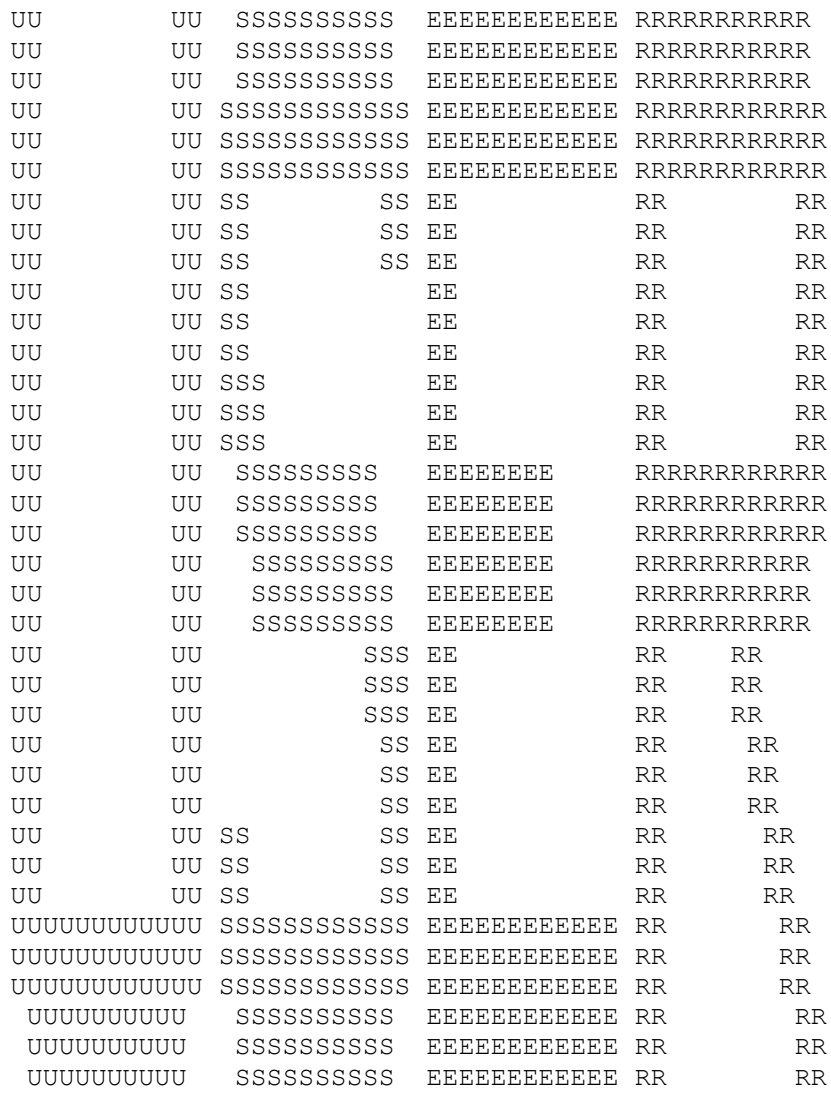
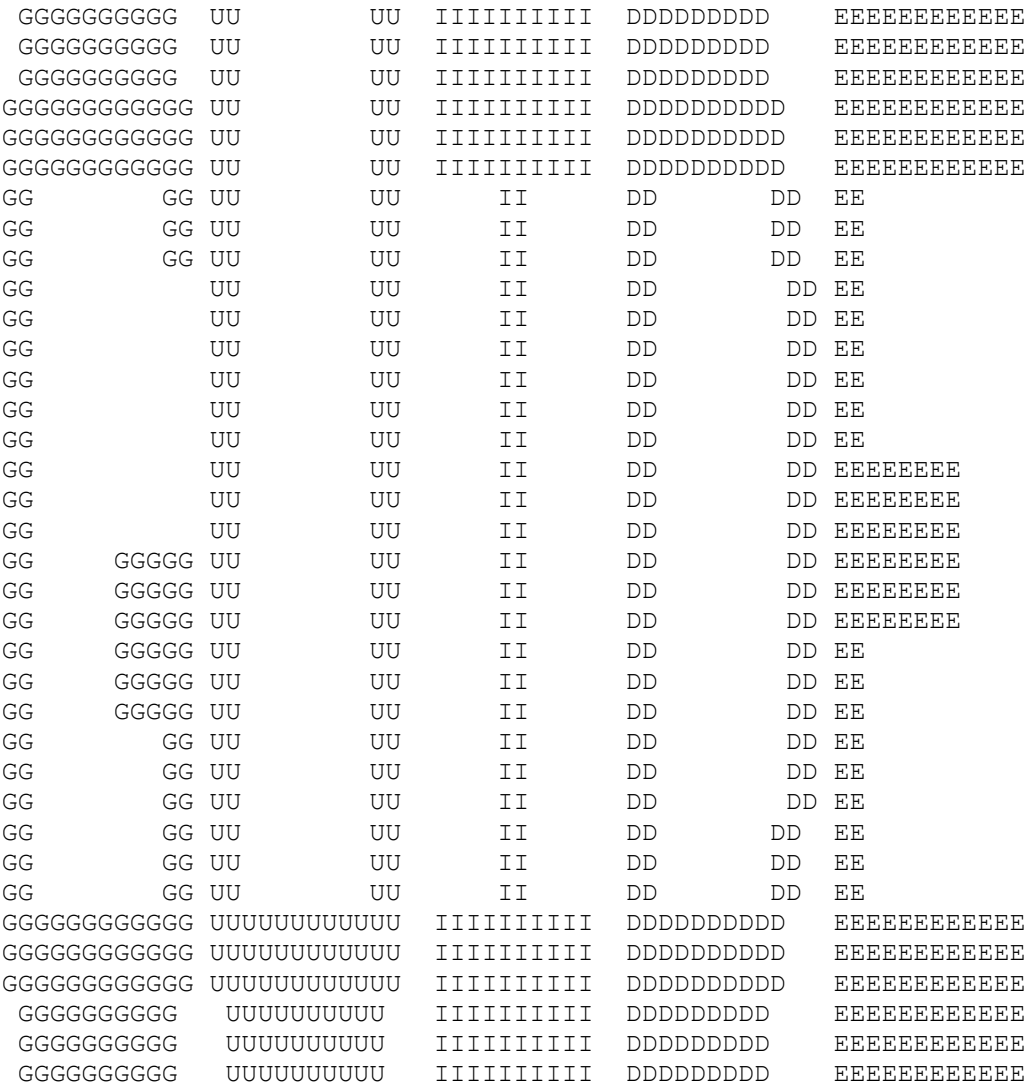
```
 SSSSSSSSSS  PPPPPPPPPPP   AAAAAAAAAA   SSSSSSSSSS  MM        MM
 SSSSSSSSSS  PPPPPPPPPPP   AAAAAAAAAA   SSSSSSSSSS  MM        MM
 SSSSSSSSSS  PPPPPPPPPPP   AAAAAAAAAA   SSSSSSSSSS  MM        MM
SSSSSSSSSSSS PPPPPPPPPPPP AAAAAAAAAAAA SSSSSSSSSSSS MMM      MMM
SSSSSSSSSSSS PPPPPPPPPPPP AAAAAAAAAAAA SSSSSSSSSSSS MMM      MMM
SSSSSSSSSSSS PPPPPPPPPPPP AAAAAAAAAAAA SSSSSSSSSSSS MMM      MMM
SS        SS PP        PP AA        AA SS        SS MMMM    MMMM
SS        SS PP        PP AA        AA SS        SS MMMM    MMMM
SS        SS PP        PP AA        AA SS        SS MMMM    MMMM
SS           PP        PP AA        AA SS           MM MM  MM MM
SS           PP        PP AA        AA SS           MM MM  MM MM
SS           PP        PP AA        AA SS           MM MM  MM MM
SSS          PP        PP AA        AA SSS          MM  MMMM  MM
SSS          PP        PP AA        AA SSS          MM  MMMM  MM
SSS          PP        PP AA        AA SSS          MM  MMMM  MM
 SSSSSSSSS   PPPPPPPPPPPP AAAAAAAAAAAA  SSSSSSSSS   MM   MM   MM
 SSSSSSSSS   PPPPPPPPPPPP AAAAAAAAAAAA  SSSSSSSSS   MM   MM   MM
 SSSSSSSSS   PPPPPPPPPPPP AAAAAAAAAAAA  SSSSSSSSS   MM   MM   MM
  SSSSSSSSS  PPPPPPPPPPP  AAAAAAAAAAAA   SSSSSSSSS  MM        MM
  SSSSSSSSS  PPPPPPPPPPP  AAAAAAAAAAAA   SSSSSSSSS  MM        MM
  SSSSSSSSS  PPPPPPPPPPP  AAAAAAAAAAAA   SSSSSSSSS  MM        MM
         SSS PP           AA        AA          SSS MM        MM
         SSS PP           AA        AA          SSS MM        MM
         SSS PP           AA        AA          SSS MM        MM
          SS PP           AA        AA           SS MM        MM
          SS PP           AA        AA           SS MM        MM
          SS PP           AA        AA           SS MM        MM
SS        SS PP           AA        AA SS        SS MM        MM
SS        SS PP           AA        AA SS        SS MM        MM
SS        SS PP           AA        AA SS        SS MM        MM
SSSSSSSSSSS PP           AA        AA SSSSSSSSSSS MM        MM
SSSSSSSSSSS PP           AA        AA SSSSSSSSSSS MM        MM
SSSSSSSSSSS PP           AA        AA SSSSSSSSSSS MM        MM
 SSSSSSSSS  PP           AA        AA  SSSSSSSSS  MM        MM
 SSSSSSSSS  PP           AA        AA  SSSSSSSSS  MM        MM
 SSSSSSSSS  PP           AA        AA  SSSSSSSSS  MM        MM
```

VERSION 5.0
AUGUST, 1977


UU UU SSSSSSSSSS EEEEEEEEEEEE RRRRRRRRRRR
UU UU SSSSSSSSSS EEEEEEEEEEEE RRRRRRRRRRR
UU UU SSSSSSSSSS EEEEEEEEEEEE RRRRRRRRRRR
UU UU SSSSSSSSSSSS EEEEEEEEEEEE RRRRRRRRRRRR
UU UU SSSSSSSSSSSS EEEEEEEEEEEE RRRRRRRRRRRR
UU UU SSSSSSSSSSSS EEEEEEEEEEEE RRRRRRRRRRRR
UU UU SS SS EE RR RR
UU UU SS SS EE RR RR
UU UU SS SS EE RR RR
UU UU SS EE RR RR
UU UU SS EE RR RR
UU UU SS EE RR RR
UU UU SSS EE RR RR
UU UU SSS EE RR RR
UU UU SSS EE RR RR
UU UU SSSSSSSSS EEEEEEEE RRRRRRRRRRR
UU UU SSSSSSSSS EEEEEEEE RRRRRRRRRRR
UU UU SSSSSSSSS EEEEEEEE RRRRRRRRRRR
UU UU SSSSSSSSS EEEEEEEE RRRRRRRRRR
UU UU SSSSSSSSS EEEEEEEE RRRRRRRRRR
UU UU SSSSSSSSS EEEEEEEE RRRRRRRRRR
UU UU SSS EE RR RR
UU UU SSS EE RR RR
UU UU SSS EE RR RR
UU UU SS EE RR RR
UU UU SS EE RR RR
UU UU SS EE RR RR
UU UU SS SS EE RR RR
UU UU SS SS EE RR RR
UU UU SS SS EE RR RR
UUUUUUUUUUUU SSSSSSSSSSS EEEEEEEEEEEE RR RR
UUUUUUUUUUUU SSSSSSSSSSS EEEEEEEEEEEE RR RR
UUUUUUUUUUUU SSSSSSSSSSS EEEEEEEEEEEE RR RR
UUUUUUUUUU SSSSSSSSS EEEEEEEEEEEE RR RR
UUUUUUUUUU SSSSSSSSS EEEEEEEEEEEE RR RR
UUUUUUUUUU SSSSSSSSS EEEEEEEEEEEE RR RR



SLAC COMPUTING SERVICES
STANFORD CENTER FOR INFORMATION PROCESSING
STANFORD LINEAR ACCELERATOR CENTER
STANFORD, CALIFORNIA 94305

User Note 77

Title:	SPASM User Guide, Version 5.0
Date:	August 15, 1977
Authors:	John Ehrman Paul Dantzig Greg Mushial Mary Artibee
Abstract:	This User Note describes Version 5.0 of SPASM, a fast single pass assembler.
Published by:	SLAC Computing Services of the Stanford Center for Information Processing (SCSCIP), located at Stanford Linear Accelerator Center (SLAC), Menlo Park, California.

TABLE OF CONTENTS

TABLE OF CONTENTS

I. General Information

SPASM is a fast Single Pass ASseMbler (whence the acronym SPASM) designed to accept a subset and a superset of the IBM System/370 Assembler Language. The generated code is assembled directly into core memory, and may be executed or interpreted there. Since the SPASM system is designed with the beginning machine language programmer in mind, it is expected that interpretation will be the normal mode of execution. Facilities are provided to help the beginner over some of the pitfalls inherent in the language, sometimes at the cost of minor restrictions on the source language. Like many student-oriented systems, SPASM provides a batching capability which removes the necessity for returning to the operating system between jobs that typically are at most a few seconds in duration.

The SPASM user is expected to have access to the IBM System/360 or System/370 Principles of Operation manual as a description of the machine code, and the IBM Operating System/360 or OS/VS and DOS/VS Assembler Language manual as a description of the base language. This User Guide is not meant to be a substitute for either of these manuals, but a supplement.

Certain SPASM features may not be available at a particular installation. Each center can choose to include or exclude these features when generating its version of SPASM. Section 1.4 describes the behavior of option-dependent features of SPASM.

Comments, suggestions, reports of errors, and technical inquiries regarding SPASM should be directed to:

John R. Ehrman
SCS-SCIP (Mail Bin 97)
Stanford Linear Accelerator Center
P. O. Box 4349
Stanford, California 94305

1.0 Control Cards

The operating system control cards necessary to use the SPASM system are described below. Each installation may modify the control cards necessary for SPASM; in which case, a local supplement concerning control cards should be published.

For DOS:

```
//  JOB  <jobname>  <local account information>
//      EXEC  SPASM
=JOB (username, etc.)
=SPASM parameters
 - - - source program - - -
=GO parameters      (if execution desired)
 - - - optional data cards if read by program - - -
=SPASM parameters
 - - - another source program - - -
=GO parameters
 - - - more assemblies - - -
/*
/&
```

For OS:

```
//jobname  JOB  (accounting information),'username'
//      EXEC  SPASM
//SYSIN DD *
=JOB (username, etc.)
=SPASM parameters
 - - - source program - - -
=GO parameters      (if execution desired)
 - - - optional data cards if read by program - - -
=SPASM parameters
 - - - another source program - - -
=GO parameters
 - - - more assemblies - - -
/*
```

There are three JCL (Job Control Language) cards needed: JOB, EXEC, and SYSIN DD (under OS). The "/*" card is the usual deck delimiter. Three control cards are recognized by the SPASM System Executive (SEX):

```
=JOB    specifies accounting information
=SPASM  specifies parameters for assembly and execution
=GO     specifies parameters for execution
```

These cards invoke the SPASM assembler and interpreter so that it is possible for a sequence of jobs to run in one batch, rather than submitting each as a separate job to the operating system. The "=" must appear in column 1. The control word (JOB, SPASM or GO) must appear in columns 2 through 71, inclusive. (Cards with "=" in column 1 but no control word are treated as logical end-of-file markers.) A blank must separate the control word from any following information on the card. Other information on the card may not contain embedded blanks.

Parameters are keyword or non-keyword. Keyword parameters consist of a keyword followed by an equal sign and a numeric quantity. Non-keyword parameters consist of 1 to 4 letters (2 to 5, if preceded by "N" indicating negation of the option requested by the parameter). Non-keyword parameters may be abbreviated by the shortest sensible string of letters, providing there is no ambiguity. In the case of ambiguity, SPASM simply uses the first parameter providing a match.

1.1 Parameters Allowed at SPASM System Invocation Time Only

These parameters are specified in the PARM string passed to SPASM by the Operating System. Normally the user will never have to specify these.

ACCT Indicates that accounting should be performed (not available in this version of SPASM).

DBUG Indicates that various system debugging options are to be permitted (overrides the OVLY parameter). DBUG is the default.

OVLY Indicates that the work and save areas of the various internal routines are to be overlaid wherever possible in order to save space (not used in this version of SPASM).

SIZE= Indicates the number of 1K units of core space to be acquired from the Operating System for use by the assembler and interpreter. 5K is always needed for assembler pointers and work areas, in addition to the space requested for individual assemblies. If SIZE=9999 is specified, SPASM will request as much space as is available (up to 4 million bytes). SIZE=20 is the default.

XMAC Indicates that a search is to be made for undefined opcodes in external macro libraries. XMAC is the default.

1.2 Parameters Allowed at Invocation and Assembly Times

ADMP Dump assembled program after assembly complete. NADMP is the default.

ERR= Indicates minimum severity code required for an error message to be printed. ERR=0 is the default.

FIX List machine instruction and address constant fixups. FIX is the default.

GO Execute program unless suppressed by errors. GO is the default.

PRNT Print source listing. PRNT is the default.

SUMP Print Short (tabular) USING Map. SUMP is the default.

SYM List Symbol Table. SYM is the default.

TYPE Perform instruction-operand type checking during assembly. TYPE is the default.

UMAP Print linear USING Map. UMAP is the default.

XPRT List source code of system macros brought from external macro libraries. NXPRT is the default.

XREF Provide Symbol Cross-Reference Table. XREF is the default. (XREF requires that SYM be specified, see below.)

1.3 Parameters Allowed at Invocation, Assembly, and Execution

BTRC Print Branch Trace Table at end of interpretation. BTRC is the default.

COND= Specifies the minimum assembler error severity code required to suppress execution. COND=6 is the default.

DUMP Dump program area after execution is complete. DUMP is the default.

ECOL Is the ecology option. It replaces page ejects with triple spacing, triple with double spacing, and other spacing (except for "+") with single spacing. ECOL is the default.

INTP Specifies execution should begin in interpretive mode (NINTP specifies free-run mode). (Execution may be changed from interpretive to free-running and back through the use of the SPASMFRE and SPASMINT macro instructions.) INTP is the default.

LNPP= Indicates number of lines per page. LNPP=60 is the default.

MINT= Specifies the maximum number of execution-time program interrupts allowed before execution should be abandoned. MINT=50 is the default.

MXLN= Indicates maximum number of execution-time print lines. MXLN=2000 is the default.

TIME= Indicates maximum execution time in seconds. TIME=3 is the default. For OS, the time is CPU (task) time; for DOS, the time is elapsed (clock) time.

The following is a sample SPASM "job".

```
=JOB    (SMEDLEY)
=SPASM  ECOL,COND=1,XPRT
 - - Assembler Language program - -
=GO     INTP,MINT=8
 - - - data deck - - -
```

In the following table the various SPASM parameters are grouped as keyword or non-keyword. Each parameter is listed with its default (if applicable) and the times when it may be specified (where I = Invocation time, A = Assembly time, and E = Execution time).

	Parameter	Default	Specification Times
Keyword	COND=	6	I,A,E
	ERR=	0	I,A
	LNPP=	60	I,A,E
	MINT=	50	I,A,E
	MXLN=	2000	I,A,E
	SIZE=	20	I
	TIME=	3	I,A,E
Non-Keyword	ACCT		I
	ADMP	NADMP	I,A
	BTRC	BTRC	I,A,E
	DBUG	DBUG	I
	DUMP	DUMP	I,A,E
	ECOL	ECOL	I,A,E
	FIX	FIX	I,A
	GO	GO	I,A
	INTP	INTP	I,A,E
	OVLY		I
	PRNT	PRNT	I,A
	SUMP	SUMP	I,A
	SYM	SYM	I,A
	TYPE	TYPE	I,A
	UMAP	UMAP	I,A
	XMAC	XMAC	I
	XPRT	NXPRT	I,A
	XREF	NXREF	I,A

1.4 Option-Dependent Features

If an optional feature is not generated, the action taken by SPASM at invocation of a non-supported option is:

1. If READFILE is not in the system, an error message is issued.

2. If System/370 Instruction Support is not in the system, an error message is issued if the use of any System/370 instruction is attempted.

3. If the Extended Error Message Facility is not in the system, the text portion of error or warning messages is not printed.

4. If the Execution-Time Branch Trace Facility is not in the system, a warning message is issued if "PRINTOUT 20" or "PRINTVAL 20" is attempted, or the BTRC parameter is specified.

2.0 SPASM Features

2.1 Pre-Defined Macro Instructions

Some macro instructions allow the programmer to perform simple debugging and input/output operations at the time the program is being executed. The macro instructions PRINTOUT, PRINTVAL, PRINTLIN, READCARD, READFILE, and DUMPOUT are executed when they are encountered. However, the macro instructions INSTRACE, MONITOR, SPASMINT, and SPASMFRE set internal switches and flags to modify the behavior of the SPASM interpreter.

All but two of the pre-defined macro instructions have two names, so that possible naming conflicts with existing macros can be avoided. The synonymous pairs are:

PRINTOUT	SPASMOUT
PRINTVAL	SPASMPRV
PRINTLIN	SPASMPRT
READCARD	SPASMCRD
READFILE	SPASMRFL
DUMPOUT	SPASMDMP
INSTRACE	SPASMITR
MONITOR	SPASMMON
	SPASMINT
	SPASMFRE
CONVERT	SPASMCVT

The operands permitted for these macro instructions may take several forms and are described in general terms first.

<loc> is a location field symbol appearing in the name field of one of these macro instructions. It must always be a legal symbol and is always optional.

* is the single character "*".

<name> refers to a symbol which is the name of some area of the program, e.g., "A". Note that a <name> may not refer to a symbol defined in a dummy control section (DSECT).

<number> is a self-defining term, e.g., "25" or "C'+'".

<numbname> is either a <number> or a <name>.

<basedisp> is an explicit base-displacement specification of the form "expression(expression)", e.g., "0(4)". Both expressions must be absolute.

<relexpr> is an expression which evaluates to an addressable, relocatable value.

<absexpr> is an expression which evaluates to an absolute value.

<locarg> can be either a <relexpr> or a <basedisp>.

<numbarg> can be either an <absexpr> or a <basedisp>.

2.1.1 PRINTOUT and PRINTVAL (Print Memory Data, Registers, Branch Trace Table, or Terminate Execution)

```
<loc>    PRINTOUT
<loc>    PRINTOUT <numbname>,...,<numbname>
<loc>    PRINTOUT <numbname>,...,<numbname>,*
<loc>    PRINTOUT *
```

The only difference between the PRINTOUT and PRINTVAL macros is that PRINTOUT precedes its printed output with a header line giving the location of the macro call, its statement number, and name. PRINTVAL does not print a header line. In the following description only PRINTOUT will be mentioned, but all comments apply equally to PRINTVAL.

The operand field of the PRINTOUT macro instruction may take the form of a list of <name>s and/or <number>s separated by commas and terminated with a blank or asterisk.

<name> prints the current value of the contents of the named area on the output listing in some useful form, usually hexadecimal. (Items named in dummy control sections are noted as such when the PRINTOUT is executed, but their values cannot be printed since they do not refer to fixed locations in memory.)

The number of characters printed for a <name> depends on the Length Attribute of the <name>. However, it is at most the length of a single print line.

0-15 prints the contents of the specified general purpose register(s).

16-19 prints the contents of the floating-point registers 0, 2, 4, or 6 (respectively).

20 prints the current contents of the Branch Trace Table, the most recent branches taken by the program. This table specifies the "before" and "after" instruction address portions of the PSW, the value of the condition code, and a branch count.

<number> is ignored if other than 0-20.

* terminates execution of the program and returns control to the program supervisor.

Examples:

```
PROUT1   PRINTOUT  0,1,2,3,A
```

will print the contents of general purpose registers 0, 1, 2, and 3, and the contents of the area of memory named A.

```
PROUT2   PRINTOUT  20
```

prints the current contents of the Branch Trace Table.

```
PROUT3   PRINTOUT  ANSWER,*
```

prints the contents of the area of memory named ANSWER and then terminates execution of the program.

```
PROUT4   PRINTVAL  NUM
```

will print the contents of the area of memory named NUM, without any identifying header line.

2.1.2 PRINTLIN (Print a Line of EBCDIC Characters)

```
<loc>    PRINTLIN <locarg>,<numbarg>
<loc>    PRINTLIN <locarg>
```

The PRINTLIN macro instruction prints preformatted lines of EBCDIC characters on the output printer. <locarg> specifies the starting location of the data to be printed. <numbarg> gives the length of characters to be printed on the line. The default print line length (determined when SPASM was generated) is used if <numbarg> is not specified; it will usually be 121 or 133.

The first character of a print line is used for carriage control, as shown below. If the ECOL option is in effect, these vertical spacings are reduced as shown under the column "ECOL Action".

Character	Print Action	ECOL Action
blank	single space	single space
0	double space	single space
-	triple space	double space
1	page eject	triple space
+	no space, print over previous line	no space, print over previous line

Examples:

```
PRINTLIN A,27
```

prints the string of 27 characters, beginning at A.

```
PRINTLIN 0(12),0(13)
```

prints a line whose first character is at the address contained in register 12, and the number of characters printed is given by the operand at the address given in register 13. This mechanism allows for "indirect" specification of line locations and lengths.

2.1.3 READCARD (Read a Card)

```
<loc>    READCARD <locarg1>
<loc>    READCARD <locarg1>,<locarg2>
```

The READCARD macro instruction reads an 80-character card image from the input stream (following the =GO control card) into the area of the program designated by <locarg1>. Because 80 characters are always read, the unwary programmer may run the risk of over-writing parts of his program if the area provided for the card image is too small. <locarg2> is taken as the ENDFILE exit address. If there is no card to be read (a logical or physical end-of-file condition), then control returns to the location given by <locarg2>. If the ENDFILE exit address is invalid, control passes to the instruction following the READCARD macro instruction as if a normal read had been made. If an attempt is made to read past the end of the deck and <locarg2> is not specified, the job is terminated. The presence of an = sign in column 1 (or whatever other character is selected to indicate the presence of a SPASM control card) signals a logical end-of-file condition.

Examples:

To read a card into the area beginning at INCARD and transfer control to ENDECK when no cards are left, we could write:

```
READCARD INCARD,ENDECK
```

To illustrate the use of a <basedisp> to do the same thing:

```
LA    8,ENDECK
LA    9,INCARD
READCARD 0(9),0(8)
```

2.1.4 READFILE (Read a Card from an External Device)

READFILE reads 80-character card images from the data set specified by the DDname SYSFILE for OS and the DLBL statement for DOS. The READFILE instruction is written exactly as was described for READCARD, and its operation is similar. (READFILE will usually be used for instructor-supplied data.)

2.1.5 DUMPOUT (Dump Out an Area of Memory)

```
<loc>    DUMPOUT  <locarg1>
<loc>    DUMPOUT  <locarg1>,<locarg2>
```

The DUMPOUT macro instruction dumps an area of memory during the running of the program, so that the progress of a computation may be followed in detail. The contents of the area is converted to hexadecimal and printed 8 words (32 bytes) to a line, followed by the same 32 bytes printed as EBCDIC characters.

If only <locarg1> is given, the area dumped is the 32 bytes beginning at the first fullword boundary which includes the location specified. If both <locarg1> and <locarg2> are given, the area dumped is the area between the two addresses specified, starting and ending at the nearest enclosing fullword boundaries.

Examples:

```
DUMP1       DUMPOUT RESULT
```

dumps the 32 bytes beginning at the first fullword location does not exceed that of the area named RESULT.

```
DUMP2       DUMPOUT 4(2),85(2)
```

dumps the area beginning 4 bytes after the address in register 2 and ending 85 bytes after the address in register 2.

2.1.6 INSTRACE (Trace Instruction Execution)

```
<loc>    INSTRACE <locarg1>
<loc>    INSTRACE <locarg1>,<locarg2>
<loc>    INSTRACE =OFF
```

INSTRACE controls the tracing of program execution by the interpreter, and is ignored if the program is executing in free-run mode. The effect of tracing is such that after each instruction is executed, the resulting PSW and general registers are printed, showing the effect of having executed that instruction.

If only <locarg1> is specified, the tracing function is performed only if the instruction address during interpretation equals the value of <locarg1>. (This is useful in determining whether control has reached a given location.) When <locarg1> and <locarg2> are both specified, any instruction which lies between the two locations is traced. The operand "=OFF" turns off instruction tracing.

Any INSTRACE instruction overrides the effect of previous ones, so that two separate instructions cannot be traced without also tracing the intervening executed instructions.

Example:

```
          INSTRACE JUMP-16,JUMP+16
```

traces the execution of instructions within 16 bytes of either side of the instruction named JUMP.

2.1.7 MONITOR (Monitor References to an Area of Memory)

```
<loc>    MONITOR  <locarg1>
<loc>    MONITOR  <locarg1>,<locarg2>
<loc>    MONITOR  =OFF
```

The MONITOR instruction checks references to memory to see if they fall within a specified area of memory. If so, the PSW and general registers of the instruction whose execution caused the reference is printed. This feature is useful for finding hard-to-locate bugs, such as an instruction that is inadvertently overwritten or a constant that is modified. If the program is executing in free-run mode, the MONITOR instruction is ignored.

If only <locarg1> is given, MONITOR prints the PSWs of any instructions that reference operands beginning at the exact address <locarg1>. If both <locarg1> and <locarg2> are specified, references to any memory location between the two addresses, inclusive, cause MONITOR printout. "=OFF" suspends all checking for memory references.

Examples:

To determine which instructions make references to memory locations between A and A+7:

```
MONITOR A,A+7
```

To monitor the area of memory pointed to by register 9:

```
MONITOR 0(9)
```

The scope of the monitored area should be kept to the essential minimum since a large amount of output can easily be generated.

2.1.8 SPASMINT (Begin Interpretive Execution)

```
<loc>    SPASMINT
```

A program running in free-run mode may change modes to execute interpretively with the SPASMINT macro. This is helpful in debugging programs containing segments of code that appear to behave badly. Also, for reasons of time and speed, one might only want to interpret a small portion of the program. Any operands on the SPASMINT instruction are ignored. If the program is already executing in interpretive mode, the SPASMINT instruction has no effect.

The speed ratio between interpreted and free-running code varies between about 20 and 90, with the higher figure applying to short and fast instructions.

2.1.9 SPASMFRE (Begin Free-Running Execution)

```
<loc>    SPASMFRE
```

A program running in interpretive mode may change modes to execute freely with the SPASMFRE macro. This is helpful in debugging programs containing frequently executed segments of code that have been fully debugged. Any operands are ignored. If the program is already executing in free-run mode, the SPASMFRE instruction has no effect.

2.1.10 CONVERT

<loc> CONVERT

CONVERT is not implemented in this version of SPASM. It is intended to perform conversions between data types for which there is no "hardware" support such as the instructions CVB, UNPK, etc.

2.2 Assembler Control Options (AOPTIONS Statement)

A number of assembler control options are available under the AOPTIONS Assembler instruction statement. These operand field items may also appear with the prefix "NO" indicating negation of the normal operand's function.

COMMA Allows the appearance of a comma to delimit items in B and X-type constants which normally may only specify a single item. With the COMMA option in effect, one could write "DC X'1,2,3,4'".

ERRORn Restricts printing of error messages to those error of severity equal to or greater than "n" ("n" is a digit from 0 to 9). For example, ERROR3 will print only the error messages of severity 3 or greater. The option NOERRORn inhibits printing errors of severity less than or equal to "n". Thus NOERROR2 has the same effect as ERROR3.

FIX Requests the printing of the Fixup Table.

MTRACE Traces AIF/AGO branches during macro expansion. This option cannot be set by any of the control card parameters.

ROUND Is the usual mode for converting constants of types D, E, F, and H. However, in situations where an unrounded constant is preferred (such as when parts of a multiple-precision number are being converted), use the operand NOROUND.

SYM Prints the Symbol Table at the conclusion of the assembly. (To obtain the cross-reference listing specified by the XREF option, the Symbol Table must be printed also.)

TYPE Checks for possible conflicts between instruction and data types.

UMAP Prints the USING/DROP Table at the conclusion of the assembly.

USEANY Allows the resolution of implied addresses according to currently active USING information, disregarding undefined USINGs. This is at variance with the specifications of the IBM Assembler, which requires that the lowest displacement and the highest numbered register be used. By specifying USEANY, it may be possible to avoid a large number of fixups due to the presence of a single undefined expression in a USING statement.

XMACPR Prints the source statements of external macros as the macro definition is encoded.

XREF Collects the cross-references beginning at this point, unless forbidden by a control card parameter. Normally this will be used to restore XREF collection after it has been turned off by use of the operand NOXREF.

2.3 Extended Branch Mnemonics

All extended branch mnemonics have RR-type equivalents, formed by adding the letter R to the end of the RX-type mnemonic.

2.4 Opcode Definition Modification

Every opcode has a definition stack on which the definition(s) associated with the opcode are kept. All redefinition (except when a DEFER replaces a NULL definition) is non-destructive, that is, during redefinition the previous opcodes on the stack are pushed down, becoming inaccessible until the definition(s) on top of them on the stack are removed. OPDEF, UNDEF and REDEF allow the user to manipulate the definition stack of any opcode (including opcodes OPDEF, UNDEF and REDEF).

Syntax of the definition opcodes:

```
<nfs>   OPDEF  opcode1,opcode2,...,opcoden
<nfs>   UNDEF  opcode1,opcode2,...,opcoden
<nfs>   REDEF  opcode1,opcode2,...,opcoden
```

OPDEF causes <nfs> to become equivalent to each opcode in the operand field. For each opcode a new entry is added to the <nfs>'s definition stack containing a pointer to that opcode. Thus <nfs> is finally equivalent to "opcoden", with "opcode1" at the bottom of its stack.

UNDEF results in the "popping" of the definition stack of each opcode in the operand field. "Popping" means that the top stack entry is removed, thus exposing the definition at the next level down. If there is only one definition on an opcode's definition stack, UNDEF results in that opcode becoming undefined. Referencing an undefined opcode results in a diagnostic.

REDEF causes a NULL definition to be added to the definition stack of each opcode in the operand field. Referencing an opcode with a NULL definition results in the opcode being ignored.

Examples:

```
LOAD     OPDEF LR,LH,L       LOAD EQUIVALENT TO L
         UNDEF LOAD          LOAD NOW EQUIVALENT TO LH

ADD      OPDEF A             ADD EQUIVALENT TO A
         REDEF ADD           ADD BECOMES NULL
```

2.5 Text Deferring Feature (Internal Text Files)

The Text Deferring Feature allows the user to save statement(s) of a source program by building them into a named file and later to have those statements assembled by using the file name as an opcode. Files can be built by adding statements to a file's bottom (DEFER or DEFRL) or top (DEFRS).

Syntax of the DEFER opcodes:

```
<nfs>   DEFER
<nfs>   DEFRL
<nfs>   DEFRS
        DEND   <optional-character-string>
```

where <nfs> (name field symbol) is used as the file name. DEFER, DEFRL, or DEFRS signals the start of deferred text. All the source text between this starting opcode and its matching DEND is added to the DEFER file whose name is given by <nfs>. Subsequent DEFER statements with the same <nfs> reference the same file, unless that file has been made inaccessible through redefinition of <nfs>. (For redefinition, see Section 2.4 on Opcode Definition Modification.)

Usually the occurrence of a symbol in the name field of a DEFER opcode causes the definition stack of that symbol to be pushed down one level before the new definition is added. If, however, the top definition is of type NULL (see the preceding section on Definition Modification), the new definition will replace the NULL definition. This allows the programmer to create multiple files under the same name in a way analogous to the redefinition of a macro.

The starting DEFER opcode (DEFER, DEFRS, or DEFRL) and its matching DEND may be thought of as a pair of statement quotes which suppress evaluation of the source text between them. The only character strings recognized through these quotes are the opcodes DEFER, DEFRL, DEFRS, and DEND. The recognition of one of these does not result in evaluation (i.e., the start or end of a DEFER file), rather the increment (for DEFER, DEFRL, and DEFRS) or decrement (for DEND) of a nesting counter. This allows arbitrary nesting of DEFER statements within a DEFER, since only the DEND which causes the counter to return to its initial value (0), will be recognized as the one which terminates the text-deferring started by the initial DEFER opcode.

The statements generated from a DEFER file are flagged with a "/" character preceding the statement. For example:

```
 *        START NEW DEFER FILE MYFILE
 MYFILE   DEFER
          LA    4,5
          LR    3,4
          AR    3,4
          DEND  MYFILE
 *        ADD THESE TO THE START OF MYFILE
 MYFILE   DEFRS
          ST    3,SAVE
          ST    4,SAVE+4
          DEND  MYFILE
 *        ASSEMBLE THE SAVED STATEMENTS.
 *        MYFILE IS USED AS AN OPCODE TO RETRIEVE THE STATEMENTS
          MYFILE
/         ST    3,SAVE
/         ST    4,SAVE+4
/         LA    4,5
/         LR    3,4
/         AR    3,4
```

There is a known bug in the DEFER feature: source text containing single (i.e., unpaired) occurrences of a character whose representation is X'32' (2-9 punch) will have those single occurrences replaced with blanks when assembled.

2.6 Assembly Listing Features

2.6.1 Forward References and Unknown Values

Because SPASM makes only one pass over the source program, some instructions and statements cannot be completely assembled when they are first encountered. In such cases the incomplete parts of an instruction are replaced by dots in the listing of the partially assembled statement.

For example, if the first executable statement in a program is a branch around a following data area, the assembler output might look like the following:

```
    LOC  OBJECT CODE        ADDR2       STMT     SOURCE STATEMENT

  012468 47F0....           ......         3           B   START
```

where the dots indicate that the parts of the instruction that cannot be assembled will be completed later, when the value of the symbol START is known.

2.6.2 Symbol Table

The Symbol Table which follows the assembly provides useful information about the symbols defined in the program. The four attributes of the symbol (value, relocatability, type, and length) are printed in hexadecimal. If the symbol has been multiply defined, the number of the statement in which it was first defined will be followed by the letters "MD". If some attribute of a symbol is undefined, it will be replaced with dots.

The Relocatability Attribute (RA) of a symbol is determined by the control section origin relative to which it is relocated. Absolute symbols have RA = 0. Symbols appearing in real control sections (CSECTs) have RAs which are numbered beginning at 1. Symbols defined in dummy control sections (DSECTs) have RAs which are numbered starting at 255, counting down by 1 with each additional DSECT.

The Type Attribute (TA) of a symbol reflects the type of statement in which the symbol is defined. The possible TAs (in decimal) and their associated statement types are:

TA	Statement Type	TA	Statement Type
0	Character Constant	12	Q-Type Address Constant
1	Zoned Decimal Constant	13	S-Type Address Constant
2	Packed Decimal Constant	15	External Name
3	Hexadecimal Constant	16	Location Counter Reference (*)
4	Binary Constant	17	DEFERred Text Name
5	Long Floating-Point	18	Machine Instruction Statement
6	Short Floating-Point	19	CSECT or START Statement
7	Fullword Fixed-Point	20	DSECT Statement
8	Halfword Fixed-Point	21	Length Attribute Reference (L')
9	A-Type Address Constant	22	Macro Instruction Statement
10	Y-Type Address Constant	23	Self-Defining Term
11	V-Type Address Constant	24	LTORG Statement

If the symbol names an area of memory defined by a DC or DS statement with a length modifier, then X'20' (decimal 32) will be added to the TA to indicate that the length is explicitly defined. Thus the statement:

```
ABLE   DC    BL2'101'
```

could cause the symbol "ABLE" to have a TA of X'24'.

Example of Symbol Table output:

```
    SYMBOL  VALUE  LNTH RA TA DEFN/M REFERENCES

   ABLE     27A346 0002 01 24     4
   BAKER    27A348 0004 01 07     5MD     6     6
   DUMMHEAD 000000 0001 FF 14    14
   DUMVAR2  00000F 0002 FF 20    16
   FORWARD  27A35C 0004 01 07    11       7     8    13
   HWORD    27A340 0002 01 08     2       3
   MISSING  ...... .... .. ..             2
   .PRIVATE 27A340 0001 01 13     1
```

2.6.3 USING Maps

Because errors in the application of USING statements can be difficult to find, SPASM provides listings at the end of the assembly (under control of the UMAP and SUMP parameters) of all USING statements in the program. The information in the USING Maps includes the register specified as a base register, the Value and Relocatability Attributes of the expression assigned to that register, and the starting and ending numbers of the group of statements to which that USING may be applied for purposes of resolving implied addresses into base-displacement form.

The USING Map provided by the UMAP parameter provides its information in linear form, while the Short USING Map specified by the SUMP parameter provides a table showing the values in all active base registers for the entire program, except that the Relocatability Attributes are omitted.

Example of Using Map (UMAP) output:

```
     REG      START-SN  DROP-SN  RA  VALUE

      12          10      END    01  27A35A
      15           1        9    01  27A340
```

2.6.4 Fixup Table

Statements which contain references to undefined symbols generate fixups. Fixups are segments of code whose assembly is completed only after the end of the program is reached. (This is because SPASM is a one-pass assembler rather than a two-pass assembler.) For each statement which generates a fixup, the Fixup Table lists the location, the generated code, the address fields (if applicable), the statement number, and an indication of the type of the field being completed. Note: Errors may be detected during fixup time, so check the Fixup Table listing carefully for error messages.

Example of Fixup Table output:

```
   LOC    RESOLUTIONS  ADDR1  ADDR2  STMT    SUBFIELD(S)

   27A350 ....F01C             27A35C    7    S2
   27A354 ..03F01CF020 27A35C 27A360    8    S1 L  S2
```

2.6.5 Type Checking

This feature informs the user (via warning messages) of potential conflicts between instruction and data types. For example, if HWORD is defined as "DS H" and the source contains the statement "ST 2,HWORD", a warning is issued since the type of the instruction (fullword) and the type of the data (halfword) are in conflict.

Example of Type Checking output:

```
LOC    OBJECT CODE  ADDR1  ADDR2  STMT SOURCE STATEMENT

27A340 0004                          2 HWORD    DC    H'4'
************* WARNING MIF-52-3,STMT=3,COLUMN=23,OPERAND=2,SUBFIELD=8
              OPERAND NOT IMPLICIT FIXED FULLWORD
27A342 5020F000             27A340   3          ST    2,HWORD
************* WARNING MIF-68-3,STMT=9,COLUMN=25,OPERAND=2,SUBFIELD=7
              OPERAND NOT PACKED DECIMAL
27A364 4F00C002             27A35C   9          CVB   0,FORWARD
```

3.0 Execution-Time Conventions

3.1 Conventions for Interpretive Mode

3.1.1 Limitations

Privileged Instructions

Privileged instructions are treated as invalid opcodes, and an appropriate interruption message is printed. Even if a branch is implied (e.g., for LPSW), the next sequential instruction is executed. Note: The SVC instruction is treated as a privileged instruction, as well as instructions pertaining to special features, such as Monitor Call, Store Clock, etc.

Extended Precision Floating-Point Instructions

The extended precision floating-point instructions are not interpreted. If the machine itself does not contain the extended precision feature, their use will cause a program interrupt.

Byte-Oriented Operand Feature

The SPASM interpreter does not support the System/370 Byte-Oriented Operand Feature.

3.1.2 Additions

Handling of Illegal Instruction Addresses

One of the most notable oversights in the design of System/370 is that illegal instruction branch addresses are determined at instruction fetch time rather than at the time the branch address is known. This leads to situations where the programmer has no idea how his program arrived at a given location, and the interruption mechanism of System/370 can do no more than to tell him that he's in the wrong place. To help diagnose this rather common programming fault, the SPASM interpreter will print a special error message when such an invalid branch is detected and ignore the branch instruction. This may lead to later faults in the program logic, but provides a guard against the possibility of trying to interpret instructions from areas well outside the program area.

3.1.3 Initial Register Settings

When the initial instruction of the program is interpreted, registers 13, 14 and 15 are initialized as follows:

R13 Save Area (18-word area aligned on a fullword boundary)
R14 Return Address (branch address upon termination of execution)
R15 Entry Address (address of the first executed instruction)

The contents of the other registers at program initiation are unknown.

3.1.4 Error and Interruption Handling by the Interpreter

When an error condition is detected during the interpretation of a program, every attempt is made to continue. This means that the program may produce many error messages of which only the first has any significance; however, other program bugs can sometimes be found by continuing to execute. There is a built-in maximum number of program interruptions (specifiable with the MINT parameter), after which execution terminates.

Other error conditions are handled as follows:

- If the instruction address is allowed to run off the end of the program area, there is no way to tell where to go next, so execution is terminated.

- If the parameters of one of the macro instruction expansions are undecipherable, the program has probably overwritten itself. An attempt is made to try to recover and continue execution, in which case the actions requested by the macro instruction will probably be lost or incorrect.

- If there are "memory-protect" errors (attempts to access data outside the program area), the instruction will generally be ignored. The TR and TRT instructions are treated specially by checking the address of each byte fetched from the translate table.

- Program interruption codes 7 to 15 are generally detected by the hardware, so that the instruction is usually in an undetermined state of completion. The other interruptions are usually detected by the interpreter, so that the instruction is not executed and the contents of registers or memory areas are not changed.

- If a branch to an invalid address is attempted, the branch will be ignored and execution will continue with the next sequential instruction.

3.2 Conventions for Free-Run Mode

3.2.1 Register Settings

As in the case of interpretive execution, registers 14 and 15 are preset to the return and entry-point addresses respectively. Register 13 contains the address of a standard 18-word save area into which the user's program may store its registers in the usual way.

When control is returned to SPASM, register 14 is used as a temporary base register. Hence the value of R14 which appears in the "Final PSW and REGs" on the output listing is 2 greater than the original return address. In addition, the final PSW value is unknown.

3.2.2 Error and Interruption Handling in Free-Running Code

Error conditions generated by free-running code are more difficult to handle, so the program should be thoroughly debugged before using the free-run option.

- Branch errors cannot be detected before the branch occurs so these errors are usually fatal.

- When a program interruption occurs outside the bounds of the user's program, SPASM assumes that control has been lost and terminates execution.

- When a specification error is caused by an odd instruction address in the PSW, SPASM sets the low-order bit of the PSW Instruction Address to zero and attempts to continue execution. This may cause further errors.

- The INSTRACE and MONITOR instructions will not cause instruction tracing or memory monitoring in free-running code.

- A bad ENDFILE address on the READCARD or READFILE macro instructions causes program termination in a free-running program if an end-of-file condition is detected.

Note: The interpretive mode of execution may be entered and exited with the SPASMINT and SPASMFRE macro instructions.

II. Reference Information

4.0 Things SPASM Does Differently from IBM's Assemblers

4.1 USING Registers

Because of the one-pass nature of SPASM, it is necessary to avoid the inherent two-pass aspect of the base register USING algorithm found in IBM Assemblers. Thus, while it might seem quite natural to write the statements:

```
        USING *,REG
REG     EQU   7
```

SPASM requires these statements be given in the reverse order, so that the value of the register is known when the USING statement is encountered. The appearance of an unknown quantity in a USING statement requires that the resolution of all implied addresses following that statement be deferred until the value has been resolved. This means that virtually all implied addresses would be saved for fixup at the end of the program, thereby greatly increasing the time and memory requirements of the program. Therefore, SPASM gives an error message whenever a USING statement uses an undefined register.

4.2 USINGs with Implied Addresses

Since the use of implied addresses is very common in places where self-defining terms appear (such as in the statement "LA 2,10" where "10" is an implied address), it is assumed when the first operand of a USING statement is undefined, it is relocatable. Otherwise SPASM would need to save all such USINGs for final resolution of the implied operands at the end of the program.

4.3 DROP Extension

A DROP statement with no operands drops all currently active base registers.

4.4 Code Overlays

If the program causes text to overlay itself through the use of ORG statements, incorrect results may be obtained if the overlaid text contains fixups. For example, the statements:

```
        LA   3,A
        ORG  *-4
        BC   15,B
```

can cause an erroneous branch if the address implied by A requires a fixup but that implied by B does not. After the fixup, the branch instruction would become:

```
BC 15,A
```

(IBM assemblers generate text in which the second instruction completely replaces the first.)

4.5 Alternate Statement Format

The Alternate Statement Format, supported by IBM Assemblers for macro statements, is supported by SPASM for all statement types. The Alternate Statement Format allows an operand field to be continued onto the next card if the operand is terminated by a comma followed by a blank (comments may follow the blank). A non-blank character must be placed in the column following the end column. The operand field continues in the continue column of the next card.

There is one very minor restriction on the use of Alternate Statement Format. If the string "L'*" appears anywhere in the operand field of a statement, that statement is not considered to be in the Alternate Statement Format and is handled accordingly. This is because L'* may be interpreted in one of two ways depending upon the type of statement in which it appears. In a machine instruction, L'* is a Length Attribute reference. In a macro instruction, it could be part of a quoted string being passed as an operand. (Note: IBM Assemblers avoid this problem by considering L'* as a part of a string if it appears in a macro call, but as a Length Attribute reference if it appears in an assembler or machine instruction statement. Because SPASM allows dynamic redefinition of opcodes, SPASM cannot tell which kind of statement it is until the operand field has been scanned and the opcode has been processed.)

4.6 Resumed CSECTs

Due to the one-pass nature of SPASM, resumed CSECTS are not assembled contiguously if other CSECT(s) come between the sections of the resumed CSECT(s). DSECT resumption is handled correctly.

4.7 Expressions

An expression may not consist of more than 16 terms, 5 levels of parentheses, or 15 operators. Unary minuses are allowed in expressions.

4.8 Continued Comment Statements

SPASM does not allow comment statements to be continued. In comment statements it is almost invariably the case that a non-blank character in the continuation column (column 72) is a mistake.

4.9 Blank Control Section Name

If a blank control section name is used (on a START or CSECT statement, or if the code is uninitiated), then the name of the control section appears in the Symbol Table as ".PRIVATE".

4.10 Code From Pre-Defined Macro Instructions

The following brief description of the actual code generated by the macro expansions should help to debug programs from core dumps and to understand the mechanism used for providing macro instruction services. The basic set of instructions generated by any of the SPASM pre-defined macro instructions described in Section 2.1 is:

```
          CNOP  2,4           ALIGN TO MIDDLE OF A FULLWORD
<loc>     STM   14,15,*+18    SAVE REGISTERS 14 and 15
          L     15,*+10       GET POINTER TO PROCESSOR ADDRESS
          L     15,0(,15)     GET PROCESSOR ADDRESS
          BALR  14,15         BRANCH TO PROCESSOR
          DC    X'n',AL3(processor-pointer)
          DS    2F            SAVE AREA FOR REGISTERS 14 AND 15
          <parameter-list>
          LM    14,15,4(14)   RESTORE REGISTERS 14 AND 15
```

The quantity "n" is a number between 1 and 11 specifying which of the macro instructions is being expanded. <parameter-list> depends on the particular macro instruction being expanded; it is an encoding of the information in the original operand list. There are two basic forms: the first for PRINTOUT and PRINTVAL and the second for the others.

For PRINTOUT and PRINTVAL, each list entry has the form:

```
DC AL1(type),AL3(tablecode)
```

where "type" specifies the nature of the list entry and "tablecode" tells where to find the relevant information in the Symbol Table.

For the other macro instructions, the list entries have the form:

```
DC XL2'type',S(argument)
```

where "type" specifies the nature of the list entry, and "argument" gives the data necessary to compute the value of the operand.

Thus the operands of the macro instructions (other than PRINTOUT and PRINTVAL) should be valid for S-type address constants. Fixups for these constants will occasionally appear in the Fixup Table if the FIX parameter is in effect.

4.11 Macro Language Differences

Macro language restrictions and features effective in SPASM include:

- System macros may be referenced if the XMAC parameter is specified, and access is provided through appropriate JCL statements referencing one or more macro libraries. (Listing of such macros can be controlled with the XPRT parameter or the AOPTIONS statement's XMACPR operand.)

- Only the K, L, N, and T attributes are recognized. (That is, the I and S attributes are not recognized.)

- If a relocatable symbol is used in an expression which must be evaluated by the Macro Expander (MXP), it is flagged with a warning diagnostic and the value of the symbol is used. No check is made to determine if Relocatability Attributes have been combined properly.

- Because of the one-pass nature of SPASM, expressions evaluated by the Macro Expander must not refer to symbols which are undefined when the evaluation occurs. (The Macro Expander must evaluate all expressions in conditional assembly statements and all variable symbol subscripts.) This restriction also applies to variable symbol references whose value is a symbol undefined at time of expansion. Note that this restriction does not apply to expressions appearing in model statement expressions except for variable symbol subscripts.

- Tracing of the flow of control during macro expansion is controlled by the MTRACE operand of the AOPTIONS statement (see Section 2.2).

5.0 Things SPASM Doesn't Do that IBM's Assemblers Do

- No capability exists for conditional assembly in open code.
- No provision has been made for use in open code of attributes other than L (length).
- CXD, DXD, EXTRN, WXTRN and ENTRY are not supported.
- The COM and CCW instructions are not supported.
- Q-type and V-type address constants are treated as A-type.
- L-type constants are not supported.
- Bit length specification in DC statements is not supported.
- The COPY, REPRO and PUNCH statements are not supported.

6.0 Things SPASM Does that IBM's Assemblers Don't (or Didn't)

6.1 Self-Defining Terms

Self-defining terms may have values up to a full word (32 bits) in length. The maximum value is X'FFFFFFFF'. (Some of the IBM Assemblers restrict values to 24 bits.)

6.2 DC and DS Statements

The syntax of the DC and DS statements has been expanded to allow the use of modifiers containing the name field symbol and the location counter reference (*). When this occurs, the modifiers containing the "*" are rescanned for each duplication of the operand, so that constants of varying lengths, scale modifiers, and exponent modifiers may be generated. For example, the following statement generates a table of the first ten powers of ten:

```
TBL     DC    10EE((*-TBL)/L'TBL)'1.0'
```

This example also illustrates the fact that SPASM allows the Length Attribute of the name field symbol to appear anywhere after it is logically known.

The lack of a data field in an operand in a DC statement causes the operand to be treated as a DS-type operand and a warning message to be issued. Thus the statement:

```
        DC    X'1',X,X'2'
```

causes a byte to be skipped between the two constant bytes, as though the three statements:

```
        DC    X'1'
        DS    X
        DC    X'2'
```

had been written.

6.3 ICTL and ISEQ Statements

Multiple ICTL and ISEQ statements are allowed. The maximum allowable number of characters in the sequence field is 8. Longer fields are truncated for sequence-checking purposes. Any errors in the operands (or null operand fields) cause the appropriate quantities to be reset to the standard default values (1, 16, and 72 for ICTL, and 73-80 for ISEQ).

6.4 Assembler Debugging Instructions

There are a number of assembler debugging instructions which perform various debugging operations (e.g., dynamic dumps of the program area, Symbol Table, and Fixup Table), as well as tracing the flow of control among assembler routines and dumping common control areas in the process. See the comments at the start of SPASM routine DBG for further information.

6.5 Length Attributes of Self-Defining Terms and Location Counter

The use of Length Attributes of self-defining terms and location counter references is allowed (e.g., L'*, L'5, and L'L'A) The Length Attribute of a literal is invalid.

6.6 Macro Language Extensions

- A macro definition may occur at any point in the source program.
- Symbolic parameter sublists may be nested to any level.
- References to the attributes of the value of a SETC symbol may be made and they are processed in the same way as references to symbolic parameter attributes.
- SETC symbols and symbolic parameters may be any term which is valid for SPASM when used in arithmetic expressions to be evaluated by the Macro Expander. Thus in "&A SETC &C", the character string corresponding to &C is not required to be a string of decimal digits. It may be a string representing any evaluatable term.
- The value of T'&SYSECT is either the string "CSECT" or "DSECT" depending on the type of the current control section (whose name is the value of &SYSECT).
- IBM Assembler Language syntax relaxations are allowed in a few minor cases. Within expressional parentheses, blanks may be used anywhere. Keyword parameters may appear in any order, and anywhere in the formal and actual parameter lists. References to &SYSLIST may be made in keyword macros. The leading & may be omitted from the variable symbol name in a declaration list. &SYSLIST(0) may be used to refer to the name field operand of the macro call statement.
- System variable symbols &SYSTIME, &SYSDATE, and &SYSDAY have the values that appear at the top of each page of the assembly listing. &SYSNAME is the name of the macro currently being expanded, and &SYSNEST is the current macro nesting level.

- MNOTE severity may be indicated by any self-defining term (not just decimal self-defining terms as in IBM Assemblers).

- Macro quotes are provided. Left and right macro quotes are indicated by the operations ALMQ and ARMQ. When a statement whose operation field contains ALMQ is encountered, subsequent lines are treated as pure text (no substitutions or evaluations are done) until a corresponding ARMQ statement is encountered. Macro quotes may be nested. No substitution is made for variable symbols appearing within macro quotes. At expansion time, the outermost macro quotes are stripped off, and the lines between them are generated exactly as they appeared in the macro definition. Thus a macro may generate a macro definition.

Example of a MACRO defining a MACRO:

```
         MACRO
         MAKEMAC  &NAME
         LCLC  &TEMP,&MNOT
&TEMP    SETC  '&&'(1,1)
&MNOT    SETC  'MNOTE'  (SO MNOTE WON'T BE RECOGNIZED IMMEDIATELY)
         ALMQ
         MACRO
         ARMQ
&TEMP.L  &NAME
         MNOTE *,'GENERATING &NAME'
         &MNOT *,'THIS IS &NAME, AT &TEMP.L'
         ALMQ
         MEND
         ARMQ
         MEND
         MAKEMAC GLOTZ
HERES    GLOTZ
         MAKEMAC FLOOP
THERES   FLOOP
```

6.7 Extended EQU Syntax

```
<nfs>     EQU  <expr1>,<expr2>,<expr3>
```

The syntax of the EQU instruction's operand field allows the specification of up to three expressions separated by commas. <expr1> is required and can be any valid assembler expression. <expr2> and <expr3> are optional and, if specified, must be absolute. If <expr2> is specified, its value (if less than 65536) will be used as the Length Attribute (LA) of <nfs> (the name field symbol). If <expr3> is specified, its value (if less than 256) will be used as the Type Attribute (TA) of <nfs>. In the absence of <expr2> and/or <expr3> (or if there is an error in <expr2> or <expr3>), the LA and/or TA of <nfs> is determined from <expr1> in the usual way. For example:

```
A    EQU   10            A HAS VA=10,RA=0,LA=1,TA=23
A    EQU   10,2          A HAS VA=10,RA=0,LA=2,TA=23
A    EQU   10,,1         A HAS VA=10,RA=0,LA=1,TA=1
A    EQU   10,2,1        A HAS VA=10,RA=0,LA=2,TA=1
```

6.8 PRINT INNER and PRINT DATA

PRINT INNER and PRINT NOINNER are SPASM extensions which allow the user to specify that inner macro calls with their actual parameters are to be printed during the expansion of any macro. PRINT INNER and PRINT GEN are independent of each other, so that inner macro calls may be printed without printing the generated code that may accompany the expansion of the inner macro. The default is PRINT NOINNER. Generated code is flagged with a "+" preceding column 1; inner macro calls are flagged with "-".

PRINT DATA and PRINT NODATA are accepted but ignored.

6.9 MALIGN

MALIGN controls macro-generated text. This opcode takes three numeric operands, and is processed in the same way as the ICTL statement. The operands (op1, op2, op3) are used to set the operation, operand, and comment columns for the generated text. The default columns are 10, 16, and 40. The operands are checked to guarantee that:

$$3 \le op1 \le 40$$
$$9 \le op2 \le 50$$
$$19 \le op3 \le 60$$

In addition, the operand field must start at least 6 columns to the right of the operation field, and the comment field must start at least 10 columns to the right of the operand field. That is:

$$op1 + 6 \le op2$$
$$op2 + 10 \le op3$$

6.10 Literals

Since a literal is actually a symbol (it simply has the side-effect of generating a constant), SPASM creates a dummy symbol of the form ".LITnnnn" for each literal encountered. "nnnn" is a unique number for each literal generated. Since literals are treated as symbols, they appear in the Symbol Table and cross-reference listing.

6.11 LTORG Statement

If a name field symbol does not appear on a LTORG statement, the location counter is not automatically aligned with a doubleword boundary. The generated constants will fall on whatever boundaries are required, with those having the most stringent alignment requirements appearing first.

LTORG may appear in a dummy control section, and any literals generated will therefore have dummy names. However, literals generated following the END statement will appear in whatever control section is in effect at that point in the program.

6.12 S-Type Address Constants

S-type address constants are allowed in literals, however a low-level diagnostic is issued. The base-displacements of implied operands are resolved with respect to the USING statements in effect at the time the constant is generated (after LTORG or END), not those in effect at the point where the constant is used.

6.13 END Statement Operands

The operand of the END statement may be a symbol or an expression.

III. SPASM Diagnostic and Error Messages

7.0 How to Interpret the Messages

This is a brief summary of the error messages printed by the SPASM assembler. Error messages are of two basic forms:

```
********** ERROR   identifier-errorcode-severity

********** WARNING identifier-errorcode-severity
```

where:

identifier is a three letter sequence identifying the functional routine in SPASM that issued the diagnostic message. It usually has some mnemonic significance; for example, errors detected by the constant processor have the identifier "KON".

errorcode is a number from 0 to 999 identifying the specific error. The lists below provide some explanation of what went wrong.

severity is a single digit. If severity is 3 or less, SPASM flags the message as a "WARNING" rather than an "ERROR". If severity is 5 or greater and the error applies to a machine instruction, the instruction is "zeroed". An approximate description of the significance of each severity level is:

0 There may be an incompatibility with an IBM assembler, but SPASM will handle it correctly.

1 There is some minor condition that should be checked.

2 There is some minor error (non-fatal).

3 There is an error that is not fatal to assembly, but which should be checked carefully.

4 There is a serious error, but the statement might work.

5 The error is bad enough that the statement probably will not work correctly.

6 The error is bad enough that the statement is useless.

7 The error may cause the location counter to be lost, and it may be reset by some default action to a useable though incorrect value.

8 The space available to the program and its associated tables is insufficient. No further code is emitted, but the tables are permited to grow into the program area to allow complete program checking.

9 Space needed for tables is unavailable. Assembly will continue with the information at hand.

Most error messages are printed immediately before or after the statement containing the error. Some errors, however, are detected only during the final fixup phase, so the Fixup Table should be checked carefully.

Whenever possible, additional information is printed along with the basic error message. (The Extended Error Message Facility prints explanatory text on the line below the error message.) This includes the column nearest the point where the error was detected, the operand number in the statement (if the error is in the operand field), the statement number, the subfield number, and so forth. Under most circumstances, this added information will be accurate; but occasionally only the basic error message will be accurate.

Some of the error messages include a subfield designator. This is of the form "SUBFIELD=xx", where "xx" can have the following forms:

A	refers to A-type address constant.
B1,D1,B2,D2	refer to the base and displacement of the appropriate operand.
DF	refers to the duplication factor.
EM	refers to the exponent modifier.
L,L1,L2	refer to the length fields of SS-type instructions.
LM	refers to the length modifier.
M1	refers to the mask field of a conditional branch instruction.
R1,R2,R3	correspond to the register operands of the erring machine instruction.
S1,S2	refer to implied addresses in the appropriate operand fields.

SB,SD	refer to the base and displacement of an S-type address constant.
SI	refers to an implicit S-type address constant (usually from macro instructions).
SM	refers to the scale modifier.
X2	refers to the index register of the instruction.
Y	refers to a Y-type address constant

7.1 Diagnostic and Error Messages

AEX - ASSEMBLER EXECUTIVE

AEX-0-0 No ADMP, zero length program

DBG - ASSEMBLER DEBUG ROUTINE

DBG-0-0 Bad dynamic patch

DEF - DEFINITION ROUTINE

DEF-10-6 No name field on OPDEF
DEF-20-6 Operand field missing
DEF-21-4 Invalid character in operand field
DEF-22-4 Operand length invalid
DEF-23-4 Undefined opcode in operand field
DEF-30-2 Assembler or machine opcode modified
DEF-40-6 Circular definition created

DFR - TEXT DEFERING ROUTINE

DFR-10-1 Illegal name field on "DEND"
DFR-20-5 EOF encountered, matching DENDs generated
DFR-30-9 Out of space, text not saved
DFR-40-2 Unmatched DEND
DFR-50-6 No name or bad name on DEFER statement
DFR-60-1 Operand field on DEFER card
DFR-70-2 Primitive opcode redefined

EQU - EQU PROCESSOR

EQU-1-6 Name Field Symbol (NFS) missing
EQU-2-6 Operand 1 undefined (NFS undefined)
EQU-3-6 Operand 1 missing (NFS undefined)
EQU-4-3 Invalid delimiter (NFS defined)
EQU-5-3 Expression 2 (expr2) invalid or relocatable - expr1's Length Attribute (LA) used
EQU-6-3 Specified LA > 65536 - expr1's LA used
EQU-7-3 Expr3 invalid or relocatable - expr1's Type Attribute (TA) used
EQU-8-3 Specified TA > 255 - expr1's TA used
EQU-9-3 Scan error in expr2 or 3, expr1's LA or TA used
EQU-10-6 Expr1 is complexly relocatable

EVL - EXPRESSION EVALUATOR

EVL-1-2 Fixed-point overflow in add or subtract
EVL-2-2 Multiply overflow
EVL-10-4 Relocatability error in multiplication or division

EXP - EXPRESSION SCANNER

EXP-5-0 Unary operator appears in an expression
EXP-10-5 Illegal character
EXP-30-5 Missing term
EXP-31-5 Missing operator
EXP-32-5 Unbalanced parentheses
EXP-33-5 Location counter reference (*) used in multiply or divide operation
EXP-34-5 Invalid delimiter
EXP-40-5 Too many terms
EXP-41-5 Too many levels of parentheses
EXP-42-5 Too many operators
EXP-50-5 Illegal literal
EXP-51-5 Nested literals not supported by SPASM

EXT - CONTROL SECTION ROUTINE

EXT-10-2 Resumed CSECT not assembled contiguously
EXT-20-6 More than 256 CSECTs and DSECTs
EXT-30-3 Illegal START card
EXT-40-5 Bad name field symbol, statement ignored
EXT-50-5 Attempt to switch section types, statement ignored

FIX - FIXUP ROUTINE

FIX-1-5 Undefined expression
FIX-2-5 Address constant undefined
FIX-3-5 Illegal use of complex relocatability

KON - CONSTANT PROCESSOR FOR DC, DS, AND LITERALS

Error messages from "KON" fall into two groups: those detected during the scanning of the operands (errorcode of 200 or greater), and those found during the conversion of a constant.

Scanning Errors

KON-200-5	Bad operand
KON-201-4	Relocatable duplication factor, set to 1 instead
KON-202-4	Duplication factor contains *, set to 1 instead
KON-203-3	Negative duplication factor, set to 1 instead
KON-204-1	Literal has duplication factor zero
KON-210-5	Invalid constant type
KON-220-5	Invalid character in length modifier
KON-221-4	Relocatable length modifier, default used instead
KON-222-3	Erron in length modifier, default used instead
KON-230-5	Invalid character in scale modifier
KON-231-4	Relocatable scale modifier, zero used instead
KON-232-1	Invalid scale modifier (ignored)
KON-233-3	Scale modifier out of range, zero used instead
KON-240-5	Invalid character in exponent modifier
KON-241-4	Relocatable exponent modifier, zero used instead
KON-242-1	Invalid exponent modifier (ignored)
KON-243-3	Exponent modifier out of range, zero used instead
KON-300-5	No data text in a literal
KON-301-0	No text in a DC operand, treat as DS-type instead
KON-310-5	Excessive location counter increment (ignored)
KON-320-7	Location counter increment ran off bottom of the program, reset to start of the program area
KON-330-5	Invalid operand delimiter
KON-331-2	Vacuous comma after an operand (ignored)
KON-332-5	Invalid data delimiter
KON-340-5	Excessive literal length
KON-500-5	Invalid expression found in scanning a field
KON-501-4	Unevaluatable expression found by EXP, some default action was taken (usually set to zero)

Conversion Errors

To determine the precise causes of conversion errors, the errorcode must be examined as a binary 8-bit pattern. The rightmost 3 bits define a number between 1 and 7. Errors 1-3 are fatal; 4-7 allow some degree of continuation. The converted value is set to zero. The leftmost 5 bits define a number of non-fatal errors. For example, a KON errorcode of 23 is decoded as 7 (floating-point characteristic out of range) plus 16 (too many decimal points); the resulting value is set to zero.

Bits 5, 6, and 7 (low-order 3 bits)

1	Invalid single ampersand
2	Invalid character
3	Invalid type conversion requested
4	Vacuous input text (zero used)
5	Excessive decimal exponent
6	Null decimal exponent where one was expected
7	Floating-point characteristic out of range

Bits 0, 1, 2, 3, and 4 (high-order 5 bits)

8	Missing delimiter, assumed after last character
16	Excess decimal points, only first one used
32	Implied length too big, maximum used instead
64	Lost precision in floating-point fraction
128	Truncation of some significant digits from a fixed-point value

---

LOC - PROCESSOR OF ORG, START, CNOP, AND END

For most of the errors detected by LOC, the invalid operand is simply ignored.

LOC-1-5	Syntax error
LOC-2-5	Relocatability error - an operand was not absolute or relocatable where such was required.
LOC-3-5	Invalid delimiter
LOC-4-5	Operand not defined
LOC-5-5	Operand has a complex Relocatability Attribute
LOC-10-0	Name field symbol has appeared in a CNOP
LOC-11-5	Missing operand for CNOP
LOC-12-5	Invalid operand for CNOP
LOC-30-5	END or ORG operand out of range
LOC-31-5	ORG operand not in current CSECT
LOC-40-5	END operand is in a dummy control section
LOC-41-5	END operand is not halfword aligned
LOC-42-3	END operand is not type "I" (operand used anyway)

LST - PROCESSOR OF PRINT, SPACE, EJECT, AND TITLE

LST-1-5	Missing TITLE or PRINT operand
LST-2-5	Invalid delimiter, terminates processing
LST-5-4	Last TITLE delimiter missing
LST-6-5	Single ampersand appears in TITLE
LST-7-5	SPACE operand not a self-defining term
LST-9-4	Invalid PRINT option (ignored)

LTP - LITERAL SCANNER AND PROCESSOR

LTP-10-5	Invalid delimiter following a literal
LTP-20-5	No space for literal to be saved
LTP-30-5	No space for processing literal

MAC - SPECIAL MACRO INSTRUCTION PROCESSOR

MAC-10-4 Unbalanced parentheses
MAC-20-5 Too many operands
MAC-30-4 Self-defining term for PRINTOUT is too big
MAC-31-4 Symbol in PRINTOUT list is too long
MAC-32-4 "*" not last item in PRINTOUT operand list
MAC-40-5 Missing operand or operands
MAC-41-5 Invalid operand
MAC-42-5 Invalid delimiter
MAC-50-5 READFILE option not available
MAC-60-6 CONVERT is not implemented in this version of SPASM

MDE - ASSEMBLY MODE STATEMENT PROCESSOR

MDE-10-4 Inconsistency in ICTL or MALIGN specification
MDE-11-4 No room for sequence field
MDE-12-5 ICTL/ISEQ/MALIGN operand not a self-defining term
MDE-13-5 ICTL/ISEQ/MALIGN operand out of range
MDE-14-5 ICTL/ISEQ/MALIGN operand conflict
MDE-15-5 Too many operands
MDE-16-4 Sequence field longer than 8 characters (8 used)
MDE-20-5 Invalid operand in AOPTIONS statement
MDE-30-5 Invalid delimiter
MDE-40-5 Missing operand in AOPTIONS statement

MIF - MACHINE INSTRUCTION FIELD PROCESSOR

Type Checking Option

If the type checking option is in effect and a type conflict is detected, the subfield parameter (SUBFIELD=) of the error message will indicate the conflicting field's type. Subtract 32 from the subfield value if the conflicting operand uses explicit length.

SUBFIELD=	Type of Statement, Symbol or Term
0	C - Character
1	Z - Zoned decimal
2	P - Packed decimal
3	X - Hexadecimal
4	B - Binary
5	D - Long floating-point
6	E - Short floating-point
7	F - Fullword fixed-point
8	H - Halfword fixed-point
9	A - Fullword address constant
10	Y - Halfword address constant

```
    11       V   - External (virtual) address constant
    12       Q   - Q-type address constant (pseudo-register offset)
    13       S   - S-type address constant
    15       EXT - EXTRN operand or external name
    16       LOC - Location counter reference
    17       DEF - DEFER statement name
    18       I   - Machine instruction
    19       J   - Control section (CSECT)
    20       K   - Control section (DSECT)
    21       L   - Length Attribute reference
    22       M   - Macro instruction
    23       N   - Self-defining term
    24       LTO - LTORG statement


MIF-10-4     Field is too small for value, which is truncated
MIF-12-5     Length or register field is not absolute (zeroed)
MIF-20-4     Length  or  register  field  invalid (e.g., not even),
             value is truncated or rounded
MIF-21-4     Alignment error, given value is used as is
MIF-22-4     Floating-point register should be 0 or 4, set to 0
MIF-23-4     SRP instruction rounding digit must be < 10, set to 0
MIF-30-5     Addressability error (instruction zeroed)
MIF-40-3     Operand not executable (I,J,M,N,LOC,DEF,X allowed)
MIF-44-3     Operand not absolute (N allowed)
MIF-48-3     Operand not implicit aligned fixed fullword  (X,F,A,V,Q
             allowed)
MIF-52-3     Operand not implicit fixed fullword (F,A,V,LTO allowed)
MIF-56-3     Operand not implicit fixed halfword (X,H,Y,LTO allowed)
MIF-60-3     Operand not implicit long float (X,D,LTO allowed)
MIF-64-3     Operand not implicit short float (X,E,LTO allowed)
MIF-68-3     Operand not packed decimal (P,X,D,LTO allowed)
MIF-72-3     Operand not zoned decimal (C,Z,X,LTO allowed)
MIF-76-3     Operand not C,Z,P,X,B,D,E,F,H,LTO
MIF-80-3     Operand not C,Z,P,X,B,D,E,F,H,A,Y,LOC,DEF,I,J,M,LTO
MIF-84-3     Operand not C,Z,P,X,B,D,E,F,H,A,Y,V,Q,S,LOC,DEF,I,J,L,
             M,N,LTO
```

MNC - MACRO ENCODER

```
MNC-2-5      Error in macro name, no definition entered
MNC-10-1     Statement sequence not compatible with IBM syntax
MNC-12-1     Sequence symbol illegal for this operator
MNC-14-1     Name field symbol use inconsistent
MNC-16-5     Operand missing
MNC-20-0     AIF expression not Boolean
MNC-22-5     Missing sequence symbol in AIF or AGO
MNC-24-5     Unbalanced right macro quote
MNC-26-5     Wrong type of variable symbol for SETA, SETB, or SETC
MNC-40-5     Variable symbol already defined, first one used
MNC-41-1     Variable symbol starts with "SYS", definition accepted
MNC-42-5     Inconsistent definition of GLOBAL symbol
```

```
MNC-44-5     Variable symbol doesn't start with &
MNC-45-5     Array dimension not an integer less than 256
MNC-46-5     Syntax error or illegal character
MNC-48-5     Duplicate sequence symbols, new one is ignored
MNC-60-5     Attribute reference does not refer to a parameter
MNC-62-5     Illegal use of character expression
MNC-64-1     Illegal blank
MNC-66-5     Syntax error, or illegal character in expression
MNC-68-5     Stack overflow -- expression too complex
MNC-70-5     Missing parenthesis
MNC-100-5    Array used without subscript, first element used
MNC-102-5    Subscripted variable symbol not an array, SUBSCRIPT
             ignored
MNC-104-5    Undefined symbol
MNC-110-5    Missing quote in character string
MNC-120-0    Boolean operator used with arithmetic operand(s)
MNC-140-5    Name does not start with a letter
MNC-142-5    Name is too long
MNC-150-1    Machine or assembler operation modified
MNC-160-5    Missing MEND supplied for macro definition
MNC-999-9    Out of space - abort encoding of this macro
```

MOP - MACHINE INSTRUCTION OPERAND FIELD PROCESSOR

```
MOP-1-5      Syntax error, wrong or unrecognizable format
MOP-2-5      Too many operands
MOP-3-5      Missing operand or operands
```

MXP - MACRO EXPANDER

```
MXP-10-5     Calls nested too deeply, return to level zero
MXP-20-1     Macro definition has no name field parameter
MXP-30-5     Error in keyword name, or syntax error in macro call
MXP-40-5     Variable not defined as a keyword
MXP-45-5     Syntax error in operand list
MXP-50-1     More operands than explicitly specified in macro defn.
MXP-60-5     Keyword name does not start with a letter
MXP-62-5     Keyword name too long
MXP-100-5    String buffer overflow, statement terminated
MXP-102-5    Subscript error
MXP-105-5    Undefined symbol's Value or Length Attribute requested
MXP-110-5    ACTR value went to zero, expansion terminated
MXP-120-1    Length Attribute undefined for a symbol
MXP-130-1    Relocatable symbol used in macro expression, value used
MXP-132-5    String too long to convert to a value
MXP-134-5    Unable to convert string to a value
MXP-140-5    Value of symbol not known, statement terminated
MXP-150-5    Sublist index not positive, statement terminated
MXP-160-1    Zero divisor, zero used for result
MXP-200-5    Expression 1 not positive in substring operation
```

```
MXP-201-5    Expression 2 negative in substring operation
MXP-202-1    Null substring generated, expr2 is 0
MXP-203-1    Substring expr1 exceeds string length (null result)
MXP-204-1    Substring expr2 exceeds string length (tail-end used)
MXP-210-5    Undefined sequence symbol, no branch taken
MXP-220-5    Erroneous statement in definition, ACTR halved
MXP-250-1    Syntax error in specification of MNOTE severity
MXP-260-0    Decimal self-defining term expected for MNOTE severity
MXP-270-1    No continuation possible (end column = 80), text lost
MXP-300-5    Unbalanced string quotes in operand
MXP-310-5    Unbalanced left parentheses in operand
MXP-999-1    Macro expansions out of space, return to level zero.
```

OPN - NAME AND OPERATION FIELD PROCESSOR

```
OPN-10-4     Name field symbol too long (more than 8 characters)
OPN-11-4     Name field symbol illegal (ignored)
OPN-12-4     Name field symbol required, but not present
OPN-20-5     Operation not found, or unknown
OPN-21-5     Operation name too long
OPN-30-5     Missing operation field
OPN-40-5     Invalid delimiter for name or operation field
OPN-50-4     Name field symbol is multiply defined, value unchanged
OPN-70-6     Circularly defined opcode
OPN-80-6     Operation not allowed in open code
OPN-99-5     Unsupported operation code
```

OTC - OPERAND TERM COLLECTOR

```
OTC-1-0      Use  of  Length Attribute of a self-defining term is an
             incompatibility  (or  the  Length Attribute of a Length
             Attribute, etc.)
OTC-2-3      Symbol is too long, first 8 characters used
OTC-3-5      Length Attribute of a literal not defined
OTC-4-1      Reference to literal in a DSECT, ignored
```

SEX - SYSTEM EXECUTIVE PROGRAM

```
SEX-20       =GO card not preceded by an assembly
SEX-30       Execution suppressed
SEX-50       Syntax error in parameter field
SEX-51       Invalid parameter option
SEX-52       Error in specification of keyword parameter
SEX-53       Parameter conflicts with invocation parameters
SEX-90       Static  patches  worked incorrectly, assembler bugs not
             patched
SEX-91       Space available less than required minimum
```

STM - STATEMENT PROCESSOR

STM-5-3	Sequence error
STM-10-3	A comment statement is continued (column 72, or the user's continuation column is non-blank)
STM-20-4	Invalid continuation card, will try to process
STM-21-7	End-of-file encountered while scanning continuation cards (statement ignored, and an END card generated to terminate assembly)
STM-22-5	Too many continuation cards
STM-99-3	Missing END card, assembly terminated
STM-100-8	Tables overrun program area - assembly is continued but no further code is emitted, and execution will not take place
STM-999-9	Available space is exhausted, have to stop assembly but as much further syntax checking as possible will be done

---

TRM - TERM AND SYMBOL SCANNING ROUTINE

The three digits of the TRM errorcodes have the following significance: 1) the ones digit indicates the type of term, 2) the hundreds digit indicates the type of error, and 3) the tens digit provides additional information concerning the erring term or its field. Using the table below, the error message "TRM-433-0" can be determined to mean a decimal self-defining term had an excessive value.

Errorcode	Meaning
0	Type unknown
1	Binary self-defining term
2	Character self-defining term
3	Decimal self-defining term
4	Hexadecimal self-defining term
5	Symbol
6	Location counter reference (*)
7	Symbol Length Attribute reference (L')
8	Literal (=)
10	Operation code or name field being scanned
20	Operand field item being scanned
30	Self-defining term being scanned
100	Illegal character detected
200	Term is too long (too many characters)
300	Term is vacuous
400	Value of term is excessive

UDF - TEXT UNDEFERING ROUTINE

UDF-10-4 No continuation possible (end column = 80), text lost

---

UFX - USING TABLE FIXUP AND MAINTENANCE ROUTINE

UFX-1-5 Absolute expression found for base value at USING fixup time - this violates a restriction of the SPASM assembler. (Because implied addresses are commonly coded in statements like LA 1,5 it would be necessary to keep all such instructions for later fixups if there were any undefined USING statements active. Therefore, it is assumed that if the expression in a USING statement is not defined at the time the statement is scanned, then it must be a relocatable expression.)
UFX-2-5 Undefined expression found in USING statement at final fixup time
UFX-3-5 Illegal use of complex relocatability

UGH - UNSUPPORTED OPERATION CODE PROCESSOR

UGH-1-6 The operation code is not supported by SPASM

USE - USING AND DROP INSTRUCTION PROCESSOR

USE-10-5 Missing operand
USE-11-5 Too many operands
USE-20-4 Value too big (greater than X'FFFFFF')
USE-21-4 Register not defined before USING statement
USE-22-4 Register 0 not first in list
USE-23-4 Register specification error
USE-30-3 Unnecessary DROP, register not in use
USE-40-5 Illegal use of complex relocatability

VYA - ADDRESS CONSTANT SCANNER

VYA-1-5 Vacuous constant, scan terminated
VYA-5-1 Address Constant (ADCON) in DSECT will not be fixed up
VYA-10-5 Relocatable A-type constant of length less than 3 bytes - zeroed
VYA-20-5 Relocatable Y-type constant, zeroed
VYA-30-5 Addressability error in an implicit S-type constant, zeroed
VYA-40-4 Base or displacement field of an S-type constant is

	too large and is truncated to fit
VYA-41-5	Base or displacement of an S-type constant is not absolute and is zeroed
VYA-50-1	S-type address constants in literals not compatible with IBM assembler - value depends on USING statements
VYA-60-3	Q-type constants not supported, treat as A-type
VYA-70-3	V-type constants not supported, treat as A-type
VYA-80-5	Complex relocatable expressions in A-type ADCONs only

7.2 SPASM Abnormal-End Codes

CODE	INDICATION
1	Abort from KON - SYM thinks NFS just entered is undefined
2	Abort from KON - TRM found L' inside decimal SDT
3	Abort from KON - Null expression encountered
4	Abort from SYT - Null Symbol Table tree
5	Abort from GIM - Invalid space return
6	Abort from GIM - Invalid space request (too big)
7	Abort from LTP - Recursive entry to literal processor
8	Abort from VYA - Invalid constant type found
9	Abort from VYA - Invalid statement type found
1000	Abort from SCT - Pointer vector/common area too long
1001	Abort from SCT - Inadequate work space
1002	Abort from SCT - Over twenty assembly-time program interrupts in SCT
1003	Abort from SCT - Error in invocation parameters
1004	Abort from DBG - Immediate ABEND requested
1005	Abort from SCT - Open of SYSIN/SYSPRINT unsuccessful
1006	Abort from SCT - READFILE option not generated in SCT
1010	Abort from SCT - Unable to find PRB in timer trap
1011	Abort from SCT - Space not available for SYSLIB READ buffer