned. For bits, this discrepancy can only be resolved when the field descriptor is generated for the load instruction: a non-parameter bit variable is left justified, a parameter bit variable is right justified.

Constant variable are subject to several constraints not shared by the other variables. As mentioned above, they may not be stored into. In addition, the address of a constant may not be passed as data (the LADR instruction has no provision for specifying the space the address is in: program or stack) or as the object of the address function ('@'). Note that although this addressing constant prevents pointer variables from being unused to reference constant arrays, the same effect can be achieved in the local routine by use of array addressing.

Some memory references require supporting operands in the stack at the time the actual memory reference is made: a base, an index, or a field select descriptor. In such cases, the supporting operands are inserted in the stack in the order they appear in a left-to-right scan with any basing operands appearing first. Thus a base or indirect address appears before an array index which appears before a bit or field descriptor. Further, some preliminary operands in the stack get used in the computation of others.

The following program illustrates some, but by no means all, aspects of memory reference translation.

```
P: PROC(PARAM_REMOTE_BIT2_ARRAY);  DCL               /* $W */

     AUTO_REMOTE_BYTE_SCALAR                     BYTE,
     AUTO_REMOTE_WORD_ARRAY(0)                   WORD,
     PARAM_REMOTE_BIT2_ARRAY(0)                  BIT(2);

     Q: PROC(PARAM_LOCAL_BYTE_SCALAR, PARAM_LOCAL_WORD_ARRAY);  DCL

          (INDEX, A_INDEX, P_INDEX,
             LEN, LOC, FIELDS)                   WORD,
          AUTO_LOCAL_SHORT_BYTE_SCALAR           BYTE,
          AUTO_LOCAL_SHORT_DBLE_ARRAY(0)         DOUBLE,
          FILLER(1000)                           BYTE,
          AUTO_LOCAL_LONG_BYTE_SCALAR            BYTE,
          AUTO_LOCAL_LONG_DBLE_ARRAY(0)          DOUBLE,
          STATIC_INT_BYTE_SCALAR                 BYTE   STATIC,
          STATIC_EXT_BIT1_ARRAY(0)               BIT(1) EXTERNAL,
          CBASE_BIT2_SCALAR                      BIT(2) BASED 10,
          VBASE_WORD_ARRAY(0)                    WORD   BASED STATIC_INT_BYTE_SCALAR,
          CONSTANT_BYTE_SCALAR                   BYTE   CONSTANT(12),
          PARAM_LOCAL_BYTE_SCALAR                BYTE,
          PARAM_LOCAL_WORD_ARRAY(0)              WORD,
          PTR_BYTE_SCALAR                        POINTER TO BYTE,
          PTR_WORD_ARRAY(0)                      POINTER TO WORD;

CALL P(   /* SC */
AUTO_LOCAL_SHORT_BYTE_SCALAR,
                              17 0011 *   FA 18       LB   2,AUTO_LOCAL_SHORT_BYTE_SCA
AUTO_LOCAL_SHORT_DBLE_ARRAY(INDEX),
                              19 0013 *   F2 06       LW   2,INDEX
                              21 0015 *   C3 0D       LD   3,AUTO_LOCAL_SHORT_DBLE_ARR
AUTO_REMOTE_BYTE_SCALAR,
                              23 0017 *   06 18000A   LADR 24,AUTO_REMOTE_BYTE_SCALAR
                              27 001B *   FC          LB   4
AUTO_REMOTE_WORD_ARRAY(INDEX),
                              28 001C *   06 19000C   LADR 25,AUTO_REMOTE_WORD_ARRAY
                              32 0020 *   F2 06       LW   2,INDEX
                              34 0022 *   F5          LW   5
AUTO_LOCAL_LONG_BYTE_SCALAR,
                              35 0023 *   06 080407   LADR 8,AUTO_LOCAL_LONG_BYTE_SCAL
                              39 0027 *   FC          LB   4
AUTO_LOCAL_LONG_DBLE_ARRAY(INDEX),
                              40 0028 *   06 0A0408   LADR 10,AUTO_LOCAL_LONG_DBLE_ARR
                              44 002C *   F2 06       LW   2,INDEX
                              46 002E *   C5          LD   5
STATIC_INT_BYTE_SCALAR,
                              47 002F *   F8 0000     LB   0,STATIC_INT_BYTE_SCALAR
STATIC_EXT_BIT1_ARRAY(INDEX),
                              50 0032 *   F2 06       LW   2,INDEX
                              52 0034 *   0D          XB1
                              53 0035 *   D1 0001     LF   1,STATIC_EXT_BIT1_ARRAY
CBASE_BIT2_SCALAR,
                              56 0038 *   7A          L10
                              57 0039 *   70          L0
                              58 003A *   41 1E       LBL  30
                              60 003C *   D7          LF   7
VBASE_WORD_ARRAY(INDEX) $ (LEN:LOC),
                                          61 003D *    F8 0000      LB    0,STATIC_INT_BYTE_SCALAR
                                          64 0040 *    F2 06        LW    2,INDEX
                                          66 0042 *    F2 09        LW    2,LEN
                                          68 0044 *    F2 0A        LW    2,LOC
                                          70 0046 *    2E           GFD
                                          71 0047 *    D7           LF    7
CONSTANT_BYTE_SCALAR,
                                          72 0048 *    70           L0
                                          73 0049 *    FE 0009      LB    6,CONSTANT_BYTE_SCALAR
PARAM_LOCAL_BYTE_SCALAR,
                                          76 004C *    FA 09        LB    2,PARAM_LOCAL_BYTE_SCALAR
PARAM_LOCAL_WORD_ARRAY(INDEX) $ (FIELDS),
                                          78 004E *    F2 05        LW    2,PARAM_LOCAL_WORD_ARRAY
                                          80 0050 *    F2 06        LW    2,INDEX
                                          82 0052 *    F2 0B        LW    2,FIELDS
                                          84 0054 *    D5           LF    5
PARAM_REMOTE_BIT2_ARRAY(INDEX),
                                          85 0055 *    06 190008    LADR 25,PARAM_REMOTE_BIT2_ARRAY
                                          89 0059 *    F4           LW    4
                                          90 005A *    F2 06        LW    2,INDEX
                                          92 005C *    0E           XB2
                                          93 005D *    D5           LF    5
PTR_WORD_ARRAY(INDEX),
                                          94 005E *    06 09040E    LADR 9,PTR_WORD_ARRAY
                                          98 0062 *    F2 06        LW    2,INDEX
                                         100 0064 *    F5           LW    5
PTR_BYTE_SCALAR @,
                                         101 0065 *    06 09040C    LADR 9,PTR_BYTE_SCALAR
                                         105 0069 *    F4           LW    4
                                         106 006A *    FC           LB    4
PTR_BYTE_SCALAR @ (P_INDEX),
                                         107 006B *    06 09040C    LADR 9,PTR_BYTE_SCALAR
                                         111 006F *    F4           LW    4
                                         112 0070 *    F2 08        LW    2,P_INDEX
                                         114 0072 *    FD           LB    5
PTR_WORD_ARRAY(A_INDEX) @ (P_INDEX) )   /* $C */ ;
                                         115 0073 *    06 09040E    LADR 9,PTR_WORD_ARRAY
                                         119 0077 *    F2 07        LW    2,A_INDEX
                                         121 0079 *    F5           LW    5
                                         122 007A *    F2 08        LW    2,P_INDEX
                                         124 007C *    F5           LW    5
                                         125 007D *    52 17        CALL 23
       END Q;
  END P;
```

Fig 5.6.3a

Symbol Class	Subcase	Scalar	Array
Static	Internal	L* 0,SB	L* 1,SB
	External	L* 0,0000	$\boxed{\text{index}}$ L* 1,0000
Constant Based		$\boxed{\text{cbase}}$ L0 L* 7	$\boxed{\text{cbase}}$ $\boxed{\text{index}}$ L* 7
Variable Based		$\boxed{\text{vbase}}$ L0 L* 7	$\boxed{\text{vbase}}$ $\boxed{\text{index}}$ L* 7
Constant		L0 L* 6,PB	$\boxed{\text{index}}$ L* 6,PB
Parameter	Local	see automatic Figure 5.6.3b	LW 2,EP $\boxed{\text{index}}$ L* 5
	Remote	see automatic Figure 5.6.3b	LADR Δ,size,EP LW 4 $\boxed{\text{index}}$ L* 5
Any	Pointer, no index	$\boxed{\text{pointer}}$ L* 4	$\boxed{\text{a-index}}$ $\boxed{\text{pointer}}$ L* 4
	Pointer, with index	$\boxed{\text{pointer}}$ $\boxed{\text{p-index}}$ L* 5	$\boxed{\text{a-index}}$ $\boxed{\text{pointer}}$ $\boxed{\text{p-index}}$ L* 5
	Bit	LBL bitspec LF mode,address	$\boxed{\text{index}}$ XBi LF mode,address
	Field select, 1 argument	$\boxed{\text{fields}}$ LF mode,address	$\boxed{\text{index}}$ fields LF mode,address
	Field select, 2 arguments	$\boxed{\text{length}}$ $\boxed{\text{location}}$ GFD LF mode,address	$\boxed{\text{index}}$ $\boxed{\text{length}}$ $\boxed{\text{location}}$ GFD LF mode,address

SB	- Stack base stack location
EP	- Environment pointer stack location
PB	- Program pointer program location
*	- B (byte), W (word), or D (double) depending on variable size
size	- "8" (byte), "9" (word), or "A" (double) depending on parameter size
Δ	- Differential lex level from current to variable
i	- 1 (BIT(1)), 2 (BIT(2)), 4 (BIT(4)) depending on bit size
0000	- 16 bit address of 0 filled in later by loader
local	- See text
span	- See text
mode	- Mode of parent variable
address	- Address of parent variable
cbase	- Code for loading constant base: a load literal of some variety, result is word
vbase	- Code for loading variable base, result is word
index	- Code for loading array index, result is word
a-index	- Code for loading array index, result is word
p-index	- Code for loading pointer index, result is word
pointer	- Code for loading pointer
length	- Code for loading bit field length, result is word
location	- Code for loading bit field length, result is word
fields	- Code for loading prefabricated field descriptor
bitspec	- Prefabricated descriptor of bit scalar:

Size	Parameter (right justified)	Non Parameter (left justified)
Bit (1)	"00"	"0F"
Bit (2)	"10"	"1E"
Bit (4)	"30"	"3C"

Figure 5.6.3a: Memory Reference Translation

Fig 5.6.3b

Case	Dimension	Byte Size	Word Size	Double Size
Local short-span	Scalar	LB 2,EP	LW 2,EP/2	LD 2,EP/2
	Array	$\boxed{\text{index}}$ LB 3,EP	$\boxed{\text{index}}$ LW 3,EP/2	$\boxed{\text{index}}$ LD 3,EP/2
Local long-span, Remote	Scalar	LADR Δ,8,EP LB 4	LADR Δ,9,EP LW 4	LADR Δ,A,EP LD 4
	Array	LADR Δ,8,EP $\boxed{\text{index}}$ LB 5	LADR Δ,9,EP $\boxed{\text{index}}$ LW 5	LADR Δ,A,EP $\boxed{\text{index}}$ LD 5

Note: Terms defined in Figure 5.6.3a

Figure 5.6.3b: Memory Reference Translation for Automatic Variables

Fig 5.6.3c

Hardware Address Mode	Effective Address	Use
0	SB + d16	Scalar static
1	SB + d16 + tos(x)	Array static
2	SB + EP + d8	Local short-span scalar automatic
3	SB + EP + d8 + tos(x)	Local short-span array automatic
4	SB + tos(d16)	Non-indexed pointer, Local longspan scalar automatic, Remote scalar automatic
5	SB + tos(x) + tos1(d16)	Indexed pointer, Parameter, Local long-span array automatic, Remote array automatic
6	PB + d16 + tos(x)	Constant
7	tos(x) + 4 * tos1(d18)	Based

SB - Stack base stack location

EP - Environment pointer stack location

PB - Program base program location

d8 - Displacement of 8 bits

d16 - Displacement of 16 bits

d18 - Displacement of 18 bits of which the low-order 2 are assumed 0

tos - Top of stack, use indicated in parentheses

tos1 - Next to top of stack, use indicated in parentheses

Figure 5.6.3c: Memory Reference Mode Use

5.7 PROCEDURE REFERENCES

This section discusses the translation algorithms for a called procedure. For the purposes of the present discussion it is convenient to classify procedure in 3 groups: user , micro, and built-in. Arguments are translated in the same manner for all procedures.

- Arguments: Scalar arguments are passed by value, array arguments by address.

  A scalar argument is any expression. (An expression may result in an address.) The expression is evaluated, which will leave the result in the stack.

  An array argument is a symbol declared as an array but referenced without a subscript. A LADR instruction is generated, which will place the address of the first array element in the stack.

  Note that all arguments occupy a word, except for an expression that results in a double size which will occupy 2 words.

  The following example illustrates argument translation:

```
P: PROC;  DCL       /* &C */
    SYSPUT    EXT PROC,
    BUF(79)   BYTE;
    CALL SYSPUT (BUF, LOW("1 0000" + 80));
                                    0 0000 *   5A 002B     SSP  43
                                    3 0003 *   50 140001   MARK 20,SYSPUT
                                    7 0007 *   06 080008   LADR 8,BUF
                                   11 000B *   42 00010000 LDL  65536
                                   16 0010 *   41 50       LBL  80
                                   18 0012 *   08          DBL1
                                   19 0013 * 4F00          DADD
                                   21 0015 *   37          SNGL
                                   22 0016 *   52 05       CALL 5
END P;
                                   24 0018 *   54          EXIT
```

- User Procedures: A user procedure reference generates:

```
CALL name (agr1, arg2,  .. argn);

     MARK     flags,location

     arg1

     arg2

      .
      .
      .

     argn

     CALL     length
```

  where <length> is the total size of the argument list in words including the mark; <location> is the PB relative address of the invoked procedure (which may be external); <flags> contains several pieces of information about the nature of the call as discussed in conjunction with the translation of the MARK instruction in Section 5.8.

- Micro Procedures: A micro procedure reference generates:

```
CALL name (arg1, arg2,  .. argn);

     arg1

     arg2

      .
      .
      .

     argn

     MICR location
```

  where <location> is the declared microstore address.

- Built-in Procedures: There are 17 built-in procedures all of which generate in-line code. A built-in is not declared, rather it is defined when first encountered. With the exception of the environment built-in, the definition is made in the outermost block, block 0; the environment built-in is defined anew in each block in which it is invoked. Note that the language permits a built-in name to be used as a user symbol; however, there is no way to redefine the symbol as a built-in in an inner block.

Figure 5.7a shows the attributes given to the built-ins.

The following points are worth noting. The built-ins shown with a return size may be invoked either in a call statement or expression. The built-ins without a return size may only be invoked in a call statement. The return size of abs is the same as its argument; that is *abs* is a generic built-in. The *environment* built-in is best viewed as an array of pointers whose first element is the first word of the current stack environment; it has the properties of any other array of pointers and generates code accordingly.

Arguments to built-in procedures are validated for correct size and flagged if incorrect. This is the only circumstance in the compiler where an expression is not automatically converted to the desired target size.

Fig 5.7a

Function	Number of Arguments	Argument Size	Return Size	Code Generated
abs	1	Word, Double	Word, Double	ABS[1]
carry	0,1	Word, Double	Word	TCAR[2]
dble	1,2	Word	Double	DBL1[3]
environment	1	Word	Word	[4]
high	1	Double	Word	POP
low	1	Double	Word	SNGL
nop	0	--	--	NOP
overflow	0,1	Word, Double	Word	TOVF[2]
pnop	0	--	--	PNOP
resume	1	Word	--	RESM
ssr	0	--	--	SSR
supervisor	1	Word	--	SUPV
switches	0	--	Word	ESW
trap	0	--	--	TRAP
wait	0	--	--	WAIT
xim	1	Word	Word	XIM
prtnum	1	Procedure	Word	LWL addr[5]

[1] DABS if argument size double

[2] Preceded by POP if argument size double, additional POP if 1 argument form

[3] No instructions generated if 2 argument form

[4] See text

[5] PB address of start of procedure for internal procedures, PLIB number of segment appended to PRTNUM of procedure for external procedure.

Figure 5.7a: Built-in Function Translation

5.8 INSTRUCTIONS

This section discusses the translation of 32/S instructions into loader directives. It is convenient to classify the instruction set 15 ways according to the opcode length and operand type as shown in Figure 5.8b. The mapping from instruction to class is shown in Figure 5.8d. The translation algorithms from instruction class to loader directive are also shown in Figure 5.8b and are amplified below.

Most instruction classes can occur only in-line. The branch, GOTO, and pseudo-ADDR classes may also occur out of line as fixups to previously generated directives where a 0 was used for a PB address that was a forward reference. Fixups are mechanized by changing the program counter to the fixup location, performing the fixup, then restoring the program counter. The original opcode is not rewritten as it remains unchanged.

Absolute text is implied in Figure 5.8b where a directive of the form <0x> is shown. The <x> bytes of absolute text are appended to an absolute text string if one is already in progress. A new absolute text string is begun if none is in progress or an object record boundary would be crossed.

The pseudo-instructions are created by the compiler and do not exist in the 32/S processor. They have no opcodes, consisting only of an operand.

The LADR, MARK, and GOTO instructions have an associated flag byte of absolute text described in detail in Sections 9.9, 9.2, and 9.10 respectively of Reference 6. The DLEX field, common to all instructions, measures the lex level difference between the current block and block of interest; if the reference is to the current block, DLEX is 0. For LADR the L-field encodes the size of the referenced item even if use of this field is not enabled by the I-flag which is set when LADR is used to compute the effective address of an array element specified by use of the address operator ('@'). The D-flag is set for all references except those to static data (internal, declared with STATIC area attribute, and external, declared with EXTERNAL). For MARK the Z-flag is always off indicating the called procedure is in the same segment, and the R-field is set according to the declared return size: 0 for none, 1 for word, 2 for double.

Fig 5.8a

Mnemonic	Opcode	Class
ABS	1E	1
ADD	20	1
ADDR	pseudo	11
AND	25	1
AW	E0	9
AWM	A0	9
BENT	53	1
BEQZ	1A	5
BGEZ	1C	5
BGTZ	1D	5
BLEZ	19	5
BLTZ	18	5
BNEZ	1B	5
BRA	47	5
BRB	46	3
BRF	13	5
BRT	12	5
BTOS	15	1
BXIT	58	1
BYTE	pseudo	12
CALL	52	3
CASE	14	5
DABS	3E	1
DADD	4F00	2
DAND	4F05	2
DBB	16	3
DBL	17	5
DBLE	pseudo	14
DBL1	08	1
DBL2	4F16	2
DDIV	4F03	2
DDUP	3F	1
DEQ	4F0A	2
DGE	4F0C	2
DGT	4F0D	2
DIB	48	5
DIV	23	1
DIVD	4F14	2
DLE	4F09	2
DLT	4F08	2
DMOD	4F04	2
DMUL	4F02	2
DNE	4F0B	2
DNEG	3C	1
DNOT	3D	1
DOR	4F06	2
DSBB	4A	3
DSBL	4B	5
DSLC	4F13	2
DSLL	4F10	2

Mnemonic	Opcode	Class
DSRA	4F11	2
DSRL	4F12	2
DSUB	4F01	2
DUP	1F	1
DXOR	4F07	2
EQ	2A	1
ESW	05	1
EXIT	54	1
FILL	59	3
GE	2C	1
GFD	2E	1
GOTO	57	8
GT	2D	1
IXIT	55	1
LADR	06	6
LB	F8	9
LBL	41	3
LD	C0	9
LDL	42	10
LE	29	1
LF	D0	9
LGE	3A	1
LGT	3B	1
LLE	39	1
LLT	38	1
LT	28	1
LW	F0	9
L*	7*	1
LWL	40	4
MARK	50	7
MICR	0C	4
MOD	24	1
MUL	22	1
MULD	36	1
NE	2B	1
NEG	10	1
NOP	01	1
NOT	11	1
OR	26	1
PNOP	02	1
POP	0B	1
RESM	56	1

Mnemonic	Opcode	Class
SLC	33	1
SLL	30	1
SNGL	37	1
SRA	31	1
SRL	32	1
SSP	5A	4
SSPI	5B	1
SSR	5F	1
STB	B8	9
STD	80	9
STF	88	9
STW	B0	9
STWN	A8	9
SUB	21	1
SUPV	09	1
SW	E8	9
TCAR	04	1
TOVF	03	1
TRAP	00	1
WAIT	5C	1
WORD	pseudo	13
XB1	0D	1
XB2	0E	1
XB4	0F	1
XCH	2F	1
XIM	0A	1
XOR	27	1
	07	0
	34	0
	35	0
	43	0
	44	0
	45	0
	49	0
	4C	0
	4D	0
	4E	0
	4F0E	0
	4F0F	0
	4F15	0
	4F17+	0
	51	0
	5D	0
	5E	0
	6*	0
	90	0
	98	0
	C8	0
	D8	0

Figure 5.8a: Instruction Set Classification

Class	Opcode Description	Operand Description	Total Length	Membership	Inline Directives	Fixup Directives
0	Unused			[1]	Never generated	
1	Short	None	1	ABS ... [2]	01/op	
2	Long	None	2	DADD .. [3]	02/4F+op	
3	Short	1 byte literal	2	BRB ... [4]	02/op+lit	
4	Long	2 byte literal	3	LWL MICR SSP	03/op+lit	
5	Branch	PB address	3	BEQZ .. [5]	01/op, 8B/PB address	83/fixup, 8B/PB address, 83/current
6	LADR	Flags + stack address	4	LADR	02/06+flags, 02/EP address (automatic variable) 8A/EDSN (external static variable) 8C/SB address (internal static variable)	
7	MARK	Flags + PB address	4	MARK	02/50+flags, 02/EPSN (external procedure) 02/0000 (internal forward procedure) 8B/PB address (internal backward procedure)	
8	GOTO	Flags + PB address	4	GOTO	02/57+flags, 8B/PB address	83/fixup, 01/flags, 8B/PB address, 83/current
9	Memory Reference	Varies	1-3	AW ... [6]	01/op, 8C/SB address (modes 0&1, internal variable) 8A/EDSN (modes 0&1, external variable) 01/lit (modes 2&3) 8B/PB address (mode 6) none (modes 4&5&7)	
10	LDL	4 byte literal	5	LDL	05/42+lit	
11	ADDR	PB address	2	ADDR	8B/PB address	83/fixup, 8B/address, 83/current
12	BYTE	1 byte literal	1	BYTE	01/lit	
13	WORD	2 byte literal	2	WORD	02/lit	
14	DBLE	4 byte literal	4	DBLE	04/lit	

current - current PB location
EDSN - External Data Sequence Number
EPSN - External Procedure Sequence Number
fixup - fixup PB location
flags - see text
lit - literal text or absolute operands
op - opcode

1 All opcodes with no mnemonics shown at the end of Figure 5.8b.

2
ABS
ADD
AND
BENT
BTOS
BXIT
DABS
DBL1
DDUP
DIV
DNEG
DNOT
DUP
EQ
ESW
EXIT
GE
GFD
GT

IXIT
LE
LGE
LGT
LLE
LLT
LT
L0
L1
L2
L3
L4
L5
L6
L7
L8
L9
L10
L11
L12
L13
L14
L15

MOD
MUL
MULD
NE
NEG
NOP
NOT
OR
PNOP
POP
RESM
SLC
SLL
SNGL
SRA
SRL
SSPI
SSR
SUB
SUPV

TCAR
TOVF
TRAP
WAIT
XB1
XB2
XB4
XCH
XIM
XOR

3
DADD
DAND
DBL2
DDIV
DEQ
DGE

DGT
DIVD
DLE
DLT
DMOD
DMUL

DNE
DOR
DSLC
DSLL
DSRA
DSRL

DSUB
DXOR

4
BRB
CALL

DBB
DSBB

FILL

LBL

5
BEQZ
BGEZ
BGTZ
BLEZ
BLTZ

BNEZ
BRA
BRI
BRT

CASE

DBL
DIB
DSBL

6
AW
AWM

LB
LD
LF
LW

STB
STD
STF

STW
STWN
SW

Figure 5.8b: Instruction Translation

5.9 LOADER DIRECTIVES

This section describes the loader directives used by the compiler.

Directives are written in ASCII on 80 byte fixed length records. Each record consists of a 5 byte preamble and up to 75 bytes of actual directives. Directives may cross record boundaries (although the compiler does not take advantage of this facility). The preamble consists of:

- Byte 1, Logical Record Sentinel: Always contains a pound sign ('#'). This permits logical records to be dissociated from physical records (although the compiler does not take advantage of this facility).

- Byte 2, Format Indicator: Always contains a blank ('b̸') indicating ASCII format. The loader can accommodate a binary format (although the compiler does not take advantage of this facility).

- Byte 3, Checksum: The checksum of each of the subsequently used bytes, starting with byte 4. The checksum is formed by byte addition with end-around carry.

- Byte 4, Sequence Number: The record number starting with 1 when the external procedure starts. If there are more than 255 records, the sequence number repeats going from 255 to 1.

- Byte 5, Byte Count: The number of bytes of load text in this record.

There are 15 loader directive types of which 2 are used only by the cross compiler but are included here for completeness. The first byte of each directive determines the directive type. The number of bytes in the directive is conditioned by the directive type. The directives are summarized in Figure 5.9a and described in detail below. The diagrams with each directive show the directive code, supporting operands, and the length of each field.

- 0 to "7F', Load Absolute Text: Specifies that the next <code> bytes contains absolute text. The loader will insert the text in program space without relocation. The loader program pointer will be advanced by <code> bytes.

code^{1}	text <code>

0 <= code <= "7F"

- "81", Begin External Procedure: Signals the beginning of an external procedure (or other external block). The loader will prepare for a new external procedure.

"81"^{1}

- "82", End External Procedure: Signals the end of an external procedure (or other external block). The static field contains the number of words of static storage rounded up a word boundary. The possible error codes are described in Section 4.1. The loader will increment its static relocation counter by the static length.

"82"^{1}	static length^2	error code^1

- "83", Set Program Pointer: Specifies a non-sequential change to the program pointer. The loader will set its program pointer to the location field (after relocation).

"83"^{1}	location^2

- "84", Define Cross Compiler Main Procedure Entry Point: Identifies the name of a main procedure compiled under the cross compiler. Directive "90" performs a similar function for the self compiler. The name field contains the first 8 characters of the entry name; characters beyond the 8th are lost and a short name is padded with trailing blanks. The location field contains the relocatable address of the first instruction; 0 is always used for the compilers at hand. The loader will add the name to its symbol table.

"84"^{1}	name^8	location^2

- "85", Define External Procedure Entry Point: Identifies the name of an external procedure. Use the same as directive "84".

"85"^{1}	name^8	location^2

- "86", Define External Interrupt Procedure Entry Point: Identifies the name of an external interrupt procedure. Use is the same as directive "84".

"86"^{1}	name^8	location^2

- "87", Define Cross Compiler External Variable Reference Number: Identifies the name of an external variable compiled under the cross compiler and assigns to it a reference number for later use by directive "8A". Directive "8F" performs a similar function for the self compiler. The name field contains the first 8 characters of the entry name; characters beyond the 8th are lost and a short name is padded with trailing blanks. The size field measures the extent of the variable in bytes. The loader will add the name to its symbol table.

"87"^{1}	size^2	name^8	reference number^2

- "88", Define External Procedure Reference Number: Identifies the name of an external procedure and assigns to it a reference number for later use by directive "89". The name field contains the first 8 characters of the entry name; characters beyond the 8th are lost and trailing blanks ('b̸') are used to pad a short name. The loader will add the name to its symbol table.

"88"^{1}	name^8	reference number^2

- "89", Reference External Procedure: Invokes an external procedure whose reference number was previously defined in an "88" directive. The loader will insert the runtime address of the referenced external procedure. The loader program pointer will be advanced by 2.

"89"^{1}	reference number^2

- "8A", Reference External Variable: Invokes an external variable whose reference number was previously defined in an "87" or "8F" directive. The loader will insert the runtime address of the referenced external variable in program space. The loader program pointer will be advanced by 2.

"8A"^{1}	reference number^2

- "8B", Relocate Program Address: Specifies relocation of a program address. The loader will insert the address field in program space relocating the address with respect to the loadtime program base. The loader program pointer will be advanced by 2.

"8B"^{1}	address^2

- "8C", Relocate Stack Address: Specifies relocation of a stack address. The loader will insert the address field in program space relocating the address with respect to the loadtime stack base. The loader program pointer will be advanced by 2.

"8C"^{1}	address^2

- "8F", Define Self Compiler External Variable Reference Number: Identifies the name of an external variable compiled under the self compiler and assigns to it a reference number for later use by directive "8A". Directive "87" performs a similar function for the cross compiler. The name field contains the first 8 characters of the entry name; characters beyond the 8th are lost and short name is padded with trailing blanks. The size field measures the extent of the variable in bytes. The <attributes> field encodes the variable data type. The loader will add the name to its symbol table.

"8F"^{1}	attributes ^{1}	size ^{2}	name ^{8}	reference number $_{2}$

units $^{0.4}$	size $^{0.4}$

<units>	<size>	Declared Size
1	1	BIT(1)
	2	BIT(2)
	3	BIT(4)
2	1	BYTE
	2	WORD
	3	DOUBLE
3	1	POINTER TO BYTE
	2	POINTER TO WORD
	3	POINTER TO DOUBLE

- "90", Define Self Compiler Main Procedure Entry Point: Identifies the name of a main procedure compiled under the self compiler. "84" performs a similar function for the cross compiler. Directive use is the same as directive "84".

"90"^{1}	name ^{8}	location ^{2}

Loader Code	Directive Length	Program Pointer Change	Description
0 to "7F"	<code> +1	code>	Load absolute text
"81"	1	--	Begin external procedure
"82"	4	--	End external procedure
"83"	3	set	Set program pointer
"84"	11	--	Define cross compiler main procedure entry point
"85"	11	--	Define external procedure entry point
"86"	11	--	Define external interrupt procedure entry point
"87"	14	--	Define cross compiler external variable reference number
"88"	11	--	Define external procedure reference number
"89"	3	2	Reference external procedure
"8A"	3	2	Reference external variable
"8B"	3	2	Relocate program address
"8C"	3	2	Relocate stack address
"8F"	14	--	Define self compiler external variable reference number
"90"	11	--	Define self compiler main procedure entry point

Figure 5.9a: Loader Directives

6.0 INTERNAL TRANSLATOR OPERATION

This final chapter is concerned with internal translator organization and operation. It is of interest when the compiler is being maintained or enhanced.

The routines of the compiler may be broadly classified as passive or active. The passive routines are defined as being independent of the language at hand, and could be applied to another language. The passive routines deal with the system interface I/O, utilities, symbol table, scanner, and code generation. The active routines are defined as being dependent on the language. They correspond to, and are named for, the syntax equations of Appendix L. The active routines deal with declarations, block structure, statements, expressions, and operands. This chapter discusses the passive routines first and the active routines last.

Several of the appendicies are included to illuminate translator structure. The load map of Appendix I lists all external procedures and variables giving their dynamic locations. The flag references of Appendix H show from which procedure each flag is generated; a procedure is listed more than once if it generates the flag in more than one circumstance.

The external procedure references of Appendix J show from which translator procedure each translator and system procedure (shown starred) is called. There is exactly one uncalled procedure, MPL, the main procedures. Appendix J is the inverse mapping of the external procedure references shown in the symbol table listings when the compiler itself translated.

The external variable references of Appendix K show from which translator procedure each external variable is referenced. Each variable is referenced in MPL, the main program. Once again, Appendix K is the inverse mapping of the static external references shown in the translated compiler symbol tables.

Several coding conventions are followed throughout the compiler. All external variables are defined and initialized in the main program (MPL); no use is made of static variables. I/O buffer lengths are always 1-indexed (except when the buffer is defined when they are 0-indexed). The symbol table size may exceed 32K so logical operations are substituted for arithmetic operations in symbol table addressing. All routines that return a true/false response and all flags use 0 for false (off) and 1 for true (on). Unless otherwise noted, all variables are of size word and all buffers and string text of size byte; bit variables are never used, double variables are only used during constant conversion and location counter updates.

The basic translation strategies used are one-pass, top-down parse, in memory operation, and hash-addressing of symbols.

Only a single pass is made over the source program. One-pass translation saves time, simplifies external operation, and eliminates communication between passes. However, one-pass translation complicates the processing of forward labels and requires code generation to allow for fixups.

The parse of the source program is strictly top-down. This makes it possible for the translator structure to follow the language syntax directly. Further, a top-down parse makes the generation of diagnostics simple and complete.

Translation occurs in memory. All translation context is retained in translator variables and the symbol table. In-memory operation removes dependence on external peripherals, increasing reliability, simplicity, and performance at the cost of a modest amount of additional memory.

Symbols of the target program are addressed with a hashing technique that makes symbol access time independent of symbol table size, increasing performance. Once again, this increased speed is paid for with a modest amount of additional memory.

Although the compiler is a fairly large program, the only genuine logical complexity results when a translation algorithm cannot be completed on-the-fly. This occurs with variable allocation, forward branches, forward calls, and statement type determination. Several tactics are used as a substitute for the desired ability to look ahead: the scanner can back up, the code generators can fixup previously generated code, the symbol table can link entries in chains, and when all else fails, external flags are maintained to indicate translation state.

6.1 SYSTEM INTERFACE

The compiler is entered via the main program MPL. All table size parameters are defined there (as literallys) as are all external variables. Compiler memory use is shown in Figure 6.1a. The program base (pb) and program length (pl) are extracted from the PLIB and the top of the compiler computed in stbase0 as a base address. After room is left for the stack (stacksiz bytes) and sort buffer (4 * width4 bytes) the bottom of the symbol table is established as stbase (also a base address). The symbol table continues upward for textmin0 bytes to the top of memory. (Note that the addition of memory beyond 64K is not automatically used by the symbol table because of hardware restrictions.) Memory checksums are computed in checks for the program space occupied by the compiler: from pb for pl bytes.

External variable initialization is performed in MPL. Variables which are used as static storage are set just once. Before each external procedure is translated the non-static external variables are reset, the stack preset to "55" so that stack use may be measured, and the hash table cleared. External procedures are translated until a physical EOF is encountered.

Return to the operating system is via EXIT which sets the second word of the current mark to -1 which causes a system return. (Were external labels permitted, a normal return from the main program would have been used.)

Memory integrity is checked by MEMCHECK at the end of each external procedure, and if enabled (V-toggle), before each source record is read. The checksums previously computed are validated, if enabled (M-toggle) and the unused portion of the stack determined and noted in stackuse.

Toggles state is maintained in the byte vector toggle which is initialized once per external procedure by SETTOGS (if enabled by P-toggle).

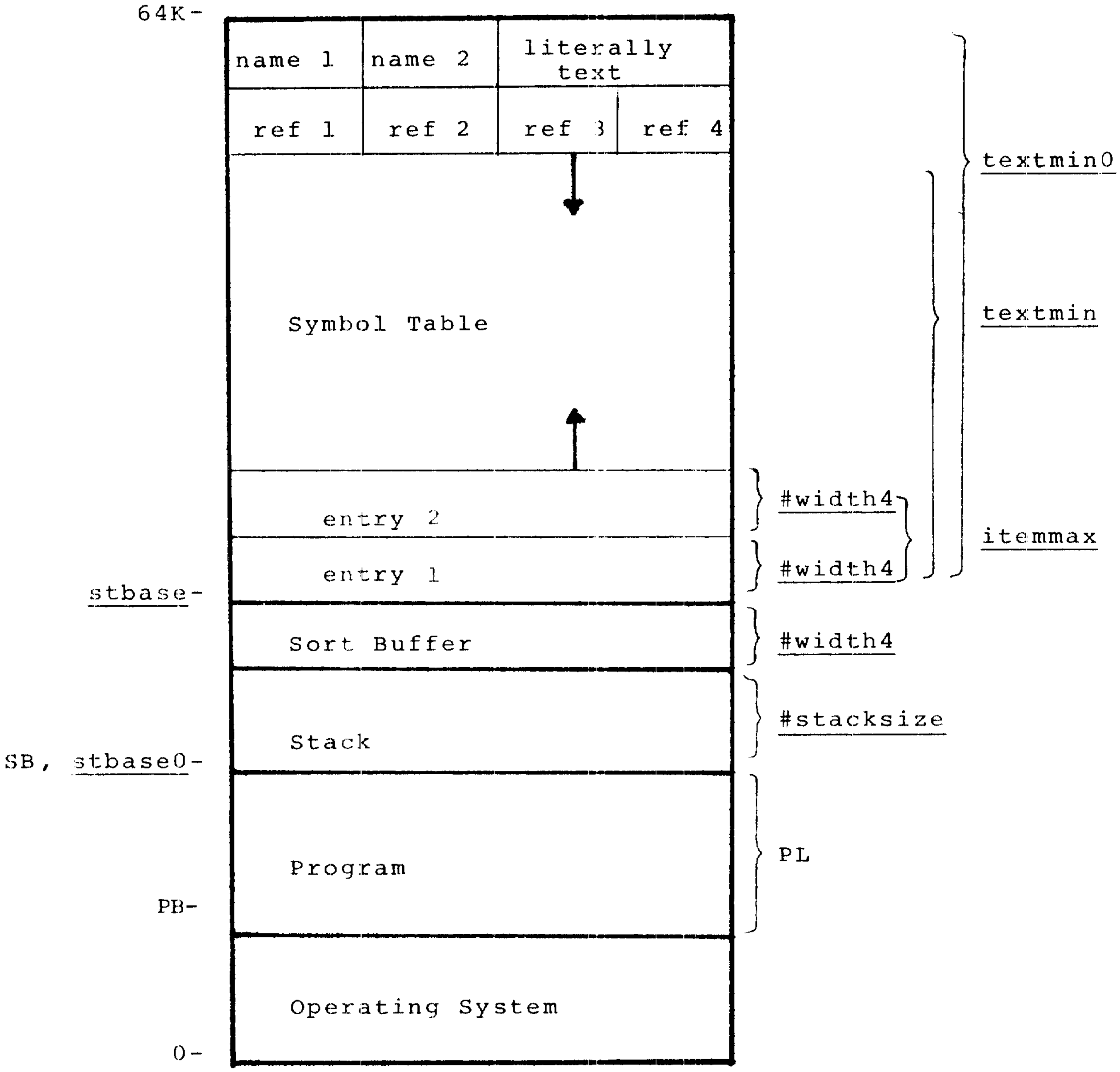


Figure 6.1a: Memory Configuration

6.2 INPUT/OUTPUT

I/O consists of physical I/O, formatted output and radix conversion.

Source input is done by SREAD. The source buffer is sobuf of size sosize; each read increments the source record counter line. The actual physical input is done by the system routine SYSGET which manages the SYSIN system file. If object in source is enabled (#-toggle), SREAD loops until a true source record is obtained. Should a physical EOF occur a normal return to the system is made if eofflag (controlled by MPL) is set, otherwise an error return is made. An EOF is legal only between external procedures. Note that object in source is processed before an EOF thus object in source may occur after the final program.

Listing output is done by PRINWRIT. The listing buffer for the annotated source is listbuf of size listsize; the first byte is used for carriage control. The actual physical output is done by the system routine SYSPUT which manages the SYSOUT system file. PRINWRIT is used for all listing functions (annotated source, generated code, etc.) and handles page titling, page numbering, forms control, and line width formatting. The current page number is maintained in page, the current line in pagelin. The type of listing line is passed to PRINWRIT so that the proper title may be used: 'A' selects the symbol table, 'Y' the summary, and anything else the source. The title skeleton is referenced through titleptr and contains the date, time, and version which are built by MPL.

Object output is done by OWRITE. The object buffer is objbuf of size objsize +1; an extra byte is added at the front (high-order position) for carriage control when the object is listed with the H-toggle. The actual physical output is done by the system routine PUT which uses a file defined by the compiler (unlike the above I/O). The working buffers are objbuf1 and objbuf2 both of size objsize+1, the FCB is objfcb. This file is opened (and closed) only if used; objecton is set when the file is opened. I/O errors are reported to the compiler which generates a compiler flag. The number of object records, even if not written, is maintained in objitem for use by SUMMARY.

The annotated source listing is constructed by LIST, an error flag listing by FLAG, the symbol table listing by LISTST, the summary listing by SUMMARY, the object listing by OWRITE, and the code listing is begun by LST-ADR, continued by the code generation routines and finished by OUT-WRT.

The listing of the annotated source is normally delayed a line to permit the level annotation to reflect translation status after line translation. The argument to LIST is a flag indicating if the listing of the next line is to be delayed. A call to LIST with the flag set is made before an error flag, the code, or the object is listed. The current state of the buffering is maintained in listwait; set means delay the next line.

The sort algoithm used by LISTST to sort the symbol table is the classic Floyd tree sort (see Reference 7); it is an in-place sort whose sort time is proportional to N*log(N), when N is the number of items.

Formatted output is aided by FORMAT which fills an output buffer according to a format specification and a list of arguments. Text is transferred directly from the format specification to the output buffer with the exception of certain reserved formatting characters. A period ('.') causes decimal conversion of the next argument into the next 5 positions of the output buffer right justified with leading blanks. A dollar sign ('$') causes hexadecimal conversion of the next argument into the next 4 positions of the output buffer right justified with leading zeroes. A pound sign ('#') inserts the next argument as text in the next 2 buffer positions. A slash ('/') inserts a carriage return/line feed in the next 2 buffer positions; this character cannot be used with the listing file as it is incompatible with the line control in PRINWRIT. Formatted output to the listing file is performed by PRINT, to the console by TYPE.

Radix conversion from internal binary to external ASCII is performed by a set of routines. The basic binary to decimal routine is B2010 which converts 32 bits to right justified form with leading zeroes; B2D10L leaves the result left justified; B2DNR leaves the result right justified with leading blanks in a specified field width. The basic binary to hexadecimal routine is BIN2HEX which converts 4 bits to a hexadecimal character; BIN2HEX2 converts 8 bits to 2 characters with leading zeroes; BIN2HEX4 converts 16 bits to 4 characters with leading zeroes.

6.3 UTILITIES

The utility routines move data, manipulate the location counters, search strings, and generate diagnostics.

A field is set from another field with MOVE, a field is set to a character with SET, and a field is set to a symbol name with MOVEST. In all cases the length of the target field is limited to the range 0 to 255 inclusive to protect against destroying program space. MOVEST uses the current symbol table entry.

The compiler maintains 3 location counters: pc (program counter), sc (stack counter), and xc (static counter). They are incremented with INC-PC, INC-SC, and INC-XC respectively; sc is decremented with DEC-SC (pc and xc cannot be decremented). The increment (or decrement) is passed to these routines in double size and all computation is performed in double precision. This ensures that any overflow in the construction of the increment or in the incrementing itself is detected. The maximum value achieved by sc(1) (the outmost block with statements) is monitored in sc1max.

INDEX provides a primitive substring capability. The prototype string is in compiler infix format (see Section 5.2.3) with items separated by periods ('.'); the target string is in symbol table string format (see Figure 6.4b). INDEX returns the ordinal item number of the target string in the prototype string, or 0 if not present.

Flags (detected errors) are generated by FLAG, fouls (consistency checks) by FOUL. The listing buffer for flags is flagbuf of size lstsize; the first byte is used for carriage control. A flag count is maintained in aborts, blunders, errors, warnings, and warningt. All warnings are counted in warningt, only unsuppressed warnings in warnings. A maximum of errmax blunders and erros is allowed. The line number of the most recent flag is maintained in lastflag. Flags are listed only if the listing is sure not to generate another flag. The offending flags are selected by the internal procedure LISTOK which currently always returns true. The processing of aborts avoids the possibility of secondary aborts by temporarily suppressing aborts (X-toggle).

6.4 SYMBOL TABLE

The symbol table occupies all of unused memory from stbase to 64K, a distance of textmin0 bytes (see Figure 6.1a). Each entry consists of a 4 * width4 (currently 16) byte fixed length part and variable-length supporting text. The fixed length entries are built from the bottom of the table up and at any moment occupy itemmax bytes. The variable length text is built from the top of the table down and at any moment has reached textmin bytes from the bottom of the table. When the fixed entries run into the variable text (itemmax exceeds textmin) the symbol table is full. All symbol table text is retained for eventual inclusion in the symbol table listing; there is no way to discard an entry.

Symbol table entries are referenced by the use of base addressing: stbase is the base address of the first entry, itembase the base address of the current entry of interest. The base addresses of successive entries differ by width4.

The fixed-length part format of a symbol table entry is shown in Figure 6.4a. The name is stored in the descriptor format shown in Figure 6.4b. The use of most of the entry fields is described in conjunction with the symbol table listing in Section 3.3.7, the remainder are described here. The entry link field is used to link variables together, hold the do level number for procedures and labels, and is a text pointer for literallys (byte address relative to stbase). The def field holds the definition line number, the ref link field a link (word index relative to stbase) to the next reference; a 0 link indicates the end of the reference chain. The spare field indicates an active base in a base chain, and an active parameter in a name list.

Of the 16 bytes in the fixed length entry, 5 are potentially unused. The high order byte of the length field is always 0 since names are restricted to 255 bytes in length. The ref link field is unused if references are suppressed (R-toggle) in which case the def field may be regarded as extraneous.

The variable-length text part of the symbol table consists of name text, literally text, and references. Text may start on any byte, references are adjusted, if necessary, to start on an even byte so that hardware word indexing works; this adjustment leaves an occasional unused byte. Name text is pointed to by the text pointer field, literally text by the entry link field (for literallys only), and references by the ref link field.

To speed symbol entry location, symbol entries are addressed through an auxillary hash table, *hashtab*, of size *hashsize*. The hash table is of fixed length; each hash table entry is a pointer to symbol table entry (as a base address); an unused hash table entry is 0. The hashing algorithm computes an initial hash table address and increment based on the symbol name. The block number (which is required to uniquely define a symbol) cannot be used as input to the hash function as it may change in the case of a forward label. The hash table length is chosen as prime so that secondary probes are guaranteed to hit each cell of the hash table. Hash table activity is monitored by noting the number of accesses, *accesses* and the number of hash table probes that result in a collison, *colisons*; the total number of probes is *accesses* + *colisons*.

Routines are provided to locate entries, add entries, put and get attributes, and compose attributes. The hash table entry of interest is always contained in *hashloc*, the symbol table entry of interest in *itembase*. Except for routines which locate an entry, the convention is followed that a routine which alters *hashloc* or *itembase* restores it.

SEARCH locates an entry of the specified name at the specified block level. *SEARCH* contains the hashing algorithm. *STATEST* searches for the specified name at all active lex levels. Both routines leave *hashloc* set to the located entry, or if none is present, to the place it will occupy; *itembase* is set only if the entry is successfully located.

Entries are added with *DEFST* and *REFST*. Using *NEWENTRY*, *itembase* is set to the next available value, *hashtab(hashloc)* to *itembase*; the text pointer, length, block and lex fields are set (the current block is used for block and lex); the number of symbol table entries, *symbols*, is incremented. Using *INSERTST* the name text is moved to the upper part of the symbol table. Should the symbol already exist *DEFST* and *REFST* simply return with *itembase* set to the desired entry.

DEFST is called when the desired symbol is the object of a definition: the symbol search occurs only at the current level. A new symbol sets the fields noted above plus def and state.

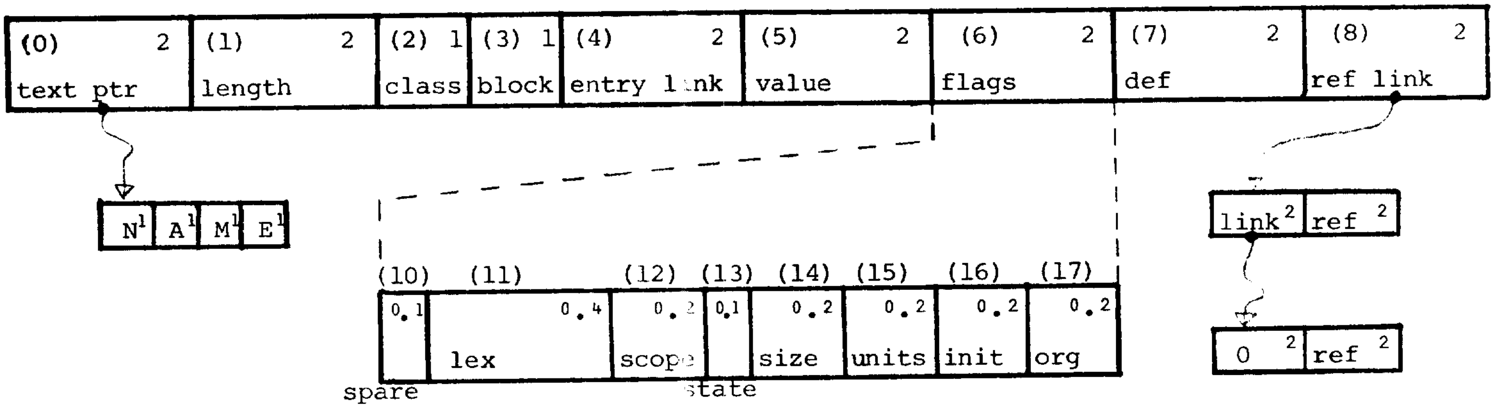
REFST is called when the desired symbol is the object of a reference: the symbol search occurs at all lex levels. A new symbol sets just the fields noted above. If the symbol is environment (checked by NAMESAME) and an old copy is found at an outer level, it is redefined at the current level. If the symbol is new and is a built-in, the definition is made in block 0 and the value, size, units, scope, state, and class fields set.

Attributes are added to a symbol with PUTST. Field codes and field values are included in Figure 6.4a. The desired entry is specified by itembase. Attributes are extracted from an entry with GETST which is symmetric with respect to PUTST except for the ref field (code 9): PUTST adds a reference to the end of the current chain, GETST returns the ref field, rather than a ref element; the ref chain must be burst manually. Each addition to a reference chain increments the total reference count, refs.

Several routines compose the attributes. BYTESIZE checks for units of byte and size of 1. LABDOBEG checks for class of label, do, or begin. GASZ, GISZ, and GPSZ translate the size and units field in different ways. PROC-ID translates the class, block, scope, and state attributes into the different procedure types. VAR-ID redefines the class attribute. GET-LEX computes the current differential lex level.

ADD-REF is a short form of the PUTST call for adding a reference to the reference chain.

Symbol table entries are linked together with ENCHAIN and the chains are burst with DECHAIN. Chains are referenced by a 2 word descriptor that points to the head and tail elements of the chain as shown in Figure 6.4c. The chains work as FIFO lists: ENCHAIN adds to the tail and DECHAIN removes from the head. Internal chain links are maintained in the entry link field as entry base addresses; 0 indicates the end of a chain. Note that an entry may be on only one chain at a time. The chains used are shown in Figure 6.4d.



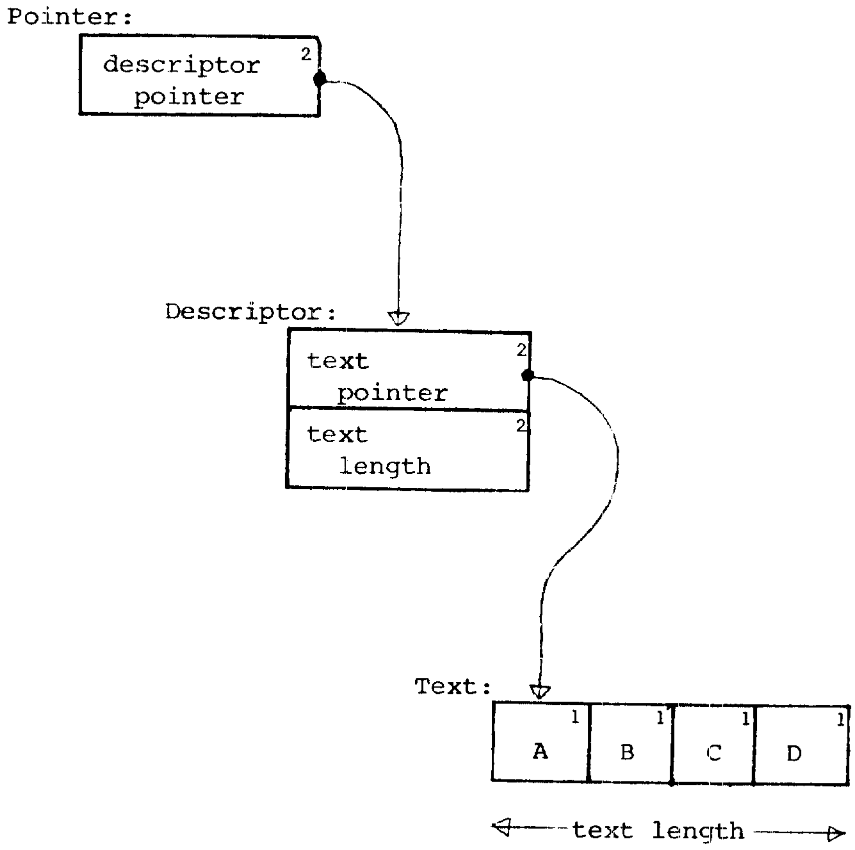
Code Interpretation

Code	Field	Class	Scope	State	Size	Units	Init	Org
0	text ptr	--	--	undefined	--	--	--	--
1	length	automatic	--	defined	1	bit	no	scalar
2	class	cbase	internal		2	byte	yes	array
3	block	vbase	external		4	ptr	--	--
4	entry link	label						
5	value	parameter						
6	flags	procedure						
7	def	main proc						
8	ref link	int proc						
9	ref	micro proc						
10	spare	builtin						
11	lex	literally						
12	scope	static						
13	state	do label						
14	size	begin label						
15	units							
16	init							
17	org							

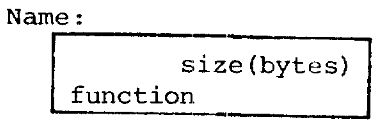
Legend:

(field code) bytes.bits
field name

Figure 6.4a: Symbol Table Entry



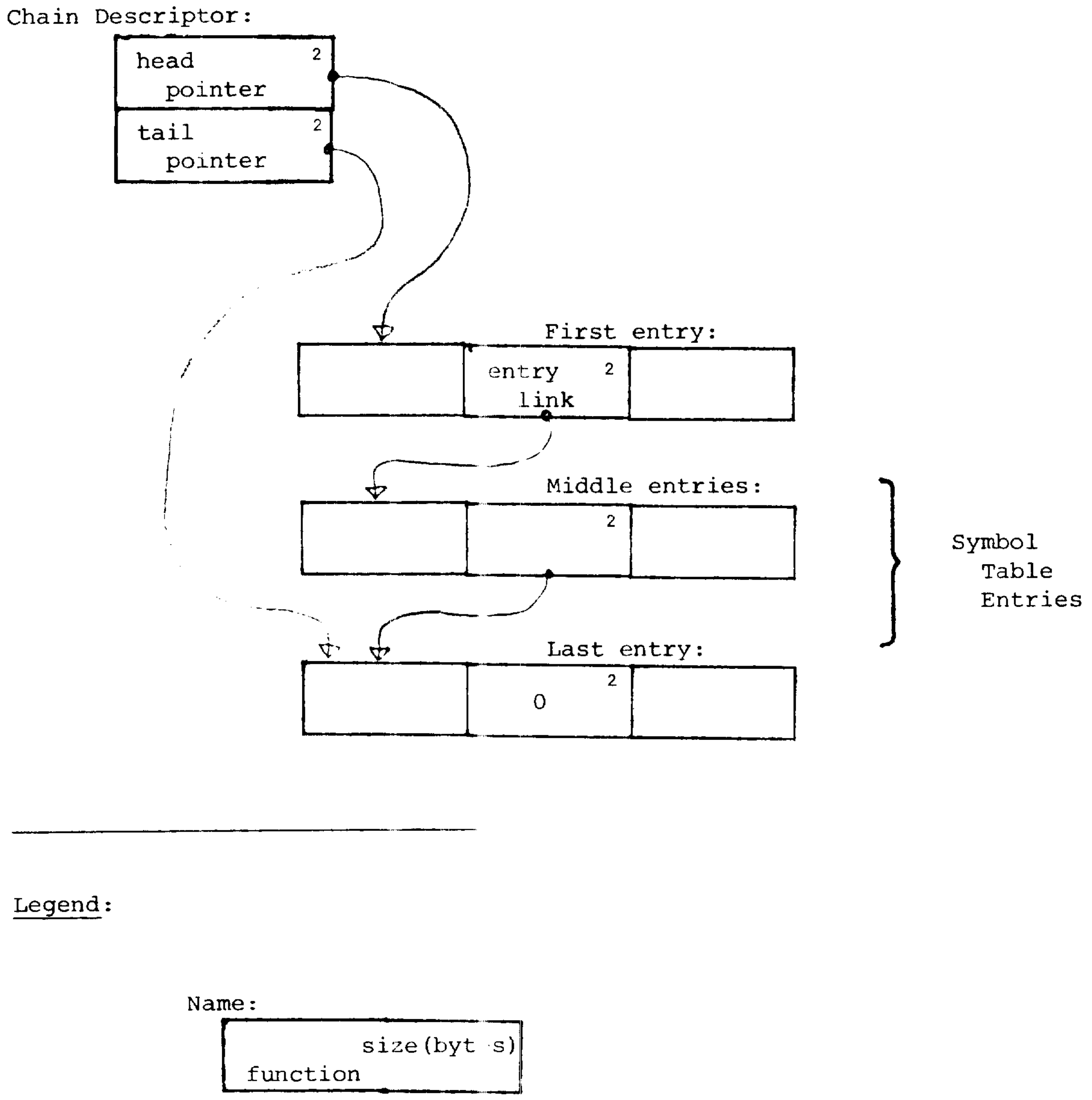
Legend:



Note:

Descriptor pointer is a SB relative byte address,
text pointer is a stbase relative byte address

Figure 6.4b: String Format



Note:
Pointers and entry links are base addresses

Figure 6.4c: Chain Format

Chain Name	Use	Reset By:	Enchained By:	Examined By:	Dechained By:
labchain	Labels in label list	LABELS	LABELS	--	BLOC, GROUP
namchain	Names in declaration name list	DCLELEMT	DCLELEMT	--	DCLELEMT
parchain	Parameters in procedure head	PROCHEAD	PROCHEAD	DCLELEMT	PROCBODY
varchain	Variables in declaration set	PROCBODY	DCLELEMT	--	PROCBODY

Figure 6.4d: Use of Chains

6.5 SCANNER

The scanner converts the source stream into tokens, the smallest unit of syntactic significance. The language at hand in context-free: a token may be identified independent of the context in which it appears; this is accomplished at the cost of reserving some identifiers for use exclusively as keywords.

The main routine of the scanner is TOKEN which returns the next token as a token code. The possible token codes are shown in Figure 6.5a.

If the token is a constant (code 1) the associated constant value is contained in value which is of double size. If the token is a symbol (code 2) the associated name text is converted directly to the descriptor string format of Figure 6.4b using pointer ptr and descriptor desc. If the token is a string (code 3) the associated string text is converted to the same descriptor-referenced string as for symbol text above; the conversion for a string, however, removes delimiting quotes and collapses internal doubled single quotes into the single quote they represent. Also the first 4 characters of a string are moved to value; if there are less than 4, characters are left justified with trailing zeroes.

To gain speed, TOKEN does a 28-way branch on the first character of a token to access the proper token processor. The 28 token cases are shown in Figure 6.5b. The special character and operator tokens present no special problems; the semicolon token (';') increments the statement counter stmts. Constants are processed by the auxillary external routine CONSTUNT which checks for 32 bit precision overflow. Strings are processed by the auxillary external routine STRING which does the quote elimination and checks for 255 character overflow. Identifier text is built by the internal routine BUILD which is optimized for speed and does case shift and checks for 255 character overflow. Identifiers are potentially (in the order of precedence) keywords, literallys, or symbols. Keyword determination is made by the internal routines KEYx.

Literally determination is done by checking the symbol table for a symbol of this name and of class literally. Should a literally be present, the literally level is incremented and TOKEN recurs. Token recursion depth is maintained in tokenlev and limited to maxlit; tokenlev differs from litlev in some instances of (illegal) literally loops. When a literally is active source text is taken from the literally text previously saved in the symbol table when the literally was defined. To mechanize literallys the literally level is maintained in litlev where level 0 is the source file. For each level litbase (litlev) contains the symbol table entry value of itembase of the active literally, litcursr(litlev) the character count of unused text, rptflag(litlev) the repeat status of the current character, and charsave(litlev) the current character itself.

The scanner provides for double token reuse: the current and previous tokens are retained for possible reuse. The reuse level is contained in reuselev. The first call to REUSE causes the next call to TOKEN to reuse the current token. The second call to REUSE, without an intervening call to TOKEN, causes the next call to TOKEN to reuse the previous token; a subsequent call to TOKEN without an intervening call to REUSE causes TOKEN to reuse the original token. Both REUSE and TOKEN protect against an illegal reuse level. The first token reuse is mechanized by swapping ptr with ptrold and tcode with tcodeold. The second reuse just increments reuselev.

Text from the source stream is distinguished as being composed of logical and physical characters. A logical character allows for leading blanks and comments and is processed by LOGCHAR; the first character of any token is a logical character. LOGCHAR processes the toggles operators. A physical character is the next character of the source stream and is processed by PHYSCHAR; the subsequent characters of most tokens are physical characters.

Physical character processing provides for single character reuse. Reuse is invoked by a call to RPTCHAR which guards against double reuse. A repeat flag vector, rptflag, is maintained for each literally level. Physical character processing also processes early source record truncation (?-toggle) and gets the next character from the source record or literally as appropriate. The current character of the source buffer, sobuf is indexed by cursor.

Several routines compose the basic scanner routines for convenience. LOOKFOR looks for the specified token and, if absent, invokes reuse. SEMI looks for a semicolon token and, if absent, generates a flag. SEMISCAN looks for a semicolon token and, if absent, generates a flag and scans tokens until a semicolon in encountered. SYMBOL reclassifies the next token as a non-identifier, new symbol, old symbol, or keyword. KONSTANT looks for a numeric constant (CONSTUNT) or string constant (STRING).

Fig 6.5a

Classes --

0	other
1	constant
2	symbol
3	string
21- 27	special characters
101-135	keywords
201-229	operators

Special Characters --

21	,
22	;
23	:
24	$
25	@
26	(
27	)

Operators --

201	+
202	-
203	↑
204	*
205	/
206	MOD
207	MULD
208	DIVD
209	SLL
210	SRA
211	SRL
212	SLC
213	<
214	<=, ↑>
215	=
216	↑=
217	>=, ↑<
218	>
219	&
220	|, !
221	XOR
222	:=
223	+=
224	LLT
225	LLE
226	LGE
227	LGT
228	LEQ
229	LNE

Keywords --

101	BASED
102	BEGIN
103	BIT
104	BY
105	BYTE
106	CALL
107	CASE
108	CONSTANT
109	DECLARE, DCL
110	DO
111	DOUBLE
112	ELSE
113	END
114	EOF
115	EXTERNAL, EXT
116	FOREVER
117	GO
118	IF
119	INTERRUPT
120	INITIAL, INIT
121	LITERALLY, LIT
122	MAIN
123	MICRO
124	POINTER
125	PROCEDURE, PROC
126	PRTNUM
127	REPEAT
128	RETURN
129	STATIC
130	THEN
131	TIMES
132	TO
133	WHILE
134	WORD
135	GOTO

Figure 6.5a: Token Codes

Fig 6.5b

Class	Case	Leading Character	Token Code
Junk	0	other	0
Single character	1	,	21
	2	;	22
	3	$	24
	4	@	25
	5	(	26
	6	)	27
	7	-	202
	8	*	204
	9	=	215
	10	&	219
	11	|, !	220
Possible double character	12	:	23, 222
	13	+	201, 223
	14	↑	203, 214, 216, 217
	15	/	205
	16	>	217, 218
	17	<	213, 214
String	18	'	3
Bit string	19	"	1
Decimal number	20	0, 1, 2, 3, 4, 5 6, 7, 8, 9	1
Known symbol	21	#, H, J, K, N, O, Q, U, V, Y, Z, _	2
Possible keyword	22	A, B, C	2, keyword
	23	D, E	2, keyword
	24	F, G, I	2, keyword
	25	L	2, keyword
	26	M, P, R	2, keyword
	27	S, T, W, X	2, keyword

Figure 6.5b: Token Cases

6.6 CODE GENERATION

Code generation is organized by instruction type with a few supporting routines. The preparation of the code listing is begun by LST-ADR continued by the various OUT-xxxx routines and finished by OUT-WRT. The listing buffer for code is codebuf of length listsize; the first byte is used for carriage control.

Loader directives are built by OLT and loader records by OUT-REC. Directives are added in binary form until the current directive overflows the buffer. A record number is added, the record checksummed, and finally the record is converted to hexadecimal form. The record is built-in buffer objbuf which is of size objsize+1; objrecn holds the record number. The current buffer position is in objx and the last in lobjx.

Pseudo-instructions are processed by OUT-ADDR (ADDR), OUT-BYTE (BYTE), OUT-WORD (WORD), and OUT-DBLE (DBLE). Instructions are processed by OUT-INST except for LDL handled by OUT-LIT, which processes all other literals as well. OUT-LWL handles the case of a LWL instruction where the literal is an address.

All these OUT-xxxx routines augment the code listing, generate loader directives, update the stack and program counters, and list the code listing. OUT-ADDR and OUT-LWL require that the address type be specified as they are used to construct addresses; OUT-BYTE, OUT-WORD, OUT-DBLE, and OUT-LIT require no address type as there is no addressing; OUT-INST determines the address type from the symbol table entry of the current symbol for those instructions that require an address.

Size conversion (from double to word or form word to double) across a binary operator is handled by A-CVT for the assignment operators and B-CVT for the expression operators. Memory referencing is handled by GET-MODE which determines the hardware mode of the memory reference and generates accessing instructions (see Subsection 5.6.3).

6.7 PROGRAM STRUCTURE

MPL programs are structured in blocks and groups. Blocks occur as procedures and begins; groups occur as repeats, and the various do's.

The current block number is maintained in block which is used to qualify all symbol names. The outermost block (which contains only the external entry point) is block 0. The block number is incremented as each new block is entered. The current lex level is contained in lex and measures the nesting depth of the current active block. The lex level starts at 0 and returns to 0 at program end. For each active lex level, sc(lex) contains the stack counter, blocklex(lex) contains the block number (which differs from block as soon as the first block is closed), and proclex(lex) contains the itembase of the name of the current block (begin blocks use the name of the innermost active procedure block). The lex level is limited to maxlex.

The current do level is maintained in donumber; the outermost group is numbered 0. The do number is incremented as each new group is entered. The current do level is contained in dolev and measures the nesting depth of the currently active group. The do level starts at 0 and returns to 0 at program end. For each active group, donest(dolev) contains the do number. The do level is limited to domax.

The external procedure block has some special properties and is processed by EXTPROC which generates the loader title, record, console log entry, and loader begin program directive. The procedure type (main, interrupt, or regular) is determined jointly by EXTPROC and PROCHEAD. Following translation of the procedure body by PROCBODY the final exit and loader directives are issued and all level counters are checked to ensure they have returned to 0.

Internal begin blocks are handled by BLOC. They are signaled by the keyword BEGIN after the label list. Any labels that were present will appear on labchain and are converted to begin labels with an associated do number for labelled end checking by END-STMT.

Procedure head processing, common to all procedures, is handled by PROCHEAD which is called from EXTPROC and BLOCKSEN. The procedure is classified as interrupt or regular, the lex level pushed by PUSH-LEX, and any parameters defined and enchained on parchain; parameters are marked as undefined and are processed further by DCLELEMT as they are declared.

When a block is entered the lex level is pushed by PUSH-LEX which is called from BLOC, EXTPROC, and PROCHEAD. The lex and block are incremented, and the sc, proclex, and blocklex vectors set. The maximum lex level achieved is maintained in lexdepth. Finally, the do level is pushed by PUSH-DO.

When a block is terminated the lex level is pulled by PULL-LEX which is called only from END-STMT. The lex level is decremented and the do level pulled by PULL-DO. The main job of PULL-LEX is to update the forward reference tables.

The forward tables are an array of structures used to store forward label and procedure references that may require later alteration. For each element fwdbase contains the itembase of the symbol, fwdlex the lex level of the point of reference, and fwdpc the program counter at the point of reference. Entries are added to the forward tables with SAVE-REF and removed with PURGEFWD. The current entry is indexed by fwdptr which is limited to fwdmax.

The action by PULL-LEX with regard to the forward tables is:

Class	State	Block	Status in New Block	Action
Procedure				Save
Label	Defined	Other		Save
		Current		Purge
	Undefined	Other		Save
		Current	Present	Convert
			Absent	Extract

A saved entry is retained for later use. A purged entry is removed and the space occupied reclaimed. A converted entry changes the reference in forward table to the symbol in the block being entered; the original entry remains in the symbol table and plays no further part in translation (but it is included in the symbol table listing). An extracted entry changes the block and lex attributes in the symbol table to the block being entered.

These actions are clarified from the language point of view. A defined label in the current block can no longer be referenced and so it is purged. A defined label in an outer block may yet be redefined in an intermediate block and so it is saved. A defined label in an inner block will already have been purged. An undefined label in an outer block is yet to be defined and so it is saved. An undefined label in an inner block will already have been extracted to the current block. An undefined label in the current block that does not appear in the new block is extracted to the new block in hopes that it will appear there. And finally, an undefined label in the current block that does appear in the new block is the same symbol and is converted to it.

When a forward label or procedure is defined, PURGEFWD is used to scan the forward tables. A matching procedure entry generates a fixup address. A matching label entry generates a fixup GOTO instruction. Non-matching entries are saved.

When a group is entered the do level is pushed by PUSH-DO which is called from GROUP and PUSH-LEX. The dolev and donumber are incremented and the donumber is saved in donest(dolev).

When a group is terminated the do level is pulled by PULL-DO which is called from END-STMT and PULL-LEX. PULL-DO decrements dolev.

If the group is a do-case the do-case item index, caseitem, and nesting level, caselev, are maintained by GROUP for use by LIST in annotating the source listing. A stack of active case indexes is maintained in casebuf (caseptr) where the latest entry is the innermost active do-case. There is a size limit on caseptr of casemax. The first case is labeled 0 and the first nest labeled 1.

Labeled end checking is performed in END-STMT and is performed only if the keyword END is followed by a symbol. If a block is being closed the lex level is pulled by PULL-LEX (which invokes PULL-DO).
If a group is being closed the do level is pulled by PULL-DO. A symbol, if present, must already be a defined label or entry point, and must match the current do number.

6.8 DECLARATIONS

The bulk of the declarations are contained in the declaration statements and are processed by DCL-STMT and PROCBODY. Entries are processed by EXTPROC or PROCHEAD which enchain the parameters on parchain. Label lists are processed by LABELS.

PROCBODY loops on DCL-STMT until a non-declaration statement is found. DCL-STMT loops on DCLELEMT until the declaration elements (separated by commas) are exhausted.

DCLELEMT uses SYMBOL to add each symbol to the symbol table as it is encountered. Only a parameter may already be present. Symbol processing is a function of symbol type. Literallys cause the literally text to be saved in the symbol table for later extraction by PHYSCHAR. External procedures, internal procedures, and micro procedures are processed without complication; the current external procedure sequence number is maintained in numberp.

Variables are enchained on namchain. Should a name list item be a parameter, it will already appear on the parameter chain, and is marked as active with the spare field. The dimension of a variable or parameter is saved in the value field; scalars have dimension 0. After namchain is built, the size and area attributes (processed by SIZEATTR , AREAATTR) are saved in the first name which is marked as such with the spare field; this first name may be on either the namchain or parchain and the attributes it contains are referred to below as common.

When all attributes are in hand the name list is reprocessed. If the common variable class is constant, namchain is burst and discarded. The common attributes are moved to the subsequent elements, the entry link field is used to hold the dimension, and the value field set. The first value will have been left in pccons by AREATTR (which is aware of any needed word alignment).

If the common variable class is not constant, variables are moved from namchain to varchain copying common attributes. A parameter is reclassified as automatic; a based variable transfers the common base to the value field. The parameter chain is scanned (and then restored), moving common attriubtes to those parameters marked as active (that is, appearing in the current name list) and an active parameter is marked as inactive.

When all declaration statements are in hand, PROCBODY performs value allocation (which must be delayed in case parameters are defined after variables). The parameter chain is burst and stack counter values assigned to the parameters. The variable chain is next burst and the dimension moved from the value field to the entry link field, and the first-in-name-list flag (spare field) reset. Automatic variables use sc(lex) for the value, word aligning non-byte initialized variables if not the first variable of a name list. Static internal variables use xc for the value word aligning all non-byte variables. Static external variables use numberd, the external data sequence number, for the value.

The extent of each variable is determined by DELTA. The function used is defined in Figure 5.2.1a.

The required size attribute is processed by SIZEATTR. A simple size is handled by SIMPLESZ, pointer and bit size directly.

The optional area attribute is processed by AREAATTR; if none is present, automatic class and internal scope is used. The external, static, and constant based area attributes are processed without complication. The variable based area attributes ensures that the base is of a legitimate class and is a backwards reference. A constant area attribute constructs a branch around the constant text in program space, if one is not already present (consflag set); two pc values are noted: consloc contains the branch location for later fixup by AREAATTR or PROCBODY, and pccons contains the starting value for address allocations for use by DCLELEMT. An initial area attribute closes an open constant branch and invokes SAVE-REF to save the SSP instruction locations in the forward tables for later fixup by PROCBODY.

Initial or constant lists are handled by INITLIST, initial or constant strings by INITSTR. The only difference between initial and constant lists is initial lists generate FILL instructions and round text to full words. Following value allocation, PROCBODY scans the forward tables and fixes up the SSP instructions generated for initial values.

6.9 STATEMENTS

Statement classification and processing follows the organization of the syntax fairly closely.

The highest level statement classification is between executable unit and an internal procedure. BLOCKSEN makes this distinction by looking for an entry point. No label or no procedure head indicates an executable unit. In the case of an executable unit, the nature of a possible label is saved in labflag for later use by LABELS. In the case of an internal procedure the attributes of the entry point are validated against any previously declared attributes.

An executable unit (handled by EXUNIT) is classified as an unconditional executable unit (UNEXUNIT) or an if statement (IF-STMT). An unconditional executable unit is further broken into blocks (BLOC), groups (GROUP), and statements (STMT); statements are classified as null (NUL-STMT), call (CAL-STMT), or assignment (AS-STMT).

An if statment (IF-STMT) is parsed slightly differently than that shown in the formal syntax. The syntax shown in Appendix L solves the dangling else problem by introducing a balanced executable unit. With a top-down parse, this mechanism is not required and the dangling else is processed directly using the syntax:

```
if-statement ::=
    if_clause executable-unit [ELSE executable unit]

if_clause ::=
    [label-list] if-then

if-then ::=
    IF exp THEN
```

The dangling else-clause is associated with the innermost unmatched then-clause.

A return statement (RET-STMT) ensures that a return value is present if and only if one was specified, and converts the return expression to the specified size.

A go to statement (GOTOSTMT) accommodates the possible scoping configurations of the go to label, defining a new symbol if the label is new, or new in the current block. An entry is added to the forward table if the label is forward, or backward and in an outer block.

A label list (which is optional) is processed by LABELS called from IF-CLAUS and UNEXUNIT. LABELS resets the label chain (labchain), enchains all labels, and generates an SSP instruction. The first label, if any, will have been saved in labflag by BLOCKSEN. An old label (satisfying a forward go to) is checked for validity, and corresponding entries in the forward table fixed up and purged. The do number of the label is inserted by BLOC or GROUP if the executable unit turns out to be a block or group; the symbol class is altered at the same time to begin-label or do-label.

The syntax alternatives are ordered so that label processing for the unconditional executable units precedes label processing for the if statement; this insures that the label chain is not reset before the do number is added.

6.10 EXPRESSIONS

Expression processing follows the formal syntax of Appendix L with little complication. The main routine is EXP and the lowest level routine for operand processing is NUM-PRI. Each operator precedence level has an associated routine that handles code generation for all operators of that precedence: S-EXP (or operators), LOG-TERM (and operator), LOG-FACT (comparison operators), NUM-EXP (add operators), NUM-TERM (multiply operators), NUM-FACT (unary operators).

An expression is either conditional (handled by C-EXP), simple (handled by S-EXP), or invokes imbedded assignment which is handled directly in EXP. In the latter case, the first operand processed by NUM-PRI is later used as a detination. The mode of this embedded destination is saved by NUM-PRI in srefmode for later use by EXP.

6.11 OPERANDS

NUM-PRI is the main routine for processing operands. A constant operand is handled directly. A procedure reference operand is handled by PROC-REF. A storage reference operand is handled by S-REF; srefmode is set for possible later use by EXP in processing imbedded assignment. An address function ('@') operand is handled in conjunction with VAR. A subexpression operand is handled by recurring on EXP. A PRTNUM function is handled directly; the procedure that is the argument to PRTNUM may be forward, backward, or external.

Procedure references are processed by PROC-REF. The calling routine (CAL-STMT for procedure invocation, NUM-PRI for function invocation) is specified so that the presence of a return value is known. The procedure may be forward, built-in, etc. Built-in procedures generate in-line code and require the correct number and size of arguments.

Storage references are processed by S-REF which calls REF which calls VAR. S-REF handles field selection, REF indirection, and VAR indexing. The resultant reference type is encoded by each routine.

IN-SYM is used by PROC-REF, VAR, NUM-PRI, and GROUP to check the symbol table for a defined symbol when a symbol is referenced as an operand. An undeclared symbol is declared as word automatic with array organization if followed by a left parenthesis ('(').

6.12 SAMPLE MODIFICATIONS

The scope of several natural compiler modifications is outlined in this section.

- Default Toggle Setting Change: The default toggle settings are contained in SETTOGS. A change is made by altering the default settings therein.

- Built-in Function Addition: Built-in functions are recognized in REFST. The name of the new built-in is added to the string built-ins and the symbol attributes to the corresponding position in the arrays values, sizes, and units. Code generation occurs in PROC-REF. The opcode generated is added to biop and the code generation attributes to biargsz, bi#args, and biretsz. Any special code generation is dealt with by program logic.

- Operator Addition: Operators are recognized in TOKEN. A new keyword operator is added to the appropriate KEYx internal procedure. A new symbolic operator is recognized by program logic driven by the cases table. In either case a new token code is returned for use by the routine of the expression processor corresponding to the desired precedence level.

- Speed Enhancement: The basic translation strategies make any significant speed improvement unlikely. Further, speed improvements will be masked by I/O times in most circumstances. Nevertheless, fine-tuning the scanner is the obvious candidate for speed improvement. The blank (and comment) scanning circuitry is the most heavily used loop in the compiler. It comprises the first loop in LOGCHAR and the first IF in PHYSCHAR.

- Symbol Table Size Increase: Short of symbol table entry format alteration or compiler size reduction (see below), a bigger symbol table requires more real memory and altered symbol table addressing. The simplest approach is to place the additional real memory in consecutive modules in bank 1 and alter the memory size determination in MPL that comprises the computation of stbase0, stbase, and textmin0.

- Memory Size Reduction: A 32K byte compiler is feasible with unaltered translation algorithms if most listing features are sacrificed and a minimal fixed I/O system is introduced. The symbol table listing (LISTST) and program summary (SUMMARY) would be eliminated entirely. The program listing would be converted to a simple source listing with coded flags; this effectively eliminates FLAG, FORMAT, LIST, PRINT, and PRINWRIT, and the mnemonic tables from OUT-INST. All other extraneous features would be sacrificed as well: memory and stack checking (MEMCHECK), hash addressing of symbol table, cross reference fields in symbol table entry, etc. The resulting compiler would use memory roughly as follows:

Item	32K Compiler	64K Compiler
Code	23K	34K
Operating System	3	16
Stack	4	6
Symbol Table	2	8

APPENDICIES

APPENDIX A: BIBLIOGRAPHY

1) Current MPL Language: "MPL Language Reference Manual", SMPL-1, September 1975.

2) Earlier MPL Language: "Microdata 32/S Programming Language Reference Manual (MPL)", PUMPL-2, November 1973.

3) Tutorial on MPL Language: "Introduction to Microdata Programming Language (MPL)", 1975.

4) Disk Operating System: "Genasys/D", July 21, 1975.

5) Implementation Operating System: "Genasys", July 16, 1975.

6) 32/S Processor: "Microdata 32/S Computer Reference Manual", RM 20003250, January 1975.

7) Sort Algorithm: "Communications of the ACM", Volume 7 Number 12, December 1964, page 701.

```
      CLASS NUMBER SEVERITY             DESCRIPTION
      ----- ------ --------   ---------------------------------


,/*SPARE*/   M0  # 'W '
,/*MISS */   M1  # 'A NO ENTRY NAME, JOB ABORTED'
,/*RANGE*/   M2  # 'E STRING OVER 255 BYTES, TAIL IGNORED'
,/*MISS */   M3  # 'E NO RIGHT PAREN, ASSUMED'
,/*RANGE*/   M4  # 'E BIT STRING RADIX NOT 1 2 3 4, IGNORED'

,/*OTHER*/   M5  # 'E BAD BIT STRING DIGIT, IGNORED'
,/*MISS */   M6  # 'E NO COLON, ASSUMED'
,/*ATTR */   M7  # 'E SYMBOL ALREADY DEFINED, SEE SYMBOL TABLE'
,/*RANGE*/   M8  # 'W SYMBOL OVER 255 BYTES, TAIL IGNORED'
,/*OTHER*/   M9  # 'W '';'' IN COMMENT, IGNORED'

,/*RANGE*/  M10  # 'B DELTA LEX OVER 15, 0 USED'
,/*ATTR */  M11  # 'B SUBSCRIPT ON NON-DIMENSIONED VARIABLE, USED ANYWAY'
,/*ATTR */  M12  # 'W INDIRECTION ON NON PTR TO VARIABLE, PTR TO WORD ASSUMED'
,/*MISS */  M13  # 'B NO SUBSCRIPT EXPRESSION, SKIPPED'
,/*ATTR */  M14  # 'B FIELD SELECT NEEDS WORD VARIABLE, ASSUMED'

,/*MISS */  M15  # 'E NO LEFT PAREN, ASSUMED'
,/*MISS */  M16  # 'B NO FIELD SELECT EXPRESSION, SKIPPED'
,/*MISS */  M17  # 'B ''@'' NOT FOLLOWED BY STORAGE REF, SKIPPED'
,/*MISS */  M18  # 'B NO EXPRESSION, SKIPPED'
,/*RANGE*/  M19  # 'W CHAR STRING OVER 4 CHARS, TAIL IGNORED'

,/*MISS */  M20  # 'B NO SYMBOL, SKIPPED'
,/*ATTR */  M21  # 'B PRTNUM SYMBOL NOT A PROC NAME, SKIPPED'
,/*OTHER*/  M22  # 'B JUNK IN STATEMENT, TEXT SKIPPED TO NEXT '';'''
,/*MISS */  M23  # 'B NO EXPRESSION OPERAND, SKIPPED'
,/*RANGE*/  M24  # 'B PROGRAM OVER 65535, REDUCED MOD 2**16'

,/*RANGE*/  M25  # 'B STACK OVER 65535, REDUCED MOD 2**16'
,/*RANGE*/  M26  # 'E CONSTANT OVER 32 BITS, REDUCED MOD 2**32'
,/*MISS */  M27  # 'B NO ''THEN'', ASSUMED'
,/*MISS */  M28  # 'B NO ''ELSE'', ASSUMED'
,/*RANGE*/  M29  # 'A 50 BLUNDERS AND ERRORS, JOB ABORTED'

,/*OTHER*/  M30  # 'A MEMORY BAD, JOB ABORTED'
,/*OTHER*/  M31  # 'A SYMBOL TABLE FULL, JOB ABORTED'
,/*RANGE*/  M32  # 'A 256 BLOCKS, JOB ABORTED'
,/*RANGE*/  M33  # 'A 16 LEX LEVELS, JOB ABORTED'
,/*RANGE*/  M34  # 'A TOO MANY NESTED LITERALLY''S, JOB ABORTED'

,/*SPARE*/  M35  # 'W '
,/*OTHER*/  M36  # 'A STACK OVERFLOW, JOB ABORTED'
,/*I/O  */  M37  # 'A SOURCE PHYSICAL EOF, JOB ABORTED'
,/*OTHER*/  M38  # 'A HASH TABLE FULL, JOB ABORTED'
,/*OTHER*/  M39  # 'W HASH TABLE 90% FULL, IGNORED'

,/*I/O  */  M40  # 'A OBJECT IO ERROR, JOB ABORTED'
,/*MISS */  M41  # 'B NO ASSIGNMENT OPERATOR (=, +=, :=), IGNORED'
,/*ATTR */  M42  # 'B OPERATOR NEEDS WORD DESTINATION, IGNORED'
,/*ATTR */  M43  # 'B UNDECLARED VARIABLE, DECLARED AS WORD AUTO'
,/*RANGE*/  M44  # 'B TOO MANY FORWARD REFS TO PROCS/LABELS/INITLISTS, SKIPPED'

,/*ATTR */  M45  # 'E KEYWORD DEFINED AS SYMBOL, DEFINED BUT NO REFS POSSIBLE'
,/*MISS */  M46  # 'W NO ''PROC'', ASSUMED'
,/*MISS */  M47  # 'W NO '';'', ASSUMED'
,/*MISS */  M48  # 'B NO PARAMETER, SKIPPED'
,/*SPARE*/  M49  # 'W '
```

```
      CLASS NUMBER SEVERITY              DESCRIPTION
      ----- ------ --------    ----------------------------------

  ,/*MISS */   M50 # 'E NO EXTERNAL PROCHEAD, ''MAIN'' USED'
  ,/*MISS */   M51 # 'W NO DECLARATION LIST ELEMENT, SKIPPED'
  ,/*MISS */   M52 # 'W NO NAME LIST ITEM, SKIPPED'
  ,/*MISS */   M53 # 'E NO CONSTANT, SKIPPED'
  ,/*MISS */   M54 # 'E NO STRING, SKIPPED'

  ,/*MISS */   M55 # 'E NO INITIAL OR CONSTANT LIST ITEM, 0 USED'
  ,/*ATTR */   M56 # 'E PARAMETER NOT FULLY DECLARED, WORD USED'
  ,/*ATTR */   M57 # 'E INTERRUPT PROC DECLARED WITH LEX>1, USED AS DECLARED'
  ,/*ATTR */   M58 # 'E UNSATISFIED FORWARD LABELS OR PROCS, SEE SYMBOL TABLE'
  ,/*OTHER*/   M59 # 'E JUNK IN DCL STMT, TEXT SKIPPED TO NEXT '','' OR '';''

  ,/*RANGE*/   M60 # 'W EXTERNAL SYMBOL OVER 8 CHARS, TAIL IGNORED'
  ,/*ATTR */   M61 # 'E STRING NEEDS BYTE SIZE, TREATED AS BYTE'
  ,/*ATTR */   M62 # 'W STRING LENGTH DOES NOT MATCH DIMENSION, IGNORED'
  ,/*ATTR */   M63 # 'W INITIAL LIST TOO LONG, USED ANYWAY'
  ,/*ATTR */   M64 # 'W CONSTANT LIST TOO LONG OR SHORT, USED ANYWAY'

  ,/*ATTR */   M65 # 'E BASE VARIABLE MUST BE BACKWARD REF, CBASE=0 USED'
  ,/*ATTR */   M66 # 'B BASE NOT SCALAR STORAGE REF, USED ANYWAY'
  ,/*ATTR */   M67 # 'E BAD SIZE, WORD USED'
  ,/*ATTR */   M68 # 'E BAD BIT SIZE, BIT(4) USED'
  ,/*RANGE*/   M69 # 'W INITIAL OR CONSTANT VALUE TOO BIG, TRUNCATED ON LEFT'

  ,/*ATTR */   M70 # 'B PARAMETER MAY NOT HAVE AREA ATTRIBUTE, SEE SYMBOL TABLE'
  ,/*SPARE*/   M71 # 'W '
  ,/*ATTR */   M72 # 'E BASE MISSING, CBASE=0 USED'
  ,/*ATTR */   M73 # 'E PROC SIZE MISMATCH, NEW SIZE USED'
  ,/*OTHER*/   M74 # 'B NO ''='' IN ITERATIVE DO, ASSUMED'

  ,/*RANGE*/   M75 # 'E CONSTANT OVER 16 BITS, REDUCED MOD 2**16'
  ,/*OTHER*/   M76 # 'W DO''S NESTED TOO DEEP, NEST CHECKING INVALID'
  ,/*OTHER*/   M77 # 'B TOO MANY DO CASES, IGNORED'
  ,/*MISS */   M78 # 'B NO ''TO'' IN ITERATIVE DO, ASSUMED'
  ,/*SPARE*/   M79 # 'W '

  ,/*ATTR */   M80 # 'B BAD ''BASED'' NESTING, IGNORED'
  ,/*ATTR */   M81 # 'B VARIABLE MUST BE STORAGE REF, IGNORED'
  ,/*OTHER*/   M82 # 'E END IDENTIFIER MUST BE DO-LABEL/BEGIN-LABEL/ENTRY, IGNORED'
  ,/*OTHER*/   M83 # 'E LABEL ON END DOES NOT MATCH CURRENT BLOCK, IGNORED'
  ,/*OTHER*/   M84 # 'E LABEL ON END NOT DEFINED, IGNORED'

  ,/*OTHER*/   M85 # 'W OVER 32765 DO BLOCKS, NEST CHECKING INVALID'
  ,/*RANGE*/   M86 # 'W STACK UNDERFLOW (MAY BE DUE TO ANOTHER FLAG), IGNORED'
  ,/*MISS */   M87 # 'B NO PROC REF AFTER ''CALL'', IGNORED'
  ,/*MISS */   M88 # 'B NO LABEL REF AFTER ''GO TO'', IGNORED'
  ,/*ATTR */   M89 # 'B CURRENT PROC CANNOT RETURN A VALUE, IGNORED'

  ,/*ATTR */   M90 # 'B CURRENT PROC MUST RETURN A VALUE, IGNORED'
  ,/*MISS */   M91 # 'B NO STATEMENT AFTER ''THEN'', IGNORED'
  ,/*MISS */   M92 # 'B NO STATEMENT AFTER ''ELSE'', IGNORED'
  ,/*OTHER*/   M93 # 'B EMBEDDED ASSIGNMENT DOES NOT ALLOW FIELD SELECT, IGNORED'
  ,/*MISS */   M94 # 'B NO PROC ARG, IGNORED'

  ,/*MISS */   M95 # 'B BAD NUMBER OF ARGS TO BUILTIN PROC, ACCEPTED'
  ,/*MISS */   M96 # 'B BAD ARG SIZE FOR BUILTIN PROC, ACCEPTED'
  ,/*MISS */   M97 # 'B OVER 252 WORDS OF PROC ARGS, REDUCED MOD 252'
  ,/*MISS */   M98 # 'B ''@'' ILLEGAL WITH BASED VARIABLE OR CONSTANT, IGNORED'
  ,/*MISS */   M99 # 'B CONSTANT CANNOT BE A DESTINATION, USED ANYWAY'
```

```
           CLASS NUMBER SEVERITY           DESCRIPTION
           ----- ------ --------   ---------------------------------

                      W A R N I N G S

,/*SPARE*/   M0 # 'W '
,/*RANGE*/   M8 # 'W SYMBOL OVER 255 BYTES, TAIL IGNORED'
,/*OTHER*/   M9 # 'W '';'' IN COMMENT, IGNORED'
,/*ATTR */  M12 # 'W INDIRECTION ON NON PTR TO VARIABLE, PTR TO WORD ASSUMED'
,/*RANGE*/  M19 # 'W CHAR STRING OVER 4 CHARS, TAIL IGNORED'

,/*SPARE*/  M35 # 'W '
,/*OTHER*/  M39 # 'W HASH TABLE 90% FULL, IGNORED'
,/*MISS */  M46 # 'W NO ''PROC'', ASSUMED'
,/*MISS */  M47 # 'W NO '';'', ASSUMED'
,/*SPARE*/  M49 # 'W '

,/*MISS */  M51 # 'W NO DECLARATION LIST ELEMENT, SKIPPED'
,/*MISS */  M52 # 'W NO NAME LIST ITEM, SKIPPED'
,/*RANGE*/  M60 # 'W EXTERNAL SYMBOL OVER 8 CHARS, TAIL IGNORED'
,/*ATTR */  M62 # 'W STRING LENGTH DOES NOT MATCH DIMENSION, IGNORED'
,/*ATTR */  M63 # 'W INITIAL LIST TOO LONG, USED ANYWAY'

,/*ATTR */  M64 # 'W CONSTANT LIST TOO LONG OR SHORT, USED ANYWAY'
,/*RANGE*/  M69 # 'W INITIAL OR CONSTANT VALUE TOO BIG, TRUNCATED ON LEFT'
,/*SPARE*/  M71 # 'W '
,/*OTHER*/  M76 # 'W DO''S NESTED TOO DEEP, NEST CHECKING INVALID'
,/*SPARE*/  M79 # 'W '

,/*OTHER*/  M85 # 'W OVER 32765 DO BLOCKS, NEST CHECKING INVALID'
,/*RANGE*/  M86 # 'W STACK UNDERFLOW (MAY BE DUE TO ANOTHER FLAG), IGNORED'


                      E R R O R S

,/*RANGE*/   M2 # 'E STRING OVER 255 BYTES, TAIL IGNORED'
,/*MISS */   M3 # 'E NO RIGHT PAREN, ASSUMED'
,/*RANGE*/   M4 # 'E BIT STRING RADIX NOT 1 2 3 4, IGNORED'
,/*OTHER*/   M5 # 'E BAD BIT STRING DIGIT, IGNORED'
,/*MISS */   M6 # 'E NO COLON, ASSUMED'

,/*ATTR */   M7 # 'E SYMBOL ALREADY DEFINED, SEE SYMBOL TABLE'
,/*MISS */  M15 # 'E NO LEFT PAREN, ASSUMED'
,/*RANGE*/  M26 # 'E CONSTANT OVER 32 BITS, REDUCED MOD 2**32'
,/*ATTR */  M45 # 'E KEYWORD DEFINED AS SYMBOL, DEFINED BUT NO REFS POSSIBLE'
,/*MISS */  M50 # 'E NO EXTERNAL PROCHEAD, ''MAIN'' USED'

,/*MISS */  M53 # 'E NO CONSTANT, SKIPPED'
,/*MISS */  M54 # 'E NO STRING, SKIPPED'
,/*MISS */  M55 # 'E NO INITIAL OR CONSTANT LIST ITEM, 0 USED'
,/*ATTR */  M56 # 'E PARAMETER NOT FULLY DECLARED, WORD USED'
,/*ATTR */  M57 # 'E INTERRUPT PROC DECLARED WITH LEX>1, USED AS DECLARED'

,/*ATTR */  M58 # 'E UNSATISFIED FORWARD LABELS OR PROCS, SEE SYMBOL TABLE'
,/*OTHER*/  M59 # 'E JUNK IN DCL STMT, TEXT SKIPPED TO NEXT '','' OR '';'''
,/*ATTR */  M61 # 'E STRING NEEDS BYTE SIZE, TREATED AS BYTE'
,/*ATTR */  M65 # 'E BASE VARIABLE MUST BE BACKWARD REF, CBASE=0 USED'
,/*ATTR */  M67 # 'E BAD SIZE, WORD USED'

,/*ATTR */  M68 # 'E BAD BIT SIZE, BIT(4) USED'
,/*ATTR */  M72 # 'E BASE MISSING, CBASE=0 USED'
,/*ATTR */  M73 # 'E PROC SIZE MISMATCH, NEW SIZE USED'
,/*RANGE*/  M75 # 'E CONSTANT OVER 16 BITS, REDUCED MOD 2**16'
,/*OTHER*/  M82 # 'E END IDENTIFIER MUST BE DO-LABEL/BEGIN-LABEL/ENTRY, IGNORED'

,/*OTHER*/  M83 # 'E LABEL ON END DOES NOT MATCH CURRENT BLOCK, IGNORED'
,/*OTHER*/  M84 # 'E LABEL ON END NOT DEFINED, IGNORED'
```

```
    CLASS NUMBER SEVERITY            DESCRIPTION
    ----- ------ --------   ---------------------------------

                   B L U N D E R S

,/*RANGE*/   M10 # ´B DELTA LEX OVER 15, 0 USED´
,/*ATTR */   M11 # ´B SUBSCRIPT ON NON-DIMENSIONED VARIABLE, USED ANYWAY´
,/*MISS */   M13 # ´B NO SUBSCRIPT EXPRESSION, SKIPPED´
,/*ATTR */   M14 # ´B FIELD SELECT NEEDS WORD VARIABLE, ASSUMED´
,/*MISS */   M16 # ´B NO FIELD SELECT EXPRESSION, SKIPPED´

,/*MISS */   M17 # ´B ´´@´´ NOT FOLLOWED BY STORAGE REF, SKIPPED´
,/*MISS */   M18 # ´B NO EXPRESSION, SKIPPED´
,/*MISS */   M20 # ´B NO SYMBOL, SKIPPED´
,/*ATTR */   M21 # ´B PRTNUM SYMBOL NOT A PROC NAME, SKIPPED´
,/*OTHER*/   M22 # ´B JUNK IN STATEMENT, TEXT SKIPPED TO NEXT ´´;´´´

,/*MISS */   M23 # ´B NO EXPRESSION OPERAND, SKIPPED´
,/*RANGE*/   M24 # ´B PROGRAM OVER 65535, REDUCED MOD 2**16´
,/*RANGE*/   M25 # ´B STACK OVER 65535, REDUCED MOD 2**16´
,/*MISS */   M27 # ´B NO ´´THEN´´, ASSUMED´
,/*MISS */   M28 # ´B NO ´´ELSE´´, ASSUMED´

,/*MISS */   M41 # ´B NO ASSIGNMENT OPERATOR (=, +=, :=), IGNORED´
,/*ATTR */   M42 # ´B OPERATOR NEEDS WORD DESTINATION, IGNORED´
,/*ATTR */   M43 # ´B UNDECLARED VARIABLE, DECLARED AS WORD AUTO´
,/*RANGE*/   M44 # ´B TOO MANY FORWARD REFS TO PROCS/LABELS/INITLISTS, SKIPPED´
,/*MISS */   M48 # ´B NO PARAMETER, SKIPPED´

,/*ATTR */   M66 # ´B BASE NOT SCALAR STORAGE REF, USED ANYWAY´
,/*ATTR */   M70 # ´B PARAMETER MAY NOT HAVE AREA ATTRIBUTE, SEE SYMBOL TABLE´
,/*OTHER*/   M74 # ´B NO ´´=´´ IN ITERATIVE DO, ASSUMED´
,/*OTHER*/   M77 # ´B TOO MANY DO CASES, IGNORED´
,/*MISS */   M78 # ´B NO ´´TO´´ IN ITERATIVE DO, ASSUMED´

,/*ATTR */   M80 # ´B BAD ´´BASED´´ NESTING, IGNORED´
,/*ATTR */   M81 # ´B VARIABLE MUST BE STORAGE REF, IGNORED´
,/*MISS */   M87 # ´B NO PROC REF AFTER ´´CALL´´, IGNORED´
,/*MISS */   M88 # ´B NO LABEL REF AFTER ´´GO TO´´, IGNORED´
,/*ATTR */   M89 # ´B CURRENT PROC CANNOT RETURN A VALUE, IGNORED´

,/*ATTR */   M90 # ´B CURRENT PROC MUST RETURN A VALUE, IGNORED´
,/*MISS */   M91 # ´B NO STATEMENT AFTER ´´THEN´´, IGNORED´
,/*MISS */   M92 # ´B NO STATEMENT AFTER ´´ELSE´´, IGNORED´
,/*OTHER*/   M93 # ´B EMBEDDED ASSIGNMENT DOES NOT ALLOW FIELD SELECT, IGNORED´
,/*MISS */   M94 # ´B NO PROC ARG, IGNORED´

,/*MISS */   M95 # ´B BAD NUMBER OF ARGS TO BUILTIN PROC, ACCEPTED´
,/*MISS */   M96 # ´B BAD ARG SIZE FOR BUILTIN PROC, ACCEPTED´
,/*MISS */   M97 # ´B OVER 252 WORDS OF PROC ARGS, REDUCED MOD 252´
,/*MISS */   M98 # ´B ´´@´´ ILLEGAL WITH BASED VARIABLE OR CONSTANT, IGNORED´
,/*MISS */   M99 # ´B CONSTANT CANNOT BE A DESTINATION, USED ANYWAY´


                   A B O R T S

,/*MISS */    M1 # ´A NO ENTRY NAME, JOB ABORTED´
,/*RANGE*/   M29 # ´A 50 BLUNDERS AND ERRORS, JOB ABORTED´
,/*OTHER*/   M30 # ´A MEMORY BAD, JOB ABORTED´
,/*OTHER*/   M31 # ´A SYMBOL TABLE FULL, JOB ABORTED´
,/*RANGE*/   M32 # ´A 256 BLOCKS, JOB ABORTED´

,/*RANGE*/   M33 # ´A 16 LEX LEVELS, JOB ABORTED´
,/*RANGE*/   M34 # ´A TOO MANY NESTED LITERALLY´´S, JOB ABORTED´
,/*OTHER*/   M36 # ´A STACK OVERFLOW, JOB ABORTED´
,/*I/O  */   M37 # ´A SOURCE PHYSICAL EOF, JOB ABORTED´
,/*OTHER*/   M38 # ´A HASH TABLE FULL, JOB ABORTED´

,/*I/O  */   M40 # ´A OBJECT IO ERROR, JOB ABORTED´
```

APPENDIX D: FLAGS -- DETAILED DESCRIPTION

0 (Warning) --

Spare

1 (Abort): No Entry Name, Job Aborted --

No entry name was found for the external procedure. The cursor indicates where the entry name was expected. The job is aborted. If the abort is suppressed the compiler will not function correctly.

2 (Error): String Over 255 Bytes, Tail Ignored --

The body of a string, not including the delimiting quotes and after collapsing consecutive single quotes into the single quotes they represent, exceeds 255 bytes. The cursor points the last valid byte. Bytes beyond the 255th are ignored and compilation continues. This flag is usually caused by a missing trailing quote.

3 (Error): No Right Paren, Assumed --

A right parenthesis (')') is required at the point indicated by the cursor but was not found. Its presence is assumed and compilation continues

4 (Error): Bit String Radix Not 1 2 3 or 4, Ignored --

The radix, indicated by the cursor, specified for a bit string constant is not 1 (binary), 2 (quaternary), 3 (octal), or 4 (hexidecimal). The current radix remains in use and compilation continues.

5 (Error): Bad Bit String Digit, Ignored --

The current digit, indicated by the cursor, in a bit string constant is not in the range 0 to <current radix -1>. The offending digit is ignored and compilation continues.

6 (Error): No Colon, Assumed --

A colon (':') is required at the point indicated by the cursor but was not found. Its presence is assumed and compilation continues.

7 (Error): Symbol Already Defined, See Symbol Table --

The symbol indicated by the cursor was previously defined in the same scope. The attributes given to the symbol are a mixture of the attributes of the multiple definitions as shown in the symbol table listing. Compilation continues and secondary flags can be expected.

8 (Warning): Symbol Over 255 Bytes, Tail Ignored --

The name of a symbol exceeds 255 bytes. The cursor indicates the last valid byte. Bytes beyond the 255th are ignored and compilation continues.

9 (Warning): ';' in Comment, Ignored --

A semicolon (';') is present in a comment as shown by the cursor. It is ignored and compilation continues. Usually this warning is caused by a missing close comment operator ('*/'). This warning does not apply to a question mark comment (see Appendix F).

10 (Blunder): Delta Lex Over 15, 0 Used --

A lex level difference of more than 15 is necessary at the point indicated by the cursor to generate code. A delta lex of 0 is used and compilation continues. (The current compiler will generate Abort 33 before Blunder 10).

11 (Blunder): Subscript on Nondimensioned Variable, Used Anyway --

A subscript, indicated by the cursor, modifies a variable that was not dimensioned. Code is generated as if the variable was dimensioned and compilation continues.

12 (Warning): Indirection on Non Ptr to Variable, Ptr to Word Assumed --

The indirection operator ('@'), indicated by the cursor, has been applied to a variable that was not declared as a pointer. A size of POINTER TO WORD is assumed and compilation continues.

13 (Blunder): No Subscript Expression, Skipped --

A subscript expression is required at the point indicated by the cursor. Subscript code generation is skipped and compilation continues.

14 (Blunder): Field Select Needs Word Variable, Assumed --

The field select operator '$'), indicated by the cursor, has been applied to a variable not declared as a word. Code is generated as if word was declared and compilation continues.

15 (Error): No Left Paren, Assumed --

A left parenthesis ('(') is required as the point indicated by the cursor. Its presence is assumed and compilation continues.

16 (Blunder): No Field Select Expression, Skipped --

A field select expression is required at the point indicated by the cursor but is missing. Code generation for the field select expression is skipped and compilation continues.

17 (Blunder): '@' Not Followed By Storage Ref, Skipped --

The address operator ('@') indicated by the cursor is not followed by a storage reference. Code generation for the address operator is skipped and compilation continues.

18 (Blunder): No Expression, Skipped --

An expression is required at the point indicated by the cursor but is missing. Code generation for the expression is skipped and compilation continues. This flag is a catch-all when the exact nature of the missing expression is not diagnosed.

19 (Warning): Char String Over 4 Chars, Tail Ignored --

A character string used as a constant contains more than 4 characters. The cursor points to the last character of the offending string. Characters beyond the 4th are ignored and compilation continues.

20 (Blunder): No Symbol, Skipped --

A symbol is required at the point indicated by the cursor but is missing. Code generation for the symbol is skipped and compilation continues.

21 (Blunder): PRTNUM Symbol Not a Proc Name, Skipped --

The symbol, indicated by the cursor, is the object of the PRTNUM operator but is not a procedure name. Code generation for the PRTNUM operation is skipped and compilation continues.

22 (Blunder): Junk in Statement, Text Skipped to Next ';' --

A statement cannot be decoded at the point indicated by the cursor. Source text is skipped until the next delimiting semicolon (';') is found. (Semicolons appearing in a comment or string constant are skipped as well.) This flag is a catch-all when a more specific diagnostic is not provided for a faulty executable statement.

23 (Blunder): No Expression Operand, Skipped --

An operand (either a variable or constant) is required at the point indicated by the cursor but is missing. Code generation for the operand is skipped and compilation continues.

24 (Blunder): Program Over 65535, Reduced Mod 2**16 --

The location counter for program space (program counter) is over 65,535 bytes as of the point indicated by the cursor. Overflow in the program counter beyond 16 bits is ignored and compilation continues. A common program counter is used for all procedures nested in an external procedure.

25 (Blunder): Stack Over 65535, Reduced Mod 2**16 --

The location counter for stack space (stack counter) is over 65,535 bytes as of the point indicated by the cursor. Overflow in the stack counter beyond 16 bits is ignored and compilation continues. A separate location counter is used for each procedure nested in an external procedure. This flag is usually caused by an array of excessive size.

26 (Error): Constant Over 32 Bits, Reduced Mod 2**32 --

A constant requires more than 32 bits of precision. The cursor points to the last valid contribution to the constant. For a numeric constant, overflow beyond 32 bits is lost; for a string constant, characters beyond the 4th are lost; in both cases compilation continues. For the purposes of this flag, a constant is considered a 32 bit (unsigned) magnitude number.

27 (Blunder): No 'THEN', Assumed --

A then-clause is required in an if-statement or if-expression at the point indicated by the cursor, but the keyword THEN is missing. Its presence is assumed and compilation continues.

28 (Blunder): No 'ELSE', Assumed --

An else-clause is required in an if-expression at the point indicated by the cursor, but the keyword ELSE is missing. Its presence is assumed and compilation continues.

29 (Abort): 50 Blunders and Errors, Job Aborted --

The blunder or error indicated by the cursor is the 50th. Compilation is aborted. If the abort is suppressed, compilation continues normally. The abort is issued only once on the 50th offense. Warnings and aborts are not included in the blunder and error count.

30 (Abort): Memory Bad, Job Aborted --

A memory checksum discrepancy has occurred at the point indicated by the cursor. Compilation is aborted. If the abort is suppressed compiler behavior is unpredictable. See Section 4.4 for a discussion of memory checking.

31 (Abort): Symbol Table Full, Job Aborted --

The symbol table is full as of the point indicated by the curosr. Compilation is aborted. If the abort is suppressed, compiler behavior is unpredictable. See Section 2.3 for a discussion of remedial action.

32 (Abort): 256 Blocks, Job Aborted --

The block indicated by the cursor is the 256th, exceeding the capacity of the compiler. Compilation is aborted. If the abort is suppressed, compiler behavior is unpredictable. The only remedy for this condition is to rewrite the program to use fewer blocks.

33 (Abort): 16 Lex Levels, Job Aborted --

The block indicated by the cursor is at the 16th lex level exceeding the capacity of the compiler. Compilation is aborted. If the abort is suppressed, compiler behavior is unpredictable. The only remedy for this condition is to rewrite the program to use few lex levels.

34 (Abort): Too Many Nested Literallys, Job Aborted --

The literally indicated by the cursor requires too many references to other nested literallys.
Compilation is aborted. If the abort is suppressed, compiler behavior is unpredictable. See Section 2.3 for a discussion of literally nest depth. Usually this flag is caused by a loop in a literally chain.

35 (Warning) --

Spare

36 (Abort): Stack Overflow, Job Aborted --

The stack used by the compiler at compilation time has overflowed at the point in the user's program indicated by the cursor. Compilation is aborted. If the abort is suppressed, compiler behavior is unpredictable. See Section 2.3 for a discussion of compile-time stack use.

37 (Abort): Source Physical EOF, Job Aborted --

An unexpected end-of-file has been encountered on the source file. Compilation is aborted. If the abort is suppressed, compilation continues normally; the input file of course, may be undefined after an EOF. An EOF is legitimate only after the end of an external procedure and before the entry name of an ensuing external procedure; comments and loader text may appear in this interprogram region.

38 (Abort): Hash Table Full, Job Aborted --

The hash table used by the symbol table is full as of the point indicated by the cursor. Compilation is aborted. If the abort is suppressed, compiler behavior is unpredictable. See Section 2.3 for a discussion of hash table use.

39 (Warning): Hash Table 90% Full, Ignored --

The hash table used by the symbol table is 90% as of the point indicated by the cursor. No action is taken and compilation continues. Compilation speed will degrade as the hash table approaches 100% full, at which time compilation is aborted; see Abort 38.

40 (Abort): Object IO Error, Job Aborted --

An irrecoverable I/O error has occurred with the object file. Compilation is aborted. If the abort is suppressed, compilation will continue normally; however, the contents of the object file may be flawed. If the abort occurs before any object has been written the problem likely lies with the assignment or opening of the object file. If the abort occurs after some object has been written, the problem is necessarily with the data transfer to the object file.

41 (Blunder): No Assignment Operator (=,+=,:=), Ignored --

An assignment operator is expected at the point indicated by the cursor. Code generation for the missing assignment operator is ignored and compilation continues.

42 <u>(Blunder): Operator Needs Word Destination, Ignored</u> --

The operand indicated by the cursor must be of size word but is otherwise declared. Code generation for the operation is ignored and compilation continues.

43 <u>(Blunder): Undeclared Variable, Declared as Word Auto</u> --

The variable indicated by the cursor is undeclared. It is given automatic class, internal scope, word size, and dimension as written, but no stack space is allocated. Compilation continues.

44 <u>(Blunder): Too Many Forwards Refs to Procs/Labels/Initlists, Skipped</u> --

The forward reference indicated by the cursor overflows the forward table. The reference is not saved and will never be resolved. Compilation continues. Forward references to procedures, labels, and initial lists share a common table. A forward reference to a procedure is resolved (making room for another) when the procedure entry is encountered; a label is resolved when the label is encountered; an initial list is resolved when the first executable statement of a procedure is encountered.

45 <u>(Error): Keyword Defined as Symbol, Defined But No Refs Possible</u> --

The keyword indicated by the cursor is being defined as a symbol. The declaration is completed but subsequent references to the keyword as a symbol will result in further flags. Compilation continues.

46 <u>(Warning): No 'PROC', Assumed</u> --

The keyword PROC (or PROCEDURE) is required at the point indicated by the cursor but is missing. Its presence is assumed and compilation continues.

47 <u>(Warning): No ';', Assumed</u> --

A semicolon (';') is required at the point indicated by the cursor, but is missing. Its presence is assumed and compilation continues.

48 <u>(Blunder): No Parameter, Skipped</u> --

A parameter is required at the point indicated by the cursor but is missing. The declaration of the missing parameter is skipped and compilation continues.

49 (Warning) --

Spare

50 (Error): No External Prochead, 'MAIN' Used --

An external procedure head is required at the point indicated by the cursor, but is missing. A head of 'MAIN PROCEDURE' is used and compilation continues.

51 (Warning): No Declaration List Element, Skipped --

A declaration list element is expected at the point indicated by the cursor but is missing. Declaration of the missing element is skipped and compilation continues.

52 (Warning): No Name List Element, Skipped --

A name list element (of a declaration list element) is expected at the point indicated by the cursor but is missing. Declaration of the missing element is skipped and compilation continues.

53 (Error): No Constant, Skipped --

A constant is expected at the point indicated by the cursor but is missing. Code generation for the missing constant is skipped and compilation continues.

54 (Error): No String, Skipped --

A string is expected at the point indicated by the cursor but is missing. Code generation for the missing string is skipped and compilation continues.

55 (Error): No Initial or Constant List Item, 0 Used --

An initial or constant list element is expected at the point indicated by the cursor but is missing. Zero is used for the missing element and compilation continues.

56 (Error): Parameter Not Fully Declared, Word Used --

One or more parameters (a symbol in the parenthesized name list following a procedure head) is never fully declared. Word size and dimension of scalar is used. The cursor is not meaningful. Compilation continues. The offending parameters are flagged in the symbol table listing in the state ('ST') field.

57 (Error): Interrupt Proc Declared With Lex>1, Used as Declared --

The interrupt procedure, whose procedure head is indicated by the cursor, appears at a lex depth greater than 1. The procedure is used as declared but the object program may not work. Compilation continues.

58 (Error): Unsatisfied Forward Labels or Procs, See Symbol Table --

One or more forward references to a label or procedure is never satisfied by an appropriate label or entry. The cursor is not meaningful. Compilation continues. The offending symbols are flagged in the symbol table listing in the state ('ST') field.

59 (Error): Junk in DCL Stmt. Text Skipped to Next ',' or ';' --

A declaration statement cannot be decoded at the point indicated by the error cursor. Source text is skipped until the next delimiting comma (',') or semicolon (';'). (Commas or semicolons appearing in a comment or string constant are skipped as well.) Compilation continues.

60 (Warning): External Symbol Over 8 Chars, Tail Ignored --

An external symbol exceeds 8 characters. The cursor indicates the offending symbol. Characters beyond the 8th are not passed to the loader. Compilation continues.

61 (Error): String Needs Byte Size, Treated as Byte --

A variable preset with a string, to which the cursor points, is not declared as size byte. The variable if treated as size byte for the purposes of initialization, but otherwise as the declared size. Compilation continues.

62 (Warning): String Length Does Not Match Dimension, Ignored --

A variable preset with a string, to which the cursor points, has an explicit dimension that does not match the string length. The entire string is used even if a succeeding variable overlaid; the declared dimension is unaltered. Compilation continues. This flag is generated for an INITIAL variable only if the string length exceeds the declared dimension, for a CONSTANT variable if the length does not exactly match the dimension.

63 (Warning): Initial List Too Long, Used Anyway --

The initial list, indicated by the cursor, for an INITIAL variable exceeds the declared dimension. The entire list is used even if it overlays succeeding variables; the declared dimension is unaltered. Compilation continues. No flag is generated if the initial list is too short.

64 (Warning): Constant List Too Long or Short, Used Anyway --

The constant list, indicated by the cursor, for a CONSTANT variable does not match the declared dimension. The entire list is used even if it overlays succeeding variables; the declared dimension is unaltered. Compilation continues.

65 (Error): Base Variable Must Be Backward Ref, Cbase=0 Used --

The basing variable, indicated by the cursor, of a based variable has not been previously defined. A constant base of zero is used for the variable being declared and compilation continues. If the basing variable is subsequently declared it will be flagged as already defined (Error 7).

66 (Blunder): Base Not Scalar Storage Ref, Used Anyway --

The basing variable, indicated by the cursor, of a based variable is not a scalar storage reference. The base is used anyway but the generated code is erroneous. Compilation continues. The base must be of class automatic, static, parameter, constant based, or variable based.

67 (Error): Bad Size, Word Used --

The size attribute, indicated by the cursor, is missing or invalid. A size of word is used and compilation continues.

68 (Error): Bad Bit Size, Bit(4) Used --

The bit size attribute, indicated by the cursor, is not 1, 2, or 4. Bit(4) is used and compilation continues.

69 (Warning): Initial or Constant Value Too Big, Truncated on Left --

The element in an initial or constant list, indicated by the cursor, is too big for the declared size. Bits are truncated from the left. The flag is generated if the bits discarded from a 32 bit intermediate value are not all the same. This interpretation of precision differs from that used in expression translation.

70 (Blunder): Parameter May Not Have Area Attribute, See Symbol Table --

An area attribute, indicated by the cursor, may not be specified for a parameter. The attributes used for the parameter are shown in the symbol table, but inasmuch as they are inconsistent, the code generated for parameter references will be erroneous. Compilation continues.

71 (Warning) --

Spare

72 (Error): Base Missing, Cbase=0 Used --

The based variable, indicated by the cursor, contains an invalid or incomplete base reference at the point indicated by the cursor. A constant base of zero is used and compilation continues.

73 (Error): Proc Size Mismatch, New Size Used --

The new size for a procedure specified in an entry statement does not match the old size specified in a declaration statement. The new size is used for all subsequent code generation and compilation continues. Previous code generation will have used the old size. The cursor points to the procedure head of the offending entry statement.

74 (Blunder): No '=' in Iterative Do, Assumed --

An interative do does not contain an equals operator at the point indicated by the cursor. Its presence is assumed and compilation continues. Either an equal sign ('=') or a colon-equal sign (':=') may be used.

75 (Error): Constant Over 16 Bits, Reduced Mod 2**16 --

The constant indicated by the cursor contains more than 16 bits of precision while the current context allows for only 16 (word precision). The low-order 16 bits of the constant are used and compilation continues.

76 (Warning): Do's Nested Too Deep, Nest Checking Invalid --

The do indicated by the cursor in nested too deep for nest checking. Compilation continues but future labeled ends will not be properly matched. See Section 2.3 for a discussion of do nest depth.

77 (Blunder): Too Many Do Cases, Ignored --

The do case indicated by the cursor exceeds the compiler capacity for active do cases. The body of the case is translated correctly but the link to the case is lost. Compilation continues. A case is active until the matching end for the enclosing do case statement is encountered.

78 (Blunder): No 'TO' in Iterative Do, Assumed --

The keyword 'TO' is required in an iterative do at the point indicated by the cursor. Its presence is assumed and compilation continues.

79 (Warning) --

Spare

80 (Blunder): Bad 'BASED' Nesting, Ignored --

The based variable, indicated by the cursor, has a loop in the basing chain. The chain is prematurely terminated and compilation continues. The flag cannot occur in the current compiler because the basing variable must be a backward reference; see Error 65.

81 (Blunder): Variable Must Be Storage Ref, Ignored --

The variable indicated by the cursor is used in a context that requires a storage reference. Code generation for the storage reference is incorrectly generated and compilation continues.

82 (Error): End Identifier Must Be Do-Label/Begin-Label/ Entry, Ignored --

The identifier, indicated by the cursor, of a labeled end is not the label on a do or begin statement nor is it an entry to a procedure. Block closure checking for this end is ignored and compilation continues.

83 (Error): Label Does Not Match Current Block, Ignored --

The identifier, indicated by the cursor, of a labeled end does not match the label on the innermost open block or group. The mismatch is ignored, the block or group is closed, and compilation continues.

84 (Error): Label On End Not Defined, Ignored --

The identifier, indicated by the cursor, of a labeled end is undefined. Block closure checking for this end is ignored and compilation continues.

85 (Warning): Over 32,765 Do Blocks, Nest Checking Invalid --

The do indicated by the cursor is the 32,766th exceeding the compiler capacity for groups. Compilation continues, but future labeled ends will not be properly matched.

86 (Warning): Stack Underflow (May Be Due To Another Flag), Ignored --

The location counter for the program's run-time stack (stack counter) has underflowed. The underflow is ignored and compilation continues. The underflow is usually a side effect of another flag.

87 (Blunder): No Proc Ref After 'CALL', Ignored --

A procedure reference is expected at the point indicated by the cursor after the keyword CALL but is missing or invalid. Code generation for the procedure reference is ignored and compilation continues. The flag is usually the result of an undeclared procedure.

88 (Blunder): No Label Ref After 'GO TO', Ignored --

A label is expected at the point indicated by the cursor after the keywords GO TO, GOTO, or GO but is missing or invalid. Code generation for the label is ignored and compilation continues.

89 (Blunder): Current Proc Cannot Return a Value, Ignored --

A return statement in a procedure is returning a value, indicated by the cursor, but the matching procedure head did not include a return size. The procedure head size omission is ignored, code generated for the return value, and compilation continues.

90 (Blunder): Current Proc Must Return a Value, Ignored --

A return statement in a procedure requires a value at the point indicated by the cursor but none is present or is invalid. The value omission is ignored and compilation continues. This flag is usually caused by the inclusion of a size in the matching procedure head where none was intended.

91 (Blunder): No Statement After 'THEN', Ignored --

A statement is expected at the point indicated by the cursor following the keyword THEN but none is present. Code generation for the statement is ignored and compilation continues.

92 (Blunder): No Statement After 'ELSE', Ignored --

A statement is expected at the point indicated by the cursor following the keyword ELSE but none is present. Code generation for the statement is ignored and compilation continues.

93 (Blunder): Embedded Assignment Does Not Allow Field Select, Ignored --

The field select operator, indicated by the cursor, may not be used in conjunction with an embedded assignment operator (':='). Code generation for the field selection is ignored and compilation continues.

94 (Blunder): No Proc Arg, Ignored --

A procedure argument is expected at the point indicated by the cursor but is missing or invalid. Code generation for the argument is ignored and compilation continues.

95 (Blunder): Bad Number of Args to Built-in Proc, Accepted --

The number of arguments to a built-in procedure is incorrect The cursor points to the argument list. Code is generated for the arguments written but the procedure will not work correctly. Compilation continues.

96 (Blunder): Bad Arg Size for Builtin Procedure, Accepted --

The argument indicated by the cursor has an incorrect size for the specified builtin procedure. Code is generated for the size as written but the procedure will not work correctly. Compilation continues.

97 (Blunder): Over 252 Words of Proc Args, Reduced Mod 252 --

The total length of the argument list, indicated by the cursor, exceeds 252 words. Code for the argument list is generated but the call cannot be correctly executed.

98 (Blunder): '@' Illegal With Based Variable or Constant, Ignored --

The based variable or constant indicated by the cursor is the object of the address operator ('@') which is illegal. Code generation for the address operation is ignored and compilation continues.

99 (Blunder): Constant Cannot Be a Destination, Used Anyway --

The constant indicated by the cursor is used as a storage destination which is illegal. Code is generated anyway but the code will not work.

APPENDIX E: TOGG.ES -- SIMPLE LIST

Toggle	Default Settin;	Function
A	On	List symbol table
B	On	Ignore high source bit
C	Off	List generated code
D	Off	Continue object on blunder
E	Off	Space for top-of-form
F	On	List flags
H	Off	List object program
I	On	Indent code listing
L	On	List source program
M	On	Honor memory checks
N	Off	Format for narrow page
O	Off	Generate object program
P	On	Reset toggles at program start
Q	Off	Chop source program listing
R	On	Collect symbol references
S	On	Format for short page
U	Off	Upspace listing
V	Off	Check memory each record
W	Off	Suppress warnings
X	Off	Continue on abort
Y	On	List program summary
Z	On	Honor listing requests (A C F H L Y)
#	Off	Honor object in source
?	Off	Honor early source truncation

APPENDIX F: TOGGLES -- DETAILED DESCRIPTION

A (On)*, List Symbol Table --

The A-toggle, when on, enables the symbol table listing. It is checked after the compilation of a program to determine if a symbol table listing is desired. It does not affect the content of the symbol table in any way.

B (On), Ignore High Source Bit --

The B-toggle controls the use of the high-order (leftmost) bit of the 8 bits that comprise a source program character. When the B-toggle is on this bit is ignored (set to zero), when off this bit is left unchanged. The B-toggle is normally left on when using 7 bit media, e.g. paper tape, and left off when using 8 bit media.

C (Off), List Generated Code --

The C-toggle, when on, enables the generated code listing. It is checked as each 32/S instruction is constructed. It does not affect the actual code generation in any way.

D (Off), Continue Object on Blunder --

The D-toggle, when on, defeats the normal suppression of object after a blunder. (Object is suppressed by turning the H-toggle and O-toggle off.) Inasmuch as a blunder indicates a faulty object program, extreme care should be taken in using any object generated under the D-toggle.

E (Off), Space for Top-of-Form --

The E-toggle, when on, causes a sequence of carriage return/line feeds to be substituted for form feeds when a top-of-form is required. This toggle is useful when the physical listing device does not recognize form feed or interprets form feed for an inappropriate page length.

F (On), List Flags --

The F-toggle, when on, enables the listing of flags. If the F-toggle is on and the L-toggle off, the offending source line will be listed in addition to the flag; this is useful in scanning for errors. Any flags suppressed because the F-toggle is off are still reflected in the program summary.

* Default setting

H (Off), List Object Program --

The H-toggle, when on, enables the listing of the object program. The H-toggle is independent of the O-toggle (generate object). The H-toggle is turned off by the compiler when a blunder occurs, unless the D-toggle is on.

I (On), Indent Code Listing --

The I-toggle, when on, causes any generated code that is listed (which occurs when the C-toggle is on) to be indented 50 spaces for improved listing clarity. On slower printers this indentation may slow output speed excessively.

L (On), List Source Program --

The L-toggle, when on, enables the listing of the annotated source program. If the L-toggle is off, and the F-toggle is on and a flag occurs, the offending line is listed anyway.

M (On), Honor Memory Checks --

The M-toggle, when on, enables the checksumming of the program space occupied by the compiler. The frequency of this checking is controlled by the V-toggle.

N (Off), Format For Narrow Page --

The N-toggle, when on, folds all listing output after character 70. This toggle is usually turned on only when a Teletype is used as the output device, as the listing becomes difficult to read.

O (Off), Generate Object Program --

The O-toggle, when on, enables the generation of an object program on the object file. The O-toggle (generate object) is independent of the H-toggle (list object). The O-toggle is turned off by the compiler when a blunder occurs unless the D-toggle is on.

P (On), Reset Toggles at Program Start --

The P-toggle, when on, causes all toggles to be reset to their default state before each program is compiled. This reset may be defeated by turning the P-toggle off. Note that to apply the same set of toggles to a group of programs, the common set of toggles need appear just once in the first program in conjunction with the P-toggle.

Q (Off), Chop Source Program Listing --

The Q-toggle, when on, causes the annotated source listing to be truncated after the source line: that is, the right-hand annotation is not listed. No other listings or titles are affected. This toggle will increase listing speed on slow listing devices.

R (On), Collect Symbol References --

The R-toggles, when on, enables the collection of symbol references. This activity has a negligible affect on compilation speed but a significant effect on symbol table space. If symbol table space is insufficient, this toggle is normally turned off. When the symbol table is listed any symbol with no references generates a 'SUPPRESSED' message; symbols with some references listed may also have some suppressed.

S (On), Format for Short Page --

The S-toggle, when on, defines the listing page size as 51 lines, when off as 66 (including margins).

U (Off), Upspace Listing --

The U-toggle, when on, causes a top-of-form to be inserted in the listing file before the current line is listed. This toggle is set off by the compiler after it is processed. The U-toggle may be used to control the pagewise formatting of a listing. Should a U-toggle be encountered while the L-toggle (list source program) is off, the upspace request is ignored, but its suppression is indicated in the toggle summary by the appearance of an asterisk ('*').

V (Off), Check Memory Each Record --

The V-toggle, when on, causes memory checking to occur each source record. Memory is always checked once per program before the summary is listed. Memory checking consists of 2 parts: memory checksum check (performed only if the M-toggle is on) and compiler stack size check. If either check discloses a problem, the compiler aborts.

W (Off), Suppress Warnings --

The W-toggle, when on, suppresses warnings. Suppressed warnings are included in the flag summary but excluded from the console log and flag link. This toggle is useful when warnings are anticipated, but excessive use of the W-toggle is poor programming practice.

X (Off), Continue on Abort --

The X-toggle, when on, causes compilation to continue after an abort. If an abort is suppressed, an 'ABORT SUPPRESSED' message is issued on the console log and program listing. The exact response of the compiler after an abort is indicated in Appendix D in conjunction with the flag descriptions. The routine suppression of aborts is a dangerous practice.

Y (On), List Program Summary --

The Y-toggle, when on, enables the listing of the program summary.

Z (On), Honor Listing Requests --

The Z-toggle causes other listing requests to be honored. The Z-toggle is a master enable for the A, C, F, H, L, and Y-toggles. The Z-toggle is useful in turning off all listings without knowledge of which listing components are enabled.

(Off), Honor Object in Source --

The #-toggle, when on, causes source records with a pound sign ('#') in the first byte to be transmitted directly to the object file. Such records are otherwise treated as comments. The object record buffer is flushed before the source record is transmitted. No checksum or sequence is computed for the transmitted record. The #-toggle provides a mechanism for the direct construction of loader text. For example, to insert a loader end record into the object file after the last program, suffix that last program with:

```
/* &O &# */
#END
```

? (Off), Honor Early Source Truncation --

The ?-toggle, when on, causes any text in a source record beyond the first character and beyond the first question mark ('?') to be discarded. This premature record truncation occurs even if the question mark appears in a comment or string. To turn the ?-toggle off, the question mark must appear as the first character in a record lest it be treated as a comment. For example, the following program uses the question mark both to delimit comments and as string text.

```
p:   PROC; DCL                 /* &? */
     a WORD,                       ? Comment
     b BYTE,                       ? Comment
                                             /* %
? */
     c BYTE CONSTANT '???', /* &? */
     d WORD;                       ? Comment
END p;
```

The ?-toggle provides an alternate mechanism for program annotation, but is considered poor coding practice.

APPENDIX G: LIST OF FOULS

Foul	Procedure	Cause
CLEV	extproc	Case level not 0 at program end
CPTR	extproc	Case table not empty at program end
CPTX	group	Case tabel pointer not in case table
DELT	delta	Symbol table entry has bad units field
DLEV	extproc	Do level not 0 at program end
ENDS	end-stmt	End statement type out of range
FLG1	flag	Flag number out of range
FLG3	flag	Flag severity invalid
GETI	getst	Symbol table entry pointer not in symbol table
GETS	getst	Symbol table field code out of range
IDEX	index	Item not found
INSE	listst	Field description not found
INST	insertst	Text length out of range
LAB1	labels	Symbol type not an identifier
LAB2	labels	Symbol type is a keyword
LABS	labels	Symbol type out of range
LIT	physchar	Literally level less than 0
LLEV	extproc	Lex level not 0 at program end
MOST	movest	Target field length out of range
MOVE	move	Target field length out of range
PBSC	procbody	Stack counter invalid at program end
PRIN	print	Line size exceeds buffer size
PULD	pull-do	Do level less than 1
PULL	pull-lex	Lex level less than 1
PURG	purgefwd	Purge type out of range
PUTI	putst	Symbol table entry pointer not in symbol table
PUTS	putst	Symbol table field code out of range
RPTC	rptchar	Repeat character flag already active
RUSE	reuse	Reuse level out of range
SET	set	Target field length out of range
SIZE	mpl	Symbol table base too big
SPUT	prinwrit	Line size exceeds buffer size
TUSE	token	Reuse level out of range
TYPE	type	List size exceeds buffer size
WEXP	group	Expression type invalid

APPENDIX H: FLAG REFERENCES

Flag	Is Generated By Procedure --
0	--
1	extproc
2	string
3	constunt dclelemt dclelemt initlist num-pri num-pri prochead prochead proc-ref ref sizeattr s-ref var
4	constunt
5	constunt
6	extproc
7	blocksen dclelemt dclelemt dclelemt dclelemt dclelemt labels prochead
8	token
9	logchar
10	get-lex
11	var
12	ref
13	ref var var
14	s-ref
15	num-pri s-ref
16	s-ref s-ref
17	num-pri
18	c-exp c-exp group if-then num-pri
19	konstant
20	num-pri
21	num-pri
22	semiscan
23	as-stmt exp log-fact log-term num-exp num-fact num-term s-exp
24	inc-pc
25	inc-sc inc-xc
26	constunt constunt
27	if-then
28	c-exp
29	flag
30	memcheck
31	insertst putst
32	push-lex
33	push-lex
34	token
35	--
36	memcheck
37	sread
38	search
39	newentry

```
40     owrite owrite owrite
41     as-stmt exp
42     as-stmt exp group
43     in-sym
44     save-ref

45     symbol
46     dclelemt
47     semi
48     prochead prochead
49     --

50     extproc
51     dcl-stmt dcl-stmt
52     dclelemt dclelemt
53     dclelemt
54     dclelemt

55     areaattr initlist
56     procbody
57     blocksen
58     extproc
59     dcl-stmt

60     areaattr dclelemt extproc
61     initstr
62     initstr
63     initlist
64     initlist

65     areaattr
66     areaattr areaattr
67     sizeattr sizeattr
68     sizeattr
69     initlist

70     dclelemt
71     --
72     areaattr
73     blocksen
74     group

75     areaattr dclelemt
76     push-do
77     group
78     group
79     --
```

80	get-mode
81	group
82	end-stmt
83	end-stmt
84	end-stmt
85	push-do
86	dec-sc
87	cal-stmt
88	gotostmt gotostmt
89	num-pri ret-stmt
90	ret-stmt
91	if-stmt
92	if-stmt
93	exp
94	proc-ref proc-ref
95	proc-ref
96	proc-ref
97	proc-ref
98	num-pri proc-ref
99	as-stmt exp group

APPENDIX I: LOAD MAP

```
**PROCEDURE ENTRY POINTS**
             ENTRY-ADDR    STATIC-ORG
  NAME        DEC/HEX      DEC/HEX
HPL             Ø/ØØØØ        8/ØØØ8
ADD←REF      1Ø46/Ø416        8/ØØØ8
APLAATTR     1Ø6Ø/Ø424        8/ØØØ8
AS←STMT      1628/Ø65C        8/ØØØ8
A←LVT        1968/Ø7BØ        8/ØØØ8
B2D1W        2Ø14/Ø7DE        8/ØØØ8
B2D1WL       2186/Ø88A        8/ØØØ8
B2DNR        228Ø/Ø8E8        8/ØØØ8
BIN2HEX      2384/Ø95Ø        8/ØØØ8
BIN2HEX2     2414/Ø96E        8/ØØØ8
BIN2HEX1     2444/Ø98C        8/ØØØ8
BLUC         248Ø/Ø9BØ        8/ØØØ8
BLUCKSEN     258Ø/ØA14        8/ØØØ8
BYTESIZE     29Ø2/ØB56        8/ØØØ8
B←LVT        2936/ØB78        8/ØØØ8
CAL←STMT     2996/ØBB4        8/ØØØ8
CONSTUNT     3Ø94/ØC16        8/ØØØ8
C←EXP        351Ø/ØDB6        8/ØØØ8
DCLELEMT     376Ø/ØEBØ        8/ØØØ8
DCL←STMT     4912/133Ø        8/ØØØ8
DECHAIN      5Ø48/13B8        8/ØØØ8
DEC←SC       5Ø8Ø/13D8        8/ØØØ8
DEFST        5124/14Ø4        8/ØØØ8
DELTA        5176/1438        8/ØØØ8
ENCHAIN      5276/149C        8/ØØØ8
END←STMT     5344/14EØ        8/ØØØ8
EXIT         5538/15A2        8/ØØØ8
EXP          56Ø2/15E2        8/ØØØ8
EXTPROC      5826/16C2        8/ØØØ8
EXUNIT       6278/1886        8/ØØØ8
FLAG         63Ø6/18A2        8/ØØØ8
FORMAT      1Ø712/29D8        8/ØØØ8
FOUL        1Ø914/2AA2        8/ØØØ8
GASZ        1Ø966/2AD6        8/ØØØ8
GETST       11ØØ4/2AFC        8/ØØØ8
GET←LEX     11264/2CØØ        8/ØØØ8
GET←MODE    11298/2C22        8/ØØØ8
GISZ        11748/2DE4        8/ØØØ8
GOTOSTMT    11778/2EØ2        8/ØØØ8
GPSZ        12116/2F54        8/ØØØ8
GROUP       1215Ø/2F76        8/ØØØ8
IF←CLAUS    13316/34Ø4        8/ØØØ8
IF←STMT     13334/3416        8/ØØØ8
IF←THEN     1347Ø/349E        8/ØØØ8
INC←PC      1354Ø/34E4        8/ØØØ8
INC←SC      13568/35ØØ        8/ØØØ8
INC←XC      13644/354C        8/ØØØ8
INDEX       13688/3578        8/ØØØ8
INITLIST    13848/3618        8/ØØØ8
INITSTR     14812/39DC        8/ØØØ8
INSERTST    15168/3B4Ø        8/ØØØ8
IN←SYM      15264/3BAØ        8/ØØØ8
KONSTANT    15376/3C1Ø        8/ØØØ8
LABDOBEC    1543Ø/3C46        8/ØØØ8
LABELS      15458/3C62        8/ØØØ8
LIST        1575Ø/3D86        8/ØØØ8
LISTST      16Ø6Ø/3EBC        8/ØØØ8
LOGCHAR     17546/448A        8/ØØØ8
LOG←FACT    17756/455C        8/ØØØ8
LOG←TERM    17986/4642        8/ØØØ8

LOOKFOR     18Ø8Ø/46AØ        8/ØØØ8
LST←ADR     181Ø6/46BA        8/ØØØ8
MEMCHECK    18182/47Ø6        8/ØØØ8
MOVE        18518/4856        8/ØØØ8
MOVEST      18574/488E        8/ØØØ8
NAMESAME    18664/48E8        8/ØØØ8
NEWENTRY    18738/4932        8/ØØØ8
NUL←STMT    18874/49BA        8/ØØØ8
NUM←EXP     18888/49C8        8/ØØØ8
NUM←FACT    19Ø3Ø/4A56        8/ØØØ8
NUM←PRI     1915Ø/4ACE        8/ØØØ8
NUM←TERM    19792/4D5Ø        8/ØØØ8
OLT         2ØØ66/4E62        8/ØØØ8
OUT←ADDR    2Ø692/5ØD4        8/ØØØ8
OUT←BYTE    2Ø826/515A        8/ØØØ8
OUT←DBLE    2Ø918/51B6        8/ØØØ8
OUT←INST    21Ø6Ø/5244        8/ØØØ8
OUT←LIT     2376Ø/5CDØ        8/ØØØ8
OUT←LWL     24Ø14/5DCE        8/ØØØ8
OUT←REC     24132/5E44        8/ØØØ8
OUT←WORD    24354/5F22        8/ØØØ8
OUT←WRT     24458/5F8A        8/ØØØ8
OWRITE      24492/5FAC        8/ØØØ8
PHYSCHAR    24644/6Ø44        8/ØØØ8
PRINT       2493Ø/6162        8/ØØØ8
PRINWRIT    25ØØ2/61AA        8/ØØØ8
PROCBODY    26Ø7Ø/65D6        8/ØØØ8
PROCHEAD    26654/681E        8/ØØØ8
PROC←ID     26994/6972        8/ØØØ8
PROC←REF    27Ø76/69C4        8/ØØØ8
PULL←DU     27796/6C94        8/ØØØ8
PULL←LEX    27828/6CB4        8/ØØØ8
PURGEFWD    28166/6EØ6        8/ØØØ8
PUSH←DU     28324/6EA4        8/ØØØ8
PUSH←LEX    284Ø4/6EF4        8/ØØØ8
PUTST       285Ø4/6F58        8/ØØØ8
REF         28898/7ØE2        8/ØØØ8
REFST       29Ø42/7172        8/ØØØ8
RET←STMT    29444/73Ø4        8/ØØØ8
REUSE       29666/73E2        8/ØØØ8
RPTCHAR     2974Ø/742C        8/ØØØ8
SAVE←REF    29772/744C        8/ØØØ8
SEARCH      29838/748E        8/ØØØ8
SEMI        3ØØ4Ø/7558        8/ØØØ8
SEMISCAN    3ØØ68/7574        8/ØØØ8
SET         3Ø1Ø8/759C        8/ØØØ8
SETTOGS     3Ø162/75D2        8/ØØØ8
SIMPLESZ    3Ø33Ø/767A        8/ØØØ8
SIZEATTR    3Ø4Ø6/76C6        8/ØØØ8
SREAD       3Ø642/77B2        8/ØØØ8
STATEST     3Ø822/7866        8/ØØØ8
STMT        3Ø88Ø/78AØ        8/ØØØ8
STRING      3Ø942/78DE        8/ØØØ8
SUMMARY     31214/79EE        8/ØØØ8
SYMBOL      32286/7E1E        8/ØØØ8
S←EXP       324Ø6/7E96        8/ØØØ8
S←REF       32548/7F24        8/ØØØ8
TOKEN       32764/7FFC        8/ØØØ8
TYPE        3453Ø/86E2        8/ØØØ8
UNEXUNIT    3465Ø/875A        8/ØØØ8
VAR         34696/8788        8/ØØØ8
VAR←ID      34964/8894        8/ØØØ8
```

```
**EXTERNAL  DATA  ADDRESSES**
            ADDRESS
  NAME      DEC/HEX
SOSIZE        8/ØØØ8
OBJSIZE      1Ø/ØØØA
LISTSIZE     12/ØØØC
STACKSIZ     14/ØØØE
HASHSIZE     16/ØØ1Ø
MAXLEX       18/ØØ12
MAXLIT       2Ø/ØØ14
FWDMAX       22/ØØ16
DOMAX        24/ØØ18
CASEMAX      26/ØØ1A
WIDTH4       28/ØØ1C
ERRMAX       3Ø/ØØ1E
ABORTS       32/ØØ2Ø
BLUNDERS     34/ØØ22
ERRORS       36/ØØ24
WARNINGS     38/ØØ26
WARNINGT     4Ø/ØØ28
STMTS        42/ØØ2A
SYMBOLS      44/ØØ2C
REFS         46/ØØ2E
PAGELINE     48/ØØ3Ø
PAGE         5Ø/ØØ32
LASTFLAG     52/ØØ34
TITLEPTR     54/ØØ36
LISTWAIT     56/ØØ38
STBASE       58/ØØ3A
STBASEØ      6Ø/ØØ3C
ITEMBASE     62/ØØ3E
ITEMMAX      64/ØØ4Ø
TEXTMIN      66/ØØ42
TEXTMIN4     68/ØØ44
VARCHAIN     7Ø/ØØ46
PARCHAIN     72/ØØ48
NAMCHAIN     74/ØØ4A
LABCHAIN     76/ØØ4C
VARDESC      78/ØØ4E
PARDESC      82/ØØ52
NAMDESC      86/ØØ56
LABDESC      9Ø/ØØ5A
HASHLOC      94/ØØ5E
HASHTAB      96/ØØ6Ø
ACCESSES    9ØØ/Ø384
COLISONS    9Ø2/Ø386
BLOCK       9Ø4/Ø388
LEX         9Ø6/Ø38A
LINE        9Ø8/Ø38C
CURSOR      91Ø/Ø38E
LEXDEPTH    912/Ø39Ø
NUMBERP     914/Ø392
NUMBERD     916/Ø394
XC          918/Ø396
PC          92Ø/Ø398
PCLONS      922/Ø39A

SC1MAX      924/Ø39C
SC          926/Ø39E
BLOCKLEX    958/Ø3BE
PROCLEX     99Ø/Ø3DE
OBJX       1Ø22/Ø3FE
LOBJX      1Ø24/Ø4ØØ
OBJITEM    1Ø26/Ø4Ø2
OBJRECN    1Ø28/Ø4Ø4
SPECMODE   1Ø3Ø/Ø4Ø6
PTR        1Ø32/Ø4Ø8
PTROLD     1Ø34/Ø4ØA
DESC       1Ø36/Ø4ØC
DESCOLD    1Ø4Ø/Ø41Ø
ICODE      1Ø44/Ø414
TCODEOLD   1Ø46/Ø416
REUSELEV   1Ø48/Ø418
LOFFLAG    1Ø5Ø/Ø41A
LABFLAG    1Ø52/Ø41C
DOLEV      1Ø54/Ø41E
DONUMBER   1Ø56/Ø42Ø
DONEST     1Ø58/Ø422
CASEPTR    11ØØ/Ø44C
CASELEV    11Ø2/Ø44E
CASEITEM   11Ø4/Ø45Ø
CASEBUF    11Ø6/Ø452
TOKENLEV   17Ø8/Ø6AC
TITLEV     171Ø/Ø6AE
RPTFLAG    1712/Ø6BØ
CHARSAVE   1734/Ø6C6
LITBASE    1756/Ø6DC
LITCURSR   1778/Ø6F2
OBJECTON   18ØØ/Ø7Ø8
CONSFLAG   18Ø2/Ø7ØA
CONSLOC    18Ø4/Ø7ØC
FWDPTR     18Ø6/Ø7ØE
CHECKS     18Ø8/Ø71Ø
STACKUSE   1872/Ø75Ø
PB         1874/Ø752
PL         1876/Ø754
FWDBASE    1878/Ø756
FWDPC      22ØØ/Ø898
OBJFCP     2522/Ø9DA
SOBUF      255Ø/Ø9F6
OBJBUF     263Ø/ØA46
LISTBUF    2712/ØA98
FLAGBUF    2832/ØB1Ø
CODEBUF    2952/ØB88
OBJBUF1    3Ø72/ØCØØ
DUM1       3154/ØC52
OBJBUF2    3158/ØC56
DUM2       324Ø/ØCA8
TOGGLE     3244/ØCAC
FWDLEX     35ØØ/ØDAC
VERSION    3662/ØE4E
VALUE      3666/ØE52
```

APPENDIX J: EXTERNAL PROCEDURE REFERENCES

Procedure	Is Called By Procedure --
mpl	--
add-ref	areaattr blocksen end-stmt gotostmt group num-pri proc-ref token var
areaattr	dclelemt
*assign	owrite
as-stmt	stmt
a-cvt	as-stmt exp if-then log-fact num-term ref ret-stmt s-ref var
b2d10	b2d101 b2dnr
b2d101	flag out-addr out-byte out-dble out-inst out-lit out-lwl out-word
b2dnr	format list listst lst-adr prinwrit
bin2hex	bin2hex2 bin2hex4
bin2hex2	out-byte out-inst out-lit outlwl out-rec
bin2hex4	format list listst lst-adr out-addr out-dble out-inst out-lit out-lwl out-word
bloc	unexunit
blocksen	group procbody
bytesize	areaattr initstr procbody
b-cvt	log-fact log-term num-exp num-term s-exp
cal-stmt	stmt
*close	exit
constunt	token
c-exp	exp
*date	mpl
dclelemt	dcl-stmt
dcl-stmt	procbody
dechain	bloc dclelemt group procbody
dec-sc	c-exp group out-inst proc-ref ret-stmt
defst	gotostmt symbol
delta	dclelemt procbody
*display	extproc type
enchain	dclelemt labels prochead

end-stmt	group procbody
exit	flag sread
exp	as-stmt c-exp group if-then num-pri proc-ref ref ret-stmt s-ref var
extproc	mpl
exunit	blocksen if-stmt
flag	areaattr as-stmt blocksen c-exp cal-stmt constunt dclelemt dcl-stmt dec-sc end-stmt exp extproc get-lex get-mode gotostmt group if-stmt if-then inc-pc inc-sc inc-xc initlist initstr insertst in-sym konstant labels logchar log-fact log-term memcheck newentry num-exp num-fact num-pri num-term owrite procbody prochead proc-ref push-do push-lex putst ref ret-stmt save-ref search semi semiscan sizeattr sread string symbol s-exp s-ref token var
format	print type
foul	delta end-stmt extproc flag getst group index insertst labels listst mpl move movest physchar print prinwrit procbody pull-do pull-lex purgefwd putst reuse rptchar set token type
gasz	as-stmt num-pri proc-ref
getst	areaattr as-stmt blocksen bytesize dclelemt dechain delta end-stmt exp extproc gasz get-lex get-mode gisz gotostmt gpsz group initlist initstr labdobeg labels listst movest namesame num-pri olt out-inst physchar procbody proc-id proc-ref pull-lex ref refst ret-stmt search s-ref token var var-id
get-lex	gotostmt out-inst
get-mode	num-pri var
gisz	as-stmt exp gasz get-mode num-pri ret-stmt
gotostmt	stmt
gpsz	get-mode out-inst ret-stmt
group	unexunit
if-claus	if-stmt
if-stmt	exunit
if-then	c-exp if-claus
inc-pc	out-addr out-byte out-dble out-inst out-lit out-lwl out-word
inc-sc	out-inst out-lit out-lwl procbody proc-ref
inc-xc	extproc procbody
index	refst token
initlist	areaattr
initstr	areaattr
insertst	dclelemt newentry
in-sym	group num-pri proc-ref var

konstant	areaattr dclelemt initlist num-pri sizeattr
labdobeg	gotostmt pull-lex
labels	if-claus unexunit
list	flag foul memcheck mpl out-wrt owrite sread
listst	flag mpl
logchar	constunt token
log-fact	log-term
log-term	s-exp
lookfor	areaattr bloc blocksen cal-stmt c-exp dclelemt dcl-stmt end-stmt exp extproc gotostmt group if-stmt if-then initlist in-sym labels log-term mpl nul-stmt num-pri prochead proc-ref ref ret-stmt semi simplesz sizeattr s-ref var
lst-adr	out-addr out-byte out-dble out-inst out-lit out-lwl out-word
memcheck	mpl sread
move	extproc flag listst mpl prinwrit sread
movest	extproc list listst olt out-inst
namesame	refst
newentry	defst refst
nul-stmt	stmt
num-exp	log-fact
num-fact	num-term
num-pri	num-fact
num-term	num-exp
olt	dclelemt extproc out-addr out-byte out-dble out-inst out-lit out-lwl out-word procbody prochead
*open	owrite
out-addr	group purgefwd
out-byte	areaattr initlist initstr
out-dble	initlist initstr
out-inst	areaattr as-stmt a-cvt bloc blocksen b-cvt cal-stmt c-exp exp extproc get-mode gotostmt group if-stmt initlist initstr labels log-fact log-term num-exp num-fact num-pri num-term out-lit procbody proc-ref purgefwd ref ret-stmt s-exp s-ref var
out-lit	get-mode num-pri var
out-lwl	num-pri
out-rec	exit mpl olt sread
out-word	initlist initstr
out-wrt	out-addr out-byte out-dble out-inst out-lit out-lwl out-word

owrite	extproc out-rec sread
physchar	constunt logchar string token
print	flag foul memcheck summary
prinwrit	flag list listst out-wrt owrite print summary
procbody	bloc blocksen extproc
prochead	blocksen extproc
proc-id	cal-stmt numpri proc-ref
proc-ref	cal-stmt num-pri
pull-do	end-stmt pull-lex
pull-lex	end-stmt
purgefwd	blocksen labels
push-do	group push-lex
push-lex	bloc extproc prochead
*put	owrite
putst	add-ref areaattr bloc blocksen dclelemt defst enchain ext-proc get-mode gotostmt group initstr in-sym labels newentry procbody prochead pull-lex ref-st simplesz sizeattr symbol
ref	s-ref
refst	areaattr end-stmt in-sym
ret-stmt	stmt
reuse	as-stmt blocksen dcl-stmt gotostmt in-sym konstant labels log-fact lookfor num-exp num-fact num-pri num-term proc-ref symbol s-exp s-ref var
rptchar	constunt logchar string token
save-ref	areaattr gotostmt num-pri out-inst
search	defst pull-lex statest
semi	extproc prochead
semiscan	as-stmt bloc cal-stmt end-stmt gotostmt group procbody ret-stmt
set	extproc flag list listst mpl print prinwrit sread summary
settogs	mpl
simplesz	dclelemt prochead sizeattr
sizeattr	dclelemt
sread	physchar
statest	gotostmt refst token
stmt	unexunit
string	token
summary	flag mpl

symbol	blocksen dclelemt extproc labels prochead
*sysget	sread
*sysput	prinwrit
s-exp	exp
s-ref	as-stmt exp num-pri
*time	mpl
token	as-stmt blocksen dcl-stmt ext-proc gotostmt in-sym konstant labels log-fact lookfor num-exp num-fact num-pri num-term semiscan symbol s-exp s-ref
type	exit flag foul memcheck mpl
unexunit	exunit
var	num-pri ref
var-id	group var

* System procedure

APPENDIX K: EXTERNAL VARIABLE REFERENCES

Variable	Is Referenced By Procedure --
aborts	mpl flag summary
accesses	mpl search summary
block	mpl push-lex summary
blocklex(maxlex)	mpl defst gotostmt list newentry pull-lex push-lex statest
blunders	mpl flag olt procbody summary
casebuf(casemax)	mpl group
caseitem	mpl group list
caselev	mpl extproc group list
casemax	mpl group
caseptr	mpl extproc group
charsave(maxlit)	mpl physchar
checks(31)	mpl memcheck
codebuf(listsize)	mpl lst-adr out-addr out-byte out-dble out-inst out-lit out-lwl out-word out-wrt
colisons	mpl search summary
consflag	mpl areaattr procbody
consloc	mpl areaattr procbody
cursor	mpl flag physchar
desc(1)	mpl
descold(1)	mpl
dolev	mpl end-stmt extproc list pull-do push-do
domax	mpl push-do
donest(domax)	mpl end-stmt push-do
donumber	mpl bloc gotostmt group prochead push-do
duml(3)	mpl
dum2(3)	mpl
eofflag	mpl sread
errmax	mpl flag
errors	mpl flag olt procbody summary
flagbuf(listsize)	mpl flag
fwdbase(fwdmax)	mpl procbody pull-lex purgefwd save-ref

fwdlex(fwdmax)	mpl pull-lex purgefwd save-ref
fwdmax	mpl save-ref
fwdpc(fwdmax)	mpl procbody pull-lex purgefwd save-ref
fwdptr	mpl extproc procbody pull-lex purgefwd save-ref
hashloc	mpl newentry search statest
hashsize	mpl newentry search
hashtab(hashsize)	mpl newentry search
itembase	mpl areaattr as-stmt dclelemt dechain enchain exp getst get-mode group list listst newentry physchar procbody prochead proc-ref pull-lex purgefwd push-lex putst ref ret-stmt save-ref search s-ref token var
itemmax	mpl getst insertst newentry putst summary
labchain	mpl bloc group labels
labdesc(1)	mpl
labflag	mpl blocksen labels
lastflag	mpl flag summary
lex	mpl bloc blocksen des-sc defst extproc get-lex get-mode gotostmt inc-sc inc-xc labels list newentry procbody prochead pull-lex purgefwd push-lex refst ret-stmt save-ref statest
lexdepth	mpl push-lex summary
line	mpl add-ref dclelemt defst flag labels list sread summary
listbuf(listsize)	mpl list listst print summary
listsize	mpl flag list listst lst-adr out-inst out-wrt print prinwrit summary
listwait	mpl list
litbase(maxlit)	mpl phsychar token
litcursr(maxlit)	mpl physchar token
litlev	mpl physchar rptchar token
lobjx	mpl olt out-rec
maxlex	mpl push-lex
maxlit	mpl token
namchain	mpl dclelemt
namdesc(1)	mpl
numberd	mpl procbody
numberp	mpl dclelemt
objbuf(objsizel)	mpl extproc olt out-rec owrite sread
objbufl(objsizel)	mpl owrite
objbuf2(objsizel)	mpl owrite

objecton	mpl exit owrite
objfcb(13)	mpl exit owrite
objitem	mpl owrite summary
objrecn	mpl out-rec
objsize	mpl extproc olt out-rec owrite sread
objx	mpl olt out-rec
page	mpl prinwrit
pagelin	mpl prinwrit
parchain	mpl dclelemt procbody prochead
pardesc(1)	mpl
pb	mpl memcheck
pc	mpl areaattr blockgen c-exp gotostmt group if-stmt inc-pc initlist initstr labels list lst-adr num-pri out-addr out-byte out-dble out-inst out-lit out-lwl out-word procbody proc-ref purgefwd summary
pccons	mpl areaattr dclelemt
pl	mpl memcheck
proclex(maxlex)	mpl bloc list push-lex ret-stmt
ptr	mpl areaattr constant dclelemt end-stmt gotostmt initstr in-sym konstant num-pri reuse string symbol token
ptrold	mpl reuse token
refs	mpl putst summary
reuselev	mpl reuse token
rptflag(maxlit)	mpl physchar rpt-char token
sc(maxlex)	mpl dec-sc inc-sc inc-xc labels procbody push-lex
sclmax	mpl inc-sc summary
sobuf(sosize)	mpl list physchar sread
sosize	mpl list physchar sread
srefmode	mpl exp num-pri
stacksiz	mpl memcheck summary
stackuse	mpl memcheck summary
stbase	mpl getst insertst listst movest namesame newentry physchar pull-lex putst search
stbase0	mpl memcheck pull-lex
stmts	mpl summary token
symbols	mpl listst newentry summary
tcode	mpl reuse token

tcodeold	mpl reuse token
textmin	mpl dclelemt insertst newentry putst summary
textmin0	mpl summary
titleptr	mpl extproc prinwrit
toggle(255)	mpl flag list listst logchar memcheck out-wrt owrite physchar prinwrit putst settogs sread summary
tokenlev	mpl token
value	mpl areaattr constunt dclelemt initlist num-pri sizeattr string token
varchain	mpl dclelemt procbody
vardesc(1)	mpl
version	mpl
warnings	mpl flag olt
warningt	mpl flag summary
width4	mpl getst listst newentry putst
xc	mpl extproc inc-sc inc-xc procbody summary

```
 1. program ::=
         external_procedure

 2. external_procedure ::=
         entry_name : external_procedure_head procedure_body [EOF]...

 3. entry_name ::=
         identifier

 4. external_procedure_head ::=
         MAIN [procedure] ; |
         procedure_head

 5. procedure ::=
         PROCEDURE |
         PROC

 6. procedure_head ::=
         INTERRUPT [procedure] ( parameter ) ; |
         procedure [( parameter_list )] [simple_size] ;

 7. parameter ::=
         identifier

 8. parameter_list ::=
         identifier [, identifier]...

 9. simple_size ::=
         BYTE |
         WORD |
         DOUBLE

10. procedure_body ::=
         [declare_statement]... [block_sentence]... end_statement

11. declare_statement ::=
         declare declare_element [, declare_element]... ;

12. declare ::=
         DECLARE |
         DCL

13. declare_element ::=
         identifier literally string |
         entry_name type_attribute
         item_list size_attribute [area_attribute]

14. literally ::=
         LITERALLY |
         LIT
```

```
15. type_attribute ::=
         [external] procedure [simple_size] |
         ( konstant ) MICRO [simple_size]

16. external ::=
         EXTERNAL |
         EXT

17. item_list ::=
         identifier [( konstant )] |
         ( identifier [( konstant )] [, identifier [( konstant )]]... )

18. size_attribute ::=
         simple_size |
         POINTER [TO] simple_size |
         BIT ( konstant )

19. area_attribute ::=
         external |
         STATIC |
         CONSTANT string |
         CONSTANT ( [+ | -] konstant [, [+ | -] konstant]... ) |
         initial string |
         initial ( [+ | -] konstant [, [+ | -] konstant]... ) |
         BASED konstant |
         BASED identifier

20. initial ::=
         INITIAL |
         INIT

21. block_sentence ::=
         entry_name : procedure_head procedure_body |
         executable_unit

22. executable_unit ::=
         if_statement |
         unconditional_executable_unit

23. if_statement ::=
         if_clause executable_unit |
         if_clause balanced_executable_unit ELSE executable_unit

24. if_clause ::=
         [label_list] if_then

25. if_then ::=
         IF expression THEN
```

```
26. balanced_executable_unit ::=
        if_clause balanced_executable_unit ELSE
             balanced_executable_unit |
        unconditional_executable_unit

27. unconditional_executable_unit ::=
        [label_list] block |
        [label_list] group |
        [label_list] statement

28. label_list ::=
        [label :]...

29. label ::=
        identifier

30. block ::=
        BEGIN ; procedure_body

31. group ::=
        group_heading ; [block_sentence]... end_statement

32. group_heading ::=
        REPEAT expression [TIMES] |
        DO [do_specification]

33. do_specification ::=
        FOREVER |
        WHILE expression |
        CASE expression |
        identifier replace_op expression TO expression [BY expression]

34. replace_op ::=
        = | :=

35. statement ::=
        null_statement |
        return_statement |
        goto_statement |
        call_statement |
        assignment_statement

36. null_statement ::=
        ;

37. return_statement ::=
        RETURN [expression] ;

38. goto_statement ::=
        go_to label ;
```

```
39. go_to ::=
        GOTO |
        GO [TO]

40. call_statement ::=
        CALL procedure_reference ;

41. procedure_reference ::=
        entry_name [( procedure_argument [, procedure_argument]... )]

42. procedure_argument ::=
        expression |
        array_name

43. array_name ::=
        identifier

44. assignment_statement ::=
        storage_reference assignment_operator expression ;

45. assignment_operator ::=
        = | := | +=

46. expression ::=
        conditional_expression |
        simple_expression |
        storage_reference := expression

47. conditional_expression ::=
        if_then expression ELSE expression

48. simple_expression ::=
        logical_term [or_operator logical_term]...

49. or_operator ::=
        ! | ⊥ | XOR

50. logical_term ::=
        logical_factor [& logical_factor]...

51. logical_factor ::=
        numeric_expression [comparison_operator numeric_expression]...

52. comparison_operator ::=
        < | <= | ^> | = | ^= | >= | ^< | > |
        LLT | LLE | LEQ | LNE | LGE | LGT

53. numeric_expression ::=
        numeric_term [add_operator numeric_term]...
```

```
54. add_operator ::=
        + | -

55. numeric_term ::=
        numeric_factor [multiply_operator numeric_factor]...

56. multiply_operator ::=
        * | / | MOD | MULD | DIVD | SLL | SRA | SRL | SLC

57. numeric_factor ::=
        unary_operator numeric_factor |
        numeric_primary

58. unary_operator ::=
        + | - | ^

59. numeric_primary ::=
        konstant |
        procedure_reference |
        storage_reference |
        @ variable |
        ( expression ) |
        PRTNUM ( entry_name )

60. storage_reference ::=
        reference [$ ( expression [: expression] )]

61. reference ::=
        variable [@ [( expression )]]

62. variable ::=
        identifier [( expression  ]

63. end_statement ::=
        [label_list] END [label | entry_name] ;

64. konstant ::=
        constant |
        string

65. constant ::=
        decimal_number |
        bit_string

66. decimal_number ::=
        digit...

67. bit_string ::=
        " [ [( legal_size )] [legal_digit] ]... "
```

```
68. legal_size ::=
         1 | 2 | 3 | 4

69. legal_digit ::=
         digit | A | B | C | D | E | F

70. digit ::=
         0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

71. string ::=
         ' [non_quote_char]... ' [string]

72. identifier ::=
         alphabetic_character [alphameric]...

73. alphameric ::=
         alphabetic_character | digit

74. alphabetic_character ::=
         lower_case |
         upper_case |
         # | _

75. lower_case ::=
         a | b | c | d | e | f | g | h | i | j | k | l | m |
         n | o | p | q | r | s | t | u | v | w | x | y | z

76. upper_case ::=
         A | B | C | D | E | F | G | H | I | J | K | L | M |
         N | O | P | Q | R | S | T | U | V | W | X | Y | Z
```

```
54 add_operator
    53
74 alphabetic_character
    72,73
73 alphameric
    72
19 area_attribute
    13
43 array_name
    42
44 assignment_statement
    35
45 assignment_operator
    44
26 balanced_executable_unit
    23,26
   BASED
    19
   BEGIN
    30
   BIT
    18
67 bit_string
    65
30 block
    27
21 block_sentence
    10,31
   BY
    33
   BYTE
    9
   CALL
    40
40 call_statement
    35
   CASE
    33
52 comparison_operator
    51
47 conditional_expression
    46
   CONSTANT
    19
65 constant
    64
   DCL
    12
66 decimal_number
    65
```

```
13 declare_element
    11
12 declare
    11
   DECLARE
    12
11 declare_statement
    10
70 digit
    66,73
   DIVD
    56
   DO
    32
33 do_specification
    32
   DOUBLE
    9
   ELSE
    23,26,47
   END
    63
   EOF
    2
63 end_statement
    10,31
3  entry_name
    2,13,21,41,59,63
22 executable_unit
    21,23
46 expression
    25,32,33,37,42,44,46,47,59,60,
16 external                    61,62
    15,19
   EXT
    16
   EXTERNAL
    16
2  external_procedure
    1
4  external_procedure_head
    2
   FOREVER
    33
   GO
    39
39 go_to
    38
   GOTO
    39
```

```
38 goto_statement
    35
31 group
    27
32 group_heading
    31
72 identifier
    3,7,8,13,17,19,29,33,43,62
   IF
    25
24 if_clause
    23,26
23 if_statement
    22
25 if_then
    24,47
20 initial
    19
   INIT
    20
   INITIAL
    20
   INTERRUPT
    6
17 item_list
    13
64 konstant
    15,17,18,19,59
29 label
    28,38,63
28 label_list
    24,27,63
69 legal_digit
    67
68 legal_size
    67
   LEQ
    52
   LGE
    52
   LGT
    52
   LIT
    14
14 literally
    13
   LITERALLY
    14
   LLE
    52
   LLT
    52
   LNE
    52
51 logical_factor
    50
50 logical_term
    48
75 lower_case
    74
   MAIN
    4
   MICRO
    15
   MOD
    56
   MULD
    56
56 multiply_operator
    55
   non_quote_char
    71
36 null_statement
    35
53 numeric_expression
    51
57 numeric_factor
    55,57
59 numeric_primary
    57
55 numeric_term
    53
49 or_operaotr
    48
7  parameter
    6
8  parameter_list
    6
   POINTER
    18
   PROC
    5
   PROCEDURE
    5
5  procedure
    4,6,15
42 procedure_argumment
    41
10 procedure_body
    2,21,30
```

```
6  procedure_head
    4,21
41 procedure_reference
    40,59
1  program

   PRTNUM
    59
61 reference
    60
   REPEAT
    32
34 replace_op
    33
   RETURN
    37
37 return_statement
    35
48 simple_expression
    46
9  simple_size
    6,15,18
18 size_attribute
    13
   SLC
    56
   SLL
    56
   SRA
    56
   SRL
    56
35 statement
    27
   STATIC
    19
60 storage_reference
    44,46,59
71 string
    13,19,64,71
   THEN
    25
   TIMES
    32
   TO
    18,33,39
15 type_attribute
    13
58 unary_operator
    57
27 unconditional_executable_unit
    22,26
76 upper_case
    74
62 variable
    59,61
   WHILE
    33
   WORD
    9
   XOR
    49
```