

Rochester Institute of Technology

RIT Digital Institutional Repository

Theses

1982

SMA80 Structured Macro Assembler

Joseph R. Garappolo

Follow this and additional works at: <https://repository.rit.edu/theses>

Recommended Citation

Garappolo, Joseph R., "SMA80 Structured Macro Assembler" (1982). Thesis. Rochester Institute of Technology. Accessed from

This Thesis is brought to you for free and open access by the RIT Libraries. For more information, please contact repository@rit.edu.

Master of Science in Computer Science

Thesis Approval Form

This is to certify that **Joseph Garappolo** has submitted a thesis entitled:

Structured Macro Assembler

to the faculty of the School of Computer Science and Technology in partial fulfillment of the requirements for the degree of Master of Science.

Approval:

(thesis advisor)

4/30/82
(date)

(committee member)

4-30-82
(date)

(committee member)

4-30-82
(date)

SMA80
STRUCTURED MACRO ASSEMBLER

BY JOSEPH R. GARAPPOLO
March 1982

PREFACE

The paper which follows presents the project undertaken for the thesis requirement in the Masters Degree program in Computer Science at the Rochester Institute of Technology in Rochester, New York.

The project involved the design and implementation of a Structured Macro Assembler. The proposal was submitted and accepted in the spring of 1980 and has since required approximately one thousand hours of effort. The undertaking of this project has given me the opportunity to apply what I have learned in the classroom and to build upon the knowledge I have obtained through my work experience.

This does not signify the end of the project, for enhancements are being made at this time to prepare the system for actual use at my place of employment.

Joseph Garappolo

April 1982

TABLE OF CONTENTS

CHAPTER 0	INTRODUCTION	
CHAPTER 1	THE 8080/8085 PROCESSOR ARCHITECTURE	
1.1	Registers	1-1
1.2	Instruction Set	1-3
CHAPTER 2	OVERVIEW OF SMA80	
2.1	Controls	2-1
2.2	Structured Statements	2-4
2.3	Conditional Statements	2-8
2.4	Macros	2-8
CHAPTER 3	IMPLEMENTATION OF SMA80	
3.1	Memory Organization	3-6
3.2	SMA80 Controls	3-7
3.3	Invoking SMA80	3-10
3.4	Source Code To Internal Representation	3-11
	3.4.1 Building and Maintaining The Line Buffer	3-11
	3.4.2 Lexical Analyzer	3-15
	3.4.3 Reserved Words	3-20
3.5	Symbol Table	3-24
	3.5.1 Types Of Identifiers	3-26
	3.5.2 Symbol Table Initialization	3-29
3.6	Arithmetic Expression Evaluation	3-32
	3.6.1 Recursive Descent Parsing	3-33
	3.6.2 Data Types	3-33
3.7	The Three Passes Over The Source Code	3-41
	3.7.1 The First Pass	3-41
	3.7.1.1 The Intermediate File	3-42
	3.7.1.2 Structured Statements	3-52
	3.7.1.3 Conditional Assembly	3-75
	3.7.1.4 Macros	3-81
	3.7.2 The Second Pass	3-89
	3.7.3 The Third Pass	3-95
	3.7.3.1 Assembly Instructions To Object Code	3-97
	3.7.3.2 Object Code File	3-99
	3.7.3.3 Expanded Listing	3-99
CHAPTER 4	FUTURE ENHANCEMENTS AND CONCLUSTION	

APPENDIX A 8085 INSTRUCTION SET

APPENDIX B SMA80 SOURCE CODE LISTINGS

SECTION 1	INC.PLM	
	inc.eld	1
	system.eld	2
	ascii.epd	7
	smatch.epd	8
	cntl.epd	9
	flags.evd	9
	fileio.epd	10
	locnt.evd	12
	express.epd	13
	object.epd	14
	print.epd	15
	getsym.epd	16
	systyp.eld	16
	symbol.epd	19
SECTION 2	MAINOV.PLM	
SECTION 3	ASINFL.PLM	
	assembly\$init	4
	file\$init\$pass1	6
	file\$init\$pass3	7
	src\$file\$reset	8
	assembly\$finish	9
SECTION 4	SEGMNT.PLM	
	segment\$init	2
	segment\$finsh	3
SECTION 5	OPNMSG.PLM	
	open\$error	3
SECTION 6	GETLIN.PLM	
	get\$line\$init	3
	get\$line	4
SECTION 7	GETSYM.PLM	
	g\$char\$init	4
	g\$char\$cnt	4
	get\$char	5
	put\$bak	6
	put\$sym\$back	6
	get\$sym	7
	get\$ascii\$string	7
SECTION 8	CONVER.PLM	
	numin	3
	numout	4
SECTION 9	SEARCH.PLM	
	search	3
	binary\$search	3
	linerr\$search	4
SECTION 10	RSBTBL.PLM	
	rsv\$word\$table	1
SECTION 11	SYMBOL.PLM	
	sym\$init	3
	sym\$entry	4
	sym\$lookup	5

	sym\$update	7
SECTION 12	EXPRES. PLM	
	additive	3
	multiplicative	6
	logical	8
	factor	10
	expression	13
SECTION 13	CNTL. PLM	
	cntl\$init	3
	cntl\$proc	4
SECTION 14	PASS1. PLM	
	pass1\$cond	4
	skip\$tag	5
	if\$proc	6
	while\$proc	8
	for\$proc	10
	jump\$table\$fill	12
	jump\$table\$copy	12
	case\$proc	13
	copdif\$scan	16
	condif\$proc	16
	ident\$proc	19
	pass1\$proc	21
	main\$pass1	24
SECTION 15	CONASM. PLM	
	evaluate\$condition	2
SECTION 16	CONDIT. PLM	
	copy\$cond	3
	get\$expression	4
	tag\$code	6
	gen\$code	7
	condition	8
	for\$code	12
	case\$code	15
SECTION 17	MACRO. PLM	
	skip\$line	4
	macro\$init	5
	param\$copy	6
	macro\$define	7
	get\$params	8
	get\$fromal\$params	8
	get\$local\$symbols	8
	get\$macro\$body	9
	macro\$expand	12
	get\$actual\$params	12
	create\$local\$symbol	13
	get\$macro\$line	15
	param\$match	15
	find\$match	15
	get\$curr\$line	17
SECTION 18	LOCTAG. PLM	
	jump\$tag\$init	3
	jump\$tag	4
SECTION 19	COPYSR. PLM	

	copy\$source\$init	3
	empth\$buffer	4
	copy\$source\$finish	5
	copy\$source	6
	copy\$char	7
SECTION 20	PASS2. PLM	
	memory\$proc	3
	ident\$proc	5
	data\$proc	8
	extrn\$proc	10
	public\$proc	11
	pass2\$proc	12
	main\$pass2	14
SECTION 21	PASS3. PLM	
	memory\$proc	3
	inst\$proc	5
	encode\$reg	5
	ident\$proc	10
	data\$proc	13
	extrn\$proc	17
	public\$proc	18
	pass3\$proc	19
	main\$pass3	21
SECTION 22	PRINT. PLM	
	page\$advance	4
	print\$finish	5
	print\$init	8
	print\$line	9
SECTION 23	OBJECT. PLM	
	new\$segment	3
	create\$segment	3
	dcl\$extrn\$sym	3
	object\$space	3
	object\$finish	4
SECTION 24	SMATCH. PLM	
	string\$match	2
SECTION 25	GASCII. PLM	
	g\$ascii	3
	pos\$i	3
SECTION 26	ASCLNG. PLM	
	g\$ascii\$lng	2
SECTION 27	GSTRNG. PLM	
	get\$string	2
SECTION 28	CLRBUF. PLM	
	clear\$buffer	2
SECTION 29	COPYBF. PLM	
	copy\$buffs	2

Bibliography

STRUCTURED MACRO ASSEMBLER

INTRODUCTION

What is a Structured Macro Assembler? A Structured Macro Assembler is a language translator that has some of the more common statements (ie. IF_THEN_ELSE, FOR, WHILE and CASE) found in structured high level programming languages, along with the versatility and speed of an assembly language.

It is true that more systems programs are being implemented in high level languages. In spite of this, there still remain systems, due to memory restrictions and speed requirements, which must be programmed in an assembly language. Assembly language provides the programmer access to the full power of the machine because of the direct relationship between the language and the machine. A Structured Macro Assembler (SMA) eases the burden of programming in assembly language and also increases the readability of the source code. Since reliable executable code is generated for the high level statements, debug time is reduced.

The Structured Macro Assembler described in this paper executes on an Intel MDS development system under the ISIS operating system and generates relocatable object code for the 8085 microprocessor. Intel's PLM/80 is the language used to implement the assembler. The reasons for using PLM/80 are:

1. A PLM/80 Compiler exists for the popular CPM Operating System .

2. PLM/80 can produce a mixed listing which contains both the PLM/80 statements and the Assembly language (ASM/80) source code generated by the compiler. This listing can be edited to produce an assembly language source code file that can be transported to other 8080, 8085 or Z-80 based computer systems.
3. PLM/80 is a block structured language which can easily be converted to other block structured languages (i.e., Pascal).

The Structured Macro Assembler (SMA80) makes three passes over the source code. The first pass is responsible for generating assembly language source code for the structured statements, evaluating conditional assembly statements, and defining and expanding macros. The intermediate code generated by the first pass is scanned by the second pass, which resolves the first level of identifier definitions. An identifier whose definition contains other identifiers cannot be resolved until these other identifier have been defined. That is, multiple passes over the source code may be necessary in order to resolve the definitions of some identifiers, if the definition is resolvable. The intermediate code is then scanned by the third and final pass which, resolves the second level of identifier definition, creates relocatable object code and generates an expanded listing of the source code.

Chapter one gives an overview of the 8080/8085 processor for which SMA80 generates object code. The remaining chapters directly

pertain to SMA80. Chapter two serves as an introduction to SMA80 by describing its features. Chapter three covers the implementation of SMA80 and, where appropriate, compares the implemented techniques to alternative methods. Chapter four discusses future work and presents the conclusion to the SMA80 project.

The table of contents contains a detailed index to Appendix B where the source code for SMA80 can be found. The reader may find this valuable when reading the report. Appendix A contains a list of the 8080/8085 instruction set.

CHAPTER 1

THE 8085 PROCESSOR ARCHITECTURE

Of the three major classifications of computers , Micro's, Mini's and Maxi's, perhaps the one that still requires the most assembly language programming is the Microcomputer. Many Microcomputer applications require speed, and impose memory restrictions which can only be satisfied with an assembly language.

One common microprocessor is Intel's 8085. The 8085 has a 16-bit address bus (addressable up to 64KB of memory), and an 8-bit bi-directional data bus, that addresses up to 256 input ports and 256 output ports.

1.1 REGISTERS

The 8085 Processor contains a finite number of registers. These registers and their functions are:

1. The Program Counter (PC). This 16-bit register is used by the processor to point to the next byte in main memory to be fetched.
2. The Stack Pointer (SP). This 16-bit register points to the next available word (16-bits/word) of the stack. The stack resides in main memory.
3. The W and Z registers. These 8-bit registers are used internal to the processor and cannot be referenced by a program.
4. The B, C, D, E, H, L and M registers. The first six of these are 8-bit general purpose registers. Pairs of these registers are combined to form 16-bit register pairs. The register pairs are BC, DE, and HL, and are referenced as B,D and H respectively. A special register, called M, is used to access the byte of data whose address is stored in register pair HL.
5. ACCUMULATOR - This 8-bit register, referenced as register A, is where all comparisons and arithmetic operations take place.
6. The Program Status Word (PSW). This 16-bit register is divided into two 8-bit sections. Bits 8-15 contain flags which reflect the state of the Accumulator. Bits 0-7 are the Accumulator.

1.2 INSTRUCTION SET

The instruction set of the 8085 includes conditional branching, decimal and binary arithmetic, logical operators, register-to-register operations, stack control, and memory transfer instructions. All comparisons and arithmetic operations are performed in the accumulator.

While the processor contains instructions for both word and byte data most of its capability is with byte data. Appendix A contains a list of the 8080/8085 instruction set.

CHAPTER 2

OVERVIEW OF SMA80

This chapter is intended to give the reader an overview of the features and capability of SMA80. While assembler controls, structured statements, conditional assembly and macros are discussed, the chapter is not intended to be a complete users manual. At the end of this chapter there are three (3) example SMA80 program listings that use many of the features described.

2.1 SMA80 CONTROLS

A number of controls exist that allow the programmer to direct the operation of SMA80. These controls may appear in the command line used to invoke SMA80 (here on in referred to as runstring) or in the source code file. Controls that appear in the source code file are preceded by a dollar sign (\$) in column one (1). The controls are TITLE, NOOBJECT, DEBUG, NOPRINT, PRINT, NOLIST, LIST, EJECT, CODE, EXPMACRO and INCLUDE, and are described below.

Unless otherwise specified the control can appear in either the runstring or the source code.

TITLE

This control is used to define a title that will be printed at the top of each page of the listing. If this control is used it must appear in the source code. The format of the TITLE control is,

TITLE (ASCII string up to 35 characters).

If the title is longer than thirty five (35) characters it is truncated at the right.

NOOBJECT

This control specifies that no object module is to be produced.

DEBUG

This control specifies that the name and address of each symbol used in the program is to be included in the object module. This information may be used by a symbolic debugger.

NOPRINT

This control is used to prevent the creation of the expanded source code listing that is normally generated by SMA80.

PRINT

This control specifies the redirection of the expanded source code listing from the default disk file, to the file/device in the control command. The form of the PRINT control is,

PRINT(new file/device).

NOLIST and LIST

The NOLIST control specifies that the listing of the source program is to be suppressed until a LIST control is encountered. The LIST control starts a listing that has been suspended by a NOLIST control.

EJECT

This control overrides the automatic form feed that occurs every sixty lines and forces the next line of the listing to start at the top of a new page. This control must appear in the source code.

CODE

This control causes code generated by the structured statements is to be included in the listing.

EXPMACRO

This control causes code generated by macro expansion is to be included in the listing.

INCLUDE

This control causes the source code to be read from the file specified in the INCLUDE command. When an END-OF-FILE is detected in the include file, control is returned to the file that invoked the INCLUDE control. INCLUDE files may be nested up to four (4) levels. The format of the INCLUDE control is,

INCLUDE(name of disk file).

This control can only appear in the source file.

2.1 Introduction To Structured Statements

SMA80's most prominent feature is its repertoire of structured statements. SMA80 supports the IF-THEN-ELSE, WHILE, FOR and CASE statements. These statements may be nested within each other and may appear any where within the source code. The formats of the statements follows.

IF-THEN-ELSE

The format of this statement is:

```
IF (condition) THEN
.
.
ENDIF

IF (condition) THEN
.
.
ELSE
.
.
ENDIF
```

As shown above, the statement may or may not use the ELSE clause. The statement must appear as shown, with the IF and THEN on the same line. The form of the condition is:

```
condition := <expression> <conditional_operator> <expression> |
            <expression> ,
```

where ,

```
conditional_operator := EQ|NE|<|GT|>|LT|<|GE|=>|LE|=< .
```

While code is generated to evaluate the condition at runtime, the value of the expression or expressions within the condition are calculated at assemble-time. The form of the expression is:

```
exp := - exp | NOT exp | exp + exp | exp - exp | exp * exp |
      exp / exp | MOD exp | exp OR exp | exp XOR exp |
      exp AND exp
```

A CONDITON with a single expression is considered true if bit 0, the right most bit, is set to one (1). Expressions that start with a number or a period are considered to be literals, That is the

expressions is used as the actual value and not as the address of the value to be compared.

EXAMPLE:

VAR1 EQU 10

IF (.VAR1 = 10) THEN

This IF statement condition uses the value of VAR1, which is 10 to make the comparison.

IF (VAR1 = 10) THEN

This IF statement condition uses the value of VAR1 as the address where the value to be compared is stored.

WHILE

The format of the WHILE statement is:

WHILE (condition) DO

.

.

ENDWHILE .

The condition is the same format as in the IF condition.

FOR

The format of this statement is:

```
FOR <exp_1> = <exp_2> TO <exp_3> DO
.
.
ENDFOR
or
FOR <exp_1> = <exp_2> TO <exp_3> BY <exp_4>
.
.
ENDFOR .
```

Where,

1. "exp_1" is the address of the value to be incremented,
2. "exp_2" is the value or the address of the value that exp_1 is initialized to,
3. "exp_3" is the value or the address of the value that exp_1 is incremented to, and
4. "exp_4" is the optional parameter that specifies the value or the address of the value that exp_1 is incremented by.

If exp_4 is not specified exp_1 is incremented by one (1).

CASE

The format of this statement is,

```
CASE <expression> DO
DO
.
.
ENDDO
DO
.
.
ENDDO
ENDCASE

                                or

CASE <expression> TO <expression> DO
DO
.
.
ENDDO
ENDCASE .
```

The DO and ENDDO symbols start and terminate the individual cases. A case statement may contain up to thirty (30) cases. The "TO" in the second format imposes a limit to the maximum case in a case statement. If the first expression, which contains the number of the case to be executed, has a value greater than the "TO" limit, no attempt is made to execute that case. The "TO" clause prevents the attempted execution of nonexistent cases and gives the programmer the option of whether the extra code necessary to make this check should be generated. The first case of a CASE statement is case zero (0).

2.3 Conditional Assembly

Conditional assembly allows the programmer to select what code is to be include in the assembly process. The format of the conditional assembly statement is,

```
CONDIF (condition) THEN
.
.
ENDCOND
                                or
CONDIF (condition) THEN
.
.
ELSECOND
.
.
ENDCOND .
```

The condition is of the same format as that used by the IF and WHILE statements, except the condition is evaluated at assembly-time and the value of the expressions within the condition is assumed to be a absolute.

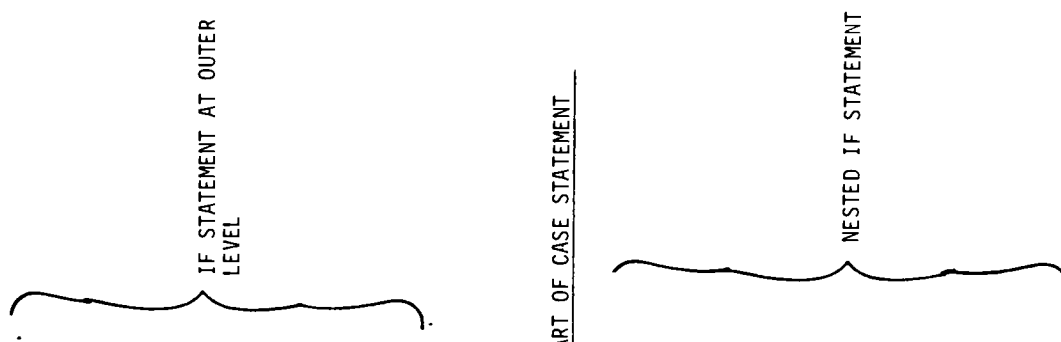
2.4 Macro Processor

SMA80's macro processor is a line oriented single pass macro processor. The one disadvantage of a single pass macro processor is that foward calls to macros are not allowed. A macro must be defined before it can be invoked. The advantage to single pass macro processor is that macros may be redefined. A macro may even redefine itself so that subsequent invocations of the same macro will expand into different code. Figure 2.3 shows some of the features of

SMA80's macro processor that are listed below.

1. Macro calls may be nested up to ten (10) levels.
2. Macros may be redefined.
3. Local symbols are supported.
4. Formal parameters can be any legal identifier. A legal identifier must start with an alphabetic or '@' and can be followed by '@'s, letters or digits.
5. Actual parameters are either legal identifiers or are enclosed by quotes. (Two adjacent quotes specify that the quote is text and not a terminating quote).
6. Comments starting with a ';' are not included in the macro body definition and do not appear during macro expansion. This saves space in the macro definition table and eliminates duplicate comments through the program. Comments not starting with the semicolon that follow assembler instructions are included in macro expansion.
7. If a macro call is nested all identifiers not matched by the macro's formal parameter list are compared to the formal parameter lists of the outer macros, in descending order. This allows general macros invoked from within other macros to share the parameters of the outer macros. This is useful when the parameter list, which is limited to one line, is too small to pass all the necessary parameters.
8. During macro expansion the macro body is fed back into the first pass. This allows macros to contain other macro definitions or any other SMA80 statements, including conditional assembly statements.

LOC	CODE DATA	LINE	SOURCE CODE
		22	\$EJECT _____ PAGE EJECT CONTROL
000A		23	\$INCLUDE(:F1:TST1.ELD) _____ INCLUDE FILE CONTROL AT FIRST LEVEL
0020		24	AAA EQU 10
		25	BBB EQU 20H
		26	;LEVEL #1
		27	\$INCLUDE(:F1:TST2.ELD) _____ INCLUDE FILE CONTROL AT SECOND LEVEL
		28	;LEVEL #2
001B 3E	0A	29	IF (.AAA EQ .BBB) THEN
001A 32	5C00 D	30	MVI A,AAA
		31	STA JOE
		32	IF (JOE EQ JIM) THEN
		33	WHILE (JOE LE .AAA) DO
		34	CASE JOE TO 10 DO
		35	DO
		36	IF (JOE EQ .BBB) THEN
		37	STA JIM
		38	ENDIF
		39	ENDDO
		40	DO
		41	STA BBB
		42	ENDDO
		43	ENDCASE
		44	ENDWHILE
		45	ENDIF
0050 32	7000 D	46	ENDIF
		47	\$INCLUDE(:F1:TST3.ELD)
		48	; THIS IS A COMMENT TEST AT LEVEL #3
		49	\$INCLUDE(:F1:TST4.ELD)
		50	;LEVEL #4
		51	ASEG
		52	AA: CASE JOE TO 10 DO
		53	DO
		54	IF (.JJJ EQ 10H) THEN
		55	IF (JOE GE JIM) THEN
		56	WHILE (JOE GT JIM) DO
		57	LXI H, JOE
		58	STA JIM
		59	FOR JOE = 1 TO JIM BY 10 DO
		60	LDA JIM
		61	CASE JOE DO
		62	DO
		63	LDA JOE
		64	ENDDO
		65	ENDCASE
		66	ENDFOR
		67	ENDWHILE
		68	LXI H, JIM
		69	LDA JOE
		70	CMP M
		71	ENDIF
		72	ENDIF
		73	ENDDO
		74	DO
		75	END OF FIRST CASE
		76	START OF SECOND CASE
		77	DO
		78	END OF FIRST CASE
		79	START OF SECOND CASE
		80	DO
		81	END OF FIRST CASE
		82	START OF SECOND CASE
		83	DO
		84	END OF FIRST CASE
		85	START OF SECOND CASE
		86	DO
		87	END OF FIRST CASE
		88	START OF SECOND CASE
		89	DO
		90	END OF FIRST CASE
		91	START OF SECOND CASE
		92	DO
		93	END OF FIRST CASE
		94	START OF SECOND CASE
		95	DO
		96	END OF FIRST CASE
		97	START OF SECOND CASE
		98	DO
		99	END OF FIRST CASE
		100	START OF SECOND CASE
		101	DO
		102	END OF FIRST CASE
		103	START OF SECOND CASE
		104	DO
		105	END OF FIRST CASE
		106	START OF SECOND CASE
		107	DO
		108	END OF FIRST CASE
		109	START OF SECOND CASE
		110	DO
		111	END OF FIRST CASE
		112	START OF SECOND CASE
		113	DO
		114	END OF FIRST CASE
		115	START OF SECOND CASE
		116	DO
		117	END OF FIRST CASE
		118	START OF SECOND CASE
		119	DO
		120	END OF FIRST CASE
		121	START OF SECOND CASE
		122	DO
		123	END OF FIRST CASE
		124	START OF SECOND CASE
		125	DO
		126	END OF FIRST CASE
		127	START OF SECOND CASE
		128	DO
		129	END OF FIRST CASE
		130	START OF SECOND CASE
		131	DO
		132	END OF FIRST CASE
		133	START OF SECOND CASE
		134	DO
		135	END OF FIRST CASE
		136	START OF SECOND CASE
		137	DO
		138	END OF FIRST CASE
		139	START OF SECOND CASE
		140	DO
		141	END OF FIRST CASE
		142	START OF SECOND CASE
		143	DO
		144	END OF FIRST CASE
		145	START OF SECOND CASE
		146	DO
		147	END OF FIRST CASE
		148	START OF SECOND CASE
		149	DO
		150	END OF FIRST CASE
		151	START OF SECOND CASE
		152	DO
		153	END OF FIRST CASE
		154	START OF SECOND CASE
		155	DO
		156	END OF FIRST CASE
		157	START OF SECOND CASE
		158	DO
		159	END OF FIRST CASE
		160	START OF SECOND CASE
		161	DO
		162	END OF FIRST CASE
		163	START OF SECOND CASE
		164	DO
		165	END OF FIRST CASE
		166	START OF SECOND CASE
		167	DO
		168	END OF FIRST CASE
		169	START OF SECOND CASE
		170	DO
		171	END OF FIRST CASE
		172	START OF SECOND CASE
		173	DO
		174	END OF FIRST CASE
		175	START OF SECOND CASE
		176	DO
		177	END OF FIRST CASE
		178	START OF SECOND CASE
		179	DO
		180	END OF FIRST CASE
		181	START OF SECOND CASE
		182	DO



START OF CASE STATEMENT

START OF FIRST CASE

NESTED IF STATEMENT

NESTED WHILE STATEMENT

END OF FIRST CASE
START OF SECOND CASE

Figure 2.1 cont.

LOC	CODE	DATA	LINE	SOURCE CODE
0085	32	7000	D	184 STA JIM
				186 ENDDO
				188 DO
008B	32	5C00	D	189 STA JOE } THIRD CASE
				191 ENDDO
				192 ENDCASE
				198
				199 DSEG
000B	21	5C00	D	200 IF (JOE EQ .JJJ) THEN
				204 LXI H, JOE
				206 ELSE
000E	21	7000	D	208 LXI H, JIM
				209 ENDIF
				211
001B	32	5C00	D	212 WHILE (JOE EQ JIM) DO
				218 STA JOE
				220 ENDWHILE
				222
003E	21	5C00	D	223 FOR JOE = 1 TO JIM BY 2 DO
0041	32	7000	D	238 LXI H, JOE
				239 STA JIM
				241 ENDFOR
				243
0047	C3	0000		244 JMP AA
004A	C3	1600		245 JMP BB
004D	C3	2700		246 JMP DD
0050	C3	3A00		247 JMP EE
0053	C3	5700		248 JMP FF
0056	C3	5A00		249 JMP GG
0059	C3	1D00		250 JMP HH
				251
005C				252 JOE: DS 20
0070				253 JIM: DS 20
				254 END

END OF CASE STATEMENT

IF STATEMENT

WHILE STATEMENT

FOR STATEMENT

END OF SECOND CASE

LINES 24 - 198 ARE FROM THE INCLUDE FILES
TST1.ELD, TST2.ELD, TST3.ELD and TST4.ELD.

MODULE INFORMATION

CODE AREA SIZE = 63H 99D
 DATA AREA SIZE = 84H 132D
 ABSOLUTE AREA SIZE = 97H 151D
 0 PROGRAM ERROR(S)

END OF SMABO

Figure 2.2

LOC	CODE DATA	LINE	SOURCE CODE
-----	-----------	------	-------------

IS-II SMA-80 ASSEMBLER

A-80 INVOKED BY:

SMABO : F1: TEST ASM CODE

000A		1	\$TITLE(TEST PROGRAM #1)
		2	\$PRINT(:LP:)
		3	VAR1 EQU 10
0000 21	0A00	4	CONDIF (VAR1 = 10) THEN
		5	LXI H,VAR1
		6	WHILE (VAR1 = 10) DO
		7	- @00001:
0003 3A	0A00	8	LDA VAR1
0006 FE	0A	9	CPI 10
0008 C2	1100	10	JNZ @00000
000B 21	0A00	11	LXI H,VAR1
000E C3	0300	12	JMP @00001
		13	ENDWHILE
		14	- @00000:
		15	ELSECOND
		16	IF (VAR1 = 11) THEN
		17	LXI H,VAR1
		18	ENDIF
		19	ENDCOND
		20	CSEG
0010		21	JJJ EQU 10H

THIS EXAMPLE IS THE SAME AS FIGURE 2.1 EXCEPT THAT THE CONTROL CODE WAS ADDED TO THE RUNSTRING CAUSING THE CODE GENERATED BY THE STRUCTURED STATEMENTS TO BE INCLUDED IN THE LISTING.

ALL LINES WITH A '-' TO THE RIGHT OF THE LINE NUMBER WERE GENERATED BY SMABO.

LDC	CODE	DATA	LINE	SOURCE CODE
			22	\$EJECT
			23	\$INCLUDE(:F1:TST1.ELD)
000A			24	AAA EQU 10
0020			25	BBB EQU 20H
			26	; LEVEL #1
			27	\$INCLUDE(:F1:TST2.ELD)
			28	; LEVEL #2
			29	IF (.AAA EQ .BBB) THEN
0011 3E	0A		30 -	MVI A,AAA
0013 FE	20		31 -	CPI BBB
0015 C2	6300	C	32 -	JNZ @00002
0018 3E	0A		33	MVI A,AAA
001A 32	5C00	D	34	STA JOE
			35	IF (JOE EQ JIM) THEN
001D 3A	5C00	D	36 -	LDA JOE
0020 21	7000	D	37 -	LXI H,JIM
0023 BE			38 -	CMP M
0024 C2	6300	C	39 -	JNZ @00003
			40	WHILE (JOE LE .AAA) DO
			41 -	@00005:
0027 3A	5C00	D	42 -	LDA JOE
002A FE	0A		43 -	CPI AAA
002C CA	3200	C	44 -	JZ \$+3
002F D2	6300	C	45 -	JNC @00004
			46	CASE JOE TO 10 DO
0032 2A	5C00	D	47 -	LHLD JOE
0035 5D			48 -	MOV E,L
0036 0E	00		49 -	MVI D,0
003B 3E	0A		50 -	MVI A,10
003A BD			51 -	CMP L
003B DA	6000	C	52 -	JC @00007
003E 21	5C00	C	53 -	LXI H,@00006
0041 19			54 -	DAD D
0042 19			55 -	DAD D
0043 5E			56 -	MOV E,M
0044 23			57 -	INX H
0045 4E			58 -	MOV D,M
0046 EB			59 -	XCHG
0047 E9			60 -	PCHL
			61 -	@00008:
			62	DO
			63	IF (JOE EQ .BBB) THEN
0048 3A	5C00	D	64 -	LDA JOE
004B FE	20		65 -	CPI BBB
004D C2	5300	C	66 -	JNZ @00009
0050 32	7000	D	67	STA JIM
			68	ENDIF
			69 -	@00009:
0053 C3	6000	C	70 -	JMP @00007
			71	ENDDO
			72 -	@00010:
			73	DO
0056 32	2000		74	STA BBB

LDC	CODE DATA	LINE	SOURCE CODE
		76	ENDDO
		77	ENDCASE
005C	4800 C	78 -	@00006:
005E	5600 C	79 -	DW @00008
		80 -	DW @00010
		81 -	@00007:
0060	C3 2700 C	82 -	JMP @00005
		83	ENDWHILE
		84 -	@00004:
		85	ENDIF
		86 -	@00003:
		87	ENDIF
		88 -	@00002:
		89	\$INCLUDE(:F1:TST3.ELD)
		90	; THIS IS A COMMENT TEST AT LEVEL #3
		91	\$INCLUDE(:F1:TST4.ELD)
		92	; LEVEL #4
		93	ASEG
		94	AA: CASE JOE TO 10 DO
0000	2A 5C00 D	95 -	LHLD JOE
0003	5D	96 -	MOV E,L
0004	0E 00	97 -	MVI D,0
0006	3E 0A	98 -	MVI A,10
0008	8D	99 -	CMP L
0009	DA 9700	100 -	JC @00012
000C	21 9100	101 -	LXI H,@00011
000F	19	102 -	DAD D
0010	19	103 -	DAD D
0011	5E	104 -	MOV E,M
0012	23	105 -	INX H
0013	4E	106 -	MOV D,M
0014	EB	107 -	XCHG
0015	E9	108 -	PCHL
		109 -	@00013:
		110	DO
		111	BB: IF (.JJJ EG 10H) THEN
0016	3E 10	112 -	MVI A,JJJ
0018	FE 10	113 -	CPI 10H
001A	C2 7F00	114 -	JNZ @00014
		115	HH: IF (JOE GE JIM) THEN
001D	3A 5C00 D	116 -	LDA JOE
0020	21 7000 D	117 -	LXI H,JIM
0023	BE	118 -	CMP M
0024	DA 7F00	119 -	JC @00015
		120	DD: WHILE (JOE GT JIM) DO
		121 -	@00017:
0027	3A 5C00 D	122 -	LDA JOE
002A	21 7000 D	123 -	LXI H,JIM
002D	BE	124 -	CMP M
002E	DA 7800	125 -	JC @00016
0031	CA 7800	126 -	JZ @00016
0034	21 5C00 D	127	LXI H, JOE
0037	32 7000 D	128	STA JIM

LOC	CODE	DATA	LINE	SOURCE CODE
003A 21	5C00	D	130 -	LXI H, JOE
003D 36	01		131 -	MVI M, 1
			132 -	@00018:
003F 3A	7000	D	133 -	LDA JIM
0042 21	5C00	D	134 -	LXI H, JOE
0045 BE			135 -	CMP M
0046 DA	7500		136 -	JC @00019
0049 C3	5700		137 -	JMP @00020
			138 -	@00021:
004C 3A	5C00	D	139 -	LDA JOE
004F C6	0A		140 -	ADI 10
0051 D2	3F00		141 -	JNC @00018
0054 C3	7500		142 -	JMP @00019
			143 -	@00020:
0057 3A	7000	D	144 -	LDA JIM
			145 -	CASE JOE DO
005A 2A	5C00	D	146 -	LHLD JOE
005D 5D			147 -	MOV E, L
005E 0E	00		148 -	MVI D, 0
0060 21	7000		149 -	LXI H, @00022
0063 19			150 -	DAD D
0064 19			151 -	DAD D
0065 5E			152 -	MOV E, M
0066 23			153 -	INX H
0067 4E			154 -	MOV D, M
0068 EB			155 -	XCHG
0069 E9			156 -	PCHL
			157 -	@00024:
006A 3A	5C00	D	158	DO
006D C3	7200		159	LDA JOE
			160 -	JMP @00023
			161	ENDDO
			162	ENDCASE
0070	6A00		163 -	@00022:
			164 -	DW @00024
			165 -	@00023:
0072 C3	4C00		166 -	JMP @00021
			167	ENDFOR
0075 C3	2700		168 -	@00019:
			169 -	JMP @00017
			170	ENDWHILE
			171 -	@00016:
0078 21	7000	D	172	LXI H, JIM
007B 3A	5C00	D	173	LDA JOE
007E BE			174	CMP M
			175	ENDIF
			176 -	@00015:
			177	ENDIF
			178 -	@00014:
007F C3	9700		179 -	JMP @00012
			180	ENDDO
			181 -	@00025:
			182	DO

LDC	CODE	DATA	LINE	SOURCE CODE
0085 32	D	7000	184	STA JIM
0088 C3		9700	185 -	JMP @00012
			186	ENDDO
			187 -	@00026: DO
			188	DO
008B 32	D	5C00	189	STA JOE
008E C3		9700	190 -	JMP @00012
			191	ENDDO
			192	ENDCASE
			193 -	@00011: DO
0091		1600	194 -	DW @00013
0093		8200	195 -	DW @00025
0095		8B00	196 -	DW @00026
			197 -	@00012: DO
			198	DO
			199	DSEG
			200	IF (JOE EQ .JJJ) THEN
0000 3A	D	5C00	201 -	LDA JOE
0003 FE	10		202 -	CPI JJJ
0005 C2	0E00		203 -	JNZ @00027
0008 21	5C00		204	LXI H, JOE
000B C3	1100		205 -	JMP @00028
			206	ELSE
			207 -	@00027: DO
000E 21	D	7000	208	LXI H, JIM
			209	ENDIF
			210 -	@00028: DO
			211	DO
			212	WHILE (JOE EQ JIM) DO
			213 -	@00030: DO
0011 3A	D	5C00	214 -	LDA JOE
0014 21	D	7000	215 -	LXI H, JIM
0017 BE			216 -	CMP M
001B C2	D	2100	217 -	JNZ @00029
001B 32	D	5C00	218	STA JOE
001E C3	D	1100	219 -	JMP @00030
			220	ENDWHILE
			221 -	@00029: DO
			222	DO
			223	FOR JOE = 1 TO JIM BY 2 DO
0021 21	D	5C00	224 -	LXI H, JOE
0024 36	01		225 -	MVI M, 1
			226 -	@00031: DO
0026 3A	D	7000	227 -	LDA JIM
0029 21	D	5C00	228 -	LXI H, JOE
002C BE			229 -	CMP M
002D DA	D	4700	230 -	JC @00032
0030 C3	D	3E00	231 -	JMP @00033
			232 -	@00034: DO
0033 3A	D	5C00	233 -	LDA JOE
0036 C6	02		234 -	ADI 2
003B D2	D	2600	235 -	JNC @00031
003B C3	D	4700	236 -	JMP @00032
			237	DO

Figure 2.2 cont.

LOC	CODE	DATA	LINE	SOURCE CODE
003E 21	5C00	D	238	LXI H, JOE
0041 32	7000	D	239	STA JIM
0044 C3	3300	D	240 -	JMP @00034
			241	ENDFOR
			242 -	@00032:
			243	
0047 C3	0000		244	JMP AA
004A C3	1600		245	JMP BB
004D C3	2700		246	JMP DD
0050 C3	3A00		247	JMP EE
0053 C3	5700		248	JMP FF
0056 C3	5A00		249	JMP GG
0059 C3	1D00		250	JMP HH
			251	
005C			252	JOE: DS 20
0070			253	JIM: DS 20
			254	END

MODULE INFORMATION

CODE AREA SIZE	=	63H	99D
DATA AREA SIZE	=	84H	132D
ABSOLUTE AREA SIZE	=	97H	151D
0 PROGRAM ERROR(S)			

END OF SMABO

Figure 2.3

LOC CODE DATA LINE SOURCE CODE

IS-II SMA-80 ASSEMBLER
IA-80 INVOKED BY: SMABO : F1:MAC1.SMA EXPMACRO NOOBJECT PRINT(:LP:)

1 \$TITLE (MACRO EXAMPLE #1)

0000

2 SETO SET 0

4 MAC1 MACRO PARAM1

5 LOCAL LOC1

6 MAC1 MACRO
7 RET ,TEST COMMENT #1
8
9 ENDMACRO

10 SAVEREG 'REGISTER FOR PARAM1'

11 DS 1

12 CALL PARAM1 TEST COMMENT #2

13 ENDMACRO

14 SAVEREG MACRO PARAM1

15 PUSH PSW PARAM1

16 PUSH H

17 PUSH B

18 PUSH D

19 STA LOC1

20 ENDMACRO

21 MAC1 SUBRTN

22 MACRO

23 RET

24 ENDMACRO

25 SAVEREG 'REGISTER FOR SUBRTN'

26 PUSH PSW REGISTER FOR SUBRTN

27 PUSH H

28 PUSH B

29 PUSH D

30 STA @00000

31 DS 1

32 CALL SUBRTN TEST COMMENT #2

33 SUBRTN:

34 LDA 10

35 MOV M,A

36 MAC1

37 RET

38 END

0000 F5

0001 E5

0002 C5

0003 D5

0004 32

0007 C 0700

0008 CD 0B00

000B 3A 0A00

000E 77

000F C9

} nested macro
definition

Outer macro definition that has
one parameter and one local symbol

} macro definition

} macro expansion.

The name of the subroutine SUBRTN has been
passed as a parameter.

} The second expansion of the macro MAC1 results in a different
expansion

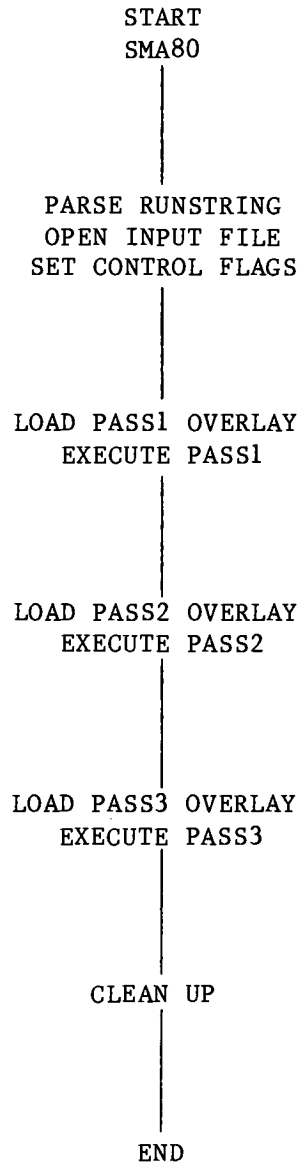
CHAPTER #3

IMPLEMENTATION OF SMA80

The process of converting the SMA80 source code to object code follows the basic steps displayed in figure 3.1.1. The functions performed by each of the steps involves the interaction and coordination of many intermediate or sub-processes (figures 3.1.2-4). Each of the sub-processes performs a unique task making the job of incorporating additional features and changes easy.

This chapter describes the techniques used to implement each of the above mentioned sub-processes. Where appropriate pseudo code will be included to aid in the presentation of ideas and techniques. To avoid undue complexity, the pseudo code does not show details which are not important in understanding the logic of a routine.

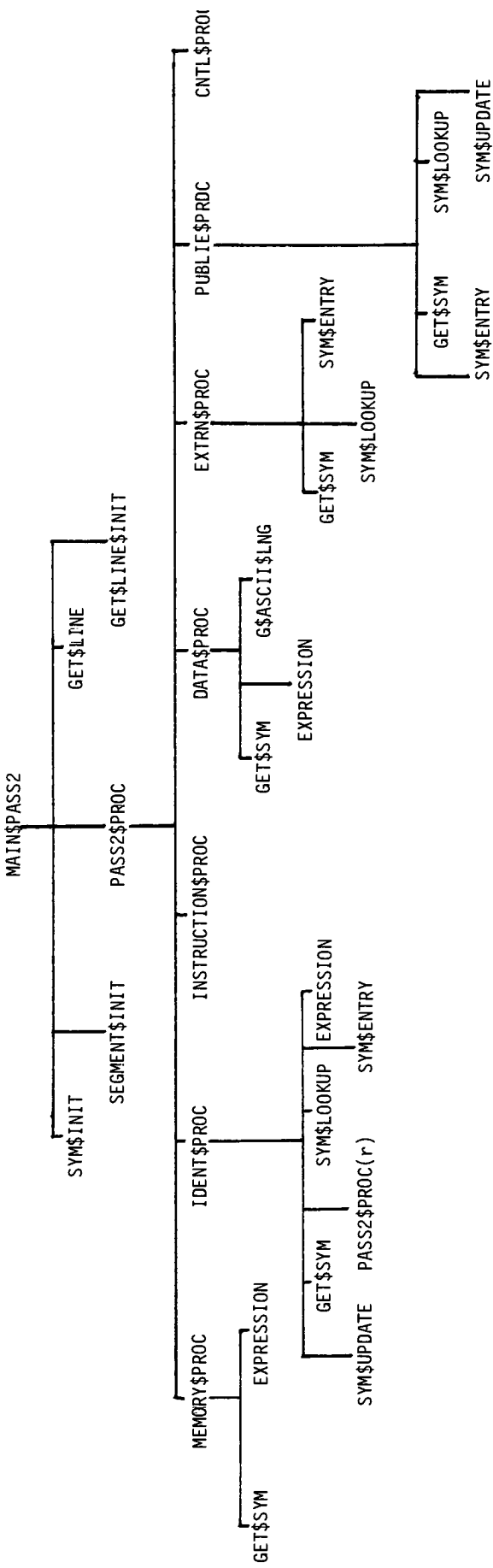
SMA80 OVERVIEW
Figure 3.1.1



[illegible]

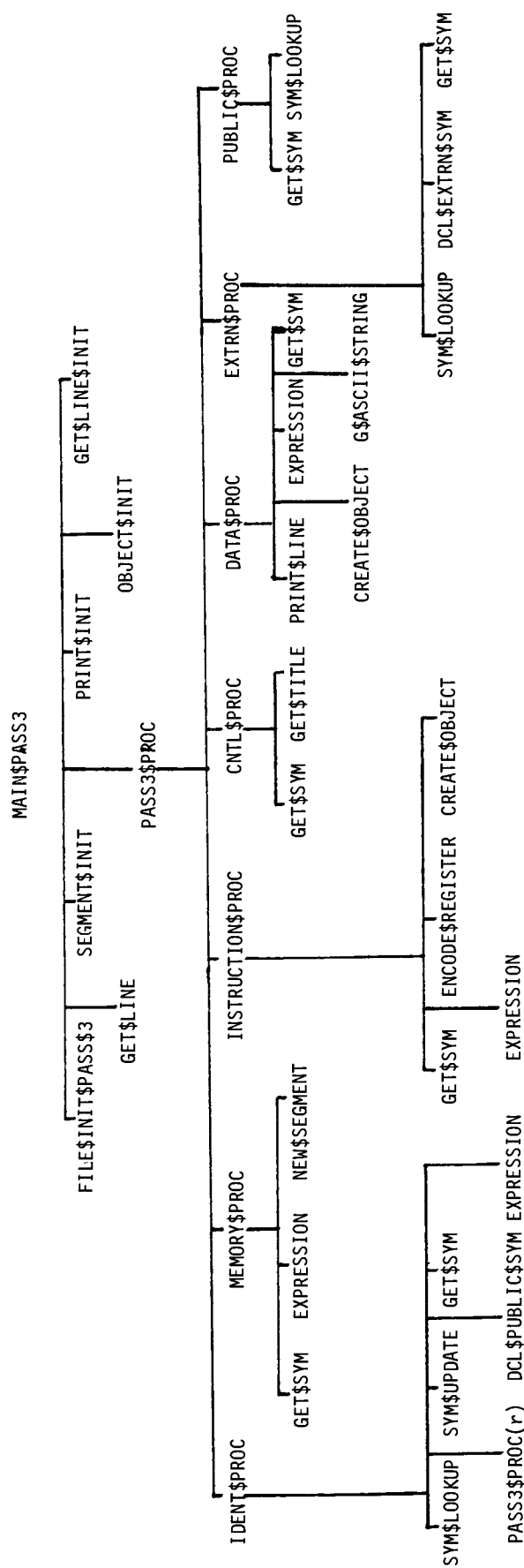
PASS1\$COND(r) - recursive call.

PASS2 SUB-PROCESSES
Figure 3.1.3



PASS2\$PROC(r) - recursive call

PASS3 SUB PROCESSES
Figure 3.1.4



PASS3\$COND(r) - recursive call

3.1 MEMORY ORGANIZATION

In general as the main memory available for use by the symbol table, I/O file buffers and the stack increases, so does the efficiency and capability of SMA80. A larger symbol table area allows more identifiers to be used in a program, larger I/O file buffers decrease the number of disk accesses, and a larger stack increases the depth to which structured statements can be nested. To increase the available buffer space SMA80 uses three disk overlays, one for each of the three passes made over the source code. Each overlay contains the modules solely associated with the individual pass. The global modules used through out the system are in memory at all times. Table 3.1.1 shows the modules that comprise the three overlays.

MODULE ORGANIZATION
TABLE 3.1.1

global modules	- mainov.obj, asinfi.obj, getsym.obj, search.obj, rsvtbl.obj, conver.obj, symbol.obj, express.obj, getlin.obj, gstrng.obj, asclng.obj, segmnt.obj, cntl.obj, opnmsg.obj, clrbuf.obj, copybf.obj, gascii.obj and smatch.obj.
PASS1 overlay	- pass1.obj, condit.obj, conasm.obj, macro.obj, loctag.obj, and copysr.obj.
PASS2 overlay	- pass2.obj
PASS3 overlay	- pass3.obj, print.obj and object.obj.

3.2 SMA80 CONTROLS

The operation of SMA80 may be directed by a number of controls which specify options such as listing code generated by the structured statements and destination of the listing.

Each of the three passes of SMA80 (section 3.3.7) is responsible for parsing and taking action on particular controls that appear in the source code. The first pass looks for the NOOBJECT, CODE, EXPMACRO, DEBUG, NOPRINT, PRINT and INCLUDE controls. All but the INCLUDE control affect the object code and listing output of SMA80 and must be issued before the initialization of the third pass which takes certain steps depending on the setting of these controls. The second pass only looks for the TITLE control which is also needed before the initialization of the third pass. Finally the third pass checks for the TITLE, NOLIST, LIST and EJECT controls.

Puesdo Code For CNTL\$PROC
Figure 3.2.1

CNTL_PROC:

```
COMMON$CNTL_1:
  IF (NOLIST control) then
    set list_flag off;
  else IF (LIST control) then
    set list_flag on;
  else return control undefined (U);
end COMMON_CNTL_1;

COMMON_CNTL_2:
  IF (NOOBJECT control) then
    set object_flag off;
  else IF (CODE control) then
    set statement_code_flag on;
  else IF (EXPMACRO control) then
    set expand_macor_flag on;
  else IF (DEBUG control) then
    set debug_flag on;
  else IF (NOPRINT control) then
    set print_flag off;
  else IF (PRINT control) then
    do;
      default_print_flag = off;
      get new list device/file
        into PRINT_FILE buffer;
    end;
  else return control undefined;
end COMMON_CNTL_2;

IF (first pass) then
  IF (INCLUDE control) then
    do;
      increment source file LEVEL;
      get include file name and open the file;
    end;
  else call COMMON_CNTL_2;
else IF (second pass) then
  IF (TITLE control) then
    /* we do not want do define title twice */
    IF (title_flag off) then
      do;
        set title_flag on;
        get title into title buffer;
      end;
    else IF (third pass) then
      IF (EJECT control) then
        /* the print routine sees this flag and
          advances the page and set print_flag
```

```

        off.
    */
    set page_flag on;
else IF (TITLE control) then
    /* title_flag_2 indicates the title
       has already been defined
    */
    IF (title_flag_2 is off) then
        set title_flag_2 on;
        /* can not define title twice */
    else set error to illegal control;
    else call COMMON_CNTL_1;
    return error if any have occurred;
end CNTL_PROC;

```

3.3 INVOKING SMA80

Before the assembly process can begin, the name of the SMA80 source code file is required. This file name, along with the optional controls, introduced in the previous section, are provided by the runstring that invokes SMA80. The format of the runstring is,

SMA80 file_name <optional controls>.

Once SMA80 begins executing control is transferred to the routine ASSEMBLY_INIT which scans the runstring for the source code file name and opens the file for read access. The remainder of the line is parsed for optional controls. If any controls are encountered CNTL_PROC is called (section 3.2). If the runstring contains any errors, a descriptive error message is displayed on the console and SMA80 is aborted.

3.4 SOURCE CODE TO INTERNAL REPRESENTATION

One of the many processes which SMA80 is responsible for is converting the source code developed by a programmer into a form efficiently usable internal to SMA80. To accomplish this the SMA80 source code is read from a disk into an input buffer in main memory. Segments of the input buffer, which constitute lines, are read into an area of memory called LINE BUFFER. The ASCII symbols in LINE BUFFER are scanned and converted into a form used internal to SMA80.

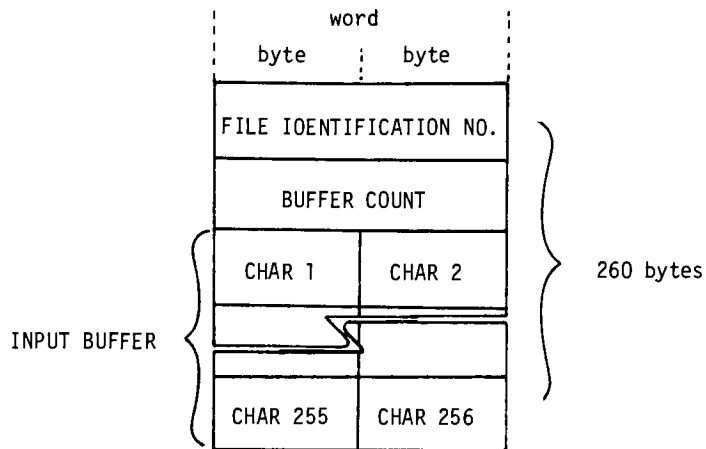
3.4.1 Building and Maintaining The Line Buffer

LINE BUFFER is filled with lines of source code (each up to 128 bytes long). The lines are terminated by a line feed. The routine which fills the LINE BUFFER and manages the input file(s) is called GET_LINE. GET_LINE is initialized at the beginning of each of the three passes over the source code made by SMA80. For the first pass GET_LINE is initialized to handle up to five (5) input files. Each of the INPUT FILE CONTROL BLOCK's (IFCB) (figure 3.4.1), contains information about the input file and has a input buffer of 256 bytes (2 disk sectors). The first IFCB is reserved for the main SMA80 source code file. The remaining four IFCB's are used for source files invoked by INCLUDE statements (see section 3.2) (thus INCLUDE files may be nested four LEVELS deep).

The second and third passes of SMA80 use as input the intermediate file generated by the first pass, making it unnecessary for GET_LINE to handle more than one input file. To increase the efficiency of building a line, the input buffer for the intermediate file is made larger. This larger buffer uses the space previously used by the four additional IFCB's, making the usable buffer space 1280 bytes or 10 disk sectors.

Every time GET_LINE is called a new line is read from the input buffer at the current file level (0-4) into the LINE BUFFER. If the input buffer becomes empty, GET_LINE accesses the disk file and fills it. When an END-OF-FILE (EOF) is reached at any level other than level zero (0), the level is decremented and the LINE BUFFER is filled from the input buffer at the new level. When the EOF is reached in the main source file (level 0), a flag is set indicating that no more source code is available.

INPUT FILE CONTROL BLOCK
figure 3.4.1

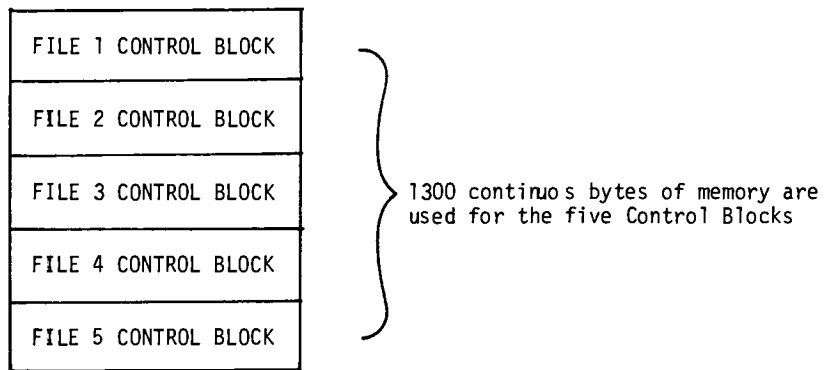


FILE IDENTIFICATION NO. - Used by the ISIS Operation system. ..

BUFFER COUNT Next byte in the input buffer to be transferred to LINE BUFFER

INPUT BUFFER 256 byte buffer which is filled from a disk file

MEMORY ORGANIZATION OF CONTROL BLOCKS



Pseudo Code for GET_LINE_INIT and GET_LINE
Figure 3.4.2

GET_LINE_INIT:

```
    IF first pass then
        set the 5 IFCB's buffer count = 256;
    else
        set the first IFCB buffer count = '280;
end GET_LINE_INIT;
```

GET_LINE:

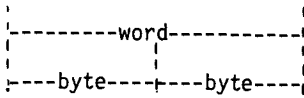
```
    while (not(over flow LINE BUFFER) and not(EOF) and
           not(EOL)) do
        if (input buffer at current level is empty) then
            fill buffer;
        fill LINE BUFFER with next character from input buffer
        at current level;
        if (character is end_file_indicator)) then
            if (level at main input file) then
                EOF;
            else do;
                close file at this level;
                decrement level;
            end;
        else if (character is end of line) then
            EOL
        end while;
end GET_LINE;
```


3.4.2 Lexical Analyzer

The process by which the contents of the LINE BUFFER are converted into the form that is used within SMA80 is called LEXICAL ANALYSIS or SCANNING. The output of the Lexical Analyzer is a stream of TOKENS, which are numbers that represent the symbols in the source program. GET_SYM, the Lexical Analyzer routine, may provide additional information besides the token (ex., the object code length for a particular instruction). All the information returned by GET_SYM is placed into a buffer called SYMBOL_BUFFER. The value of the first byte of SYMBOL_BUFFER, called SYM_TYPE, determines which one of nine classes of data structures (figure 3.4.3) is to be used when accessing the information returned by GET_SYM.

Although the process of Lexical Analysis is very simple, it is by far the most frequently called routine. The state diagram (figure 3.4.4) of the Lexical Analyzer, shows that before an ASCII string that looks like an identifier (i.e. starts with a letter and is followed by letters, digits or @'s), can be determined to be one, it must first be determined that the string is not a RESERVED WORD. Since much, if not most of the time spent by GET_SYM is searching for reserved words, the efficiency of the method used for this search is crucial.

SYMBOL CLASSES
Figure 3.4.3

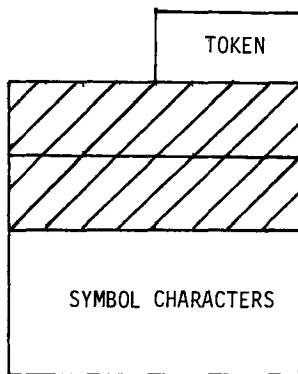


CLASS 1



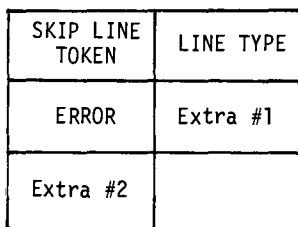
TOKEN's - DELIMITERS

CLASS 2



TOKEN's - IDENTIFIER, DO, END, AND, EQU, SET, XOR, ELSE, THEN, ENDCASE, ENDDO, ENDFOR, ENDWHILE, BY, OR, NOT, PUBLIC, EXTRN

CLASS 3



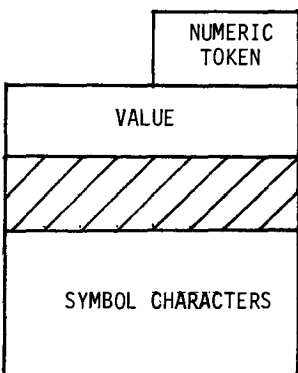
TOKEN - SKIP LINE

LINE TYPE - 1. STATEMENT LINE
2. MACRO CODE LINE
3. STATEMENT CODE LINE
4. TAG LINE

ERROR - one (1) byte ASCII ERROR code

Extra's 1&2 - reserved for future use

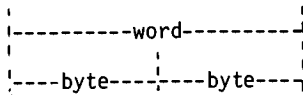
CLASS 4



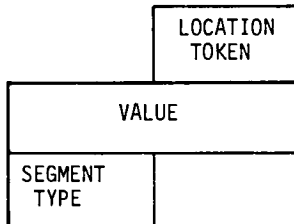
TOKEN - NUMERIC

VALUE - 16 bit integer value

SYMBOL CHARACTERS - array of characters which make up symbol.

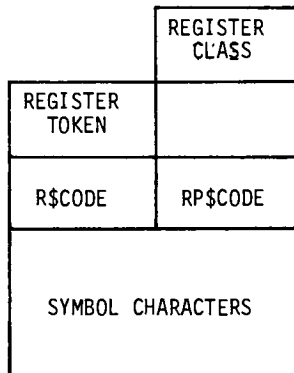


CLASS 5



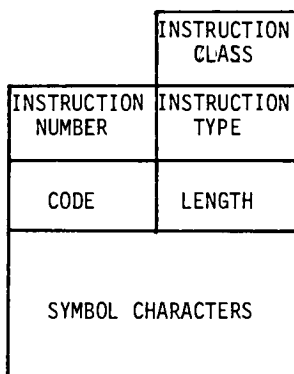
TOKEN - LOCATION
 VALUE 16 bit value
 SEGMENT TYPE- 1. CODE SEGMENT
 2. DATA SEGMENT
 3. ABSOLUTE SEGMENT

CLASS 6

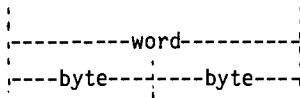


CLASS REGISTER
 TOKEN A, B, C, D, E, H, L, M, PSW, SP REGISTERS
 R\$CODE Binary code for single register
 RP\$CODE - Binary code for register pair

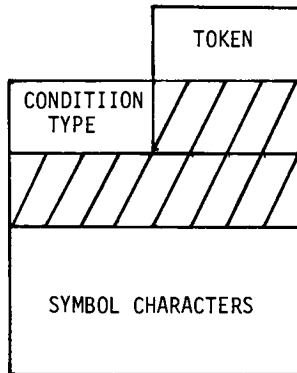
CLASS 7



CLASS INSTRUCTION
 INSTRUCTION NUMBER Instruction identification number
 INSTRUCTION TYPE Instruction format 0-12. Used when building object code.
 CODE - Object code for OPCODE part of instruction
 LENGTH - Bytes of object code needed for instruction

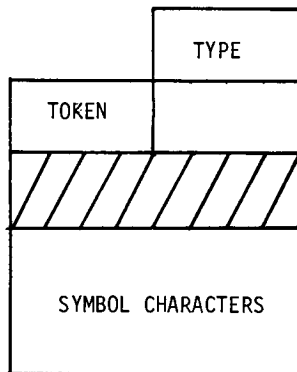


CLASS 8



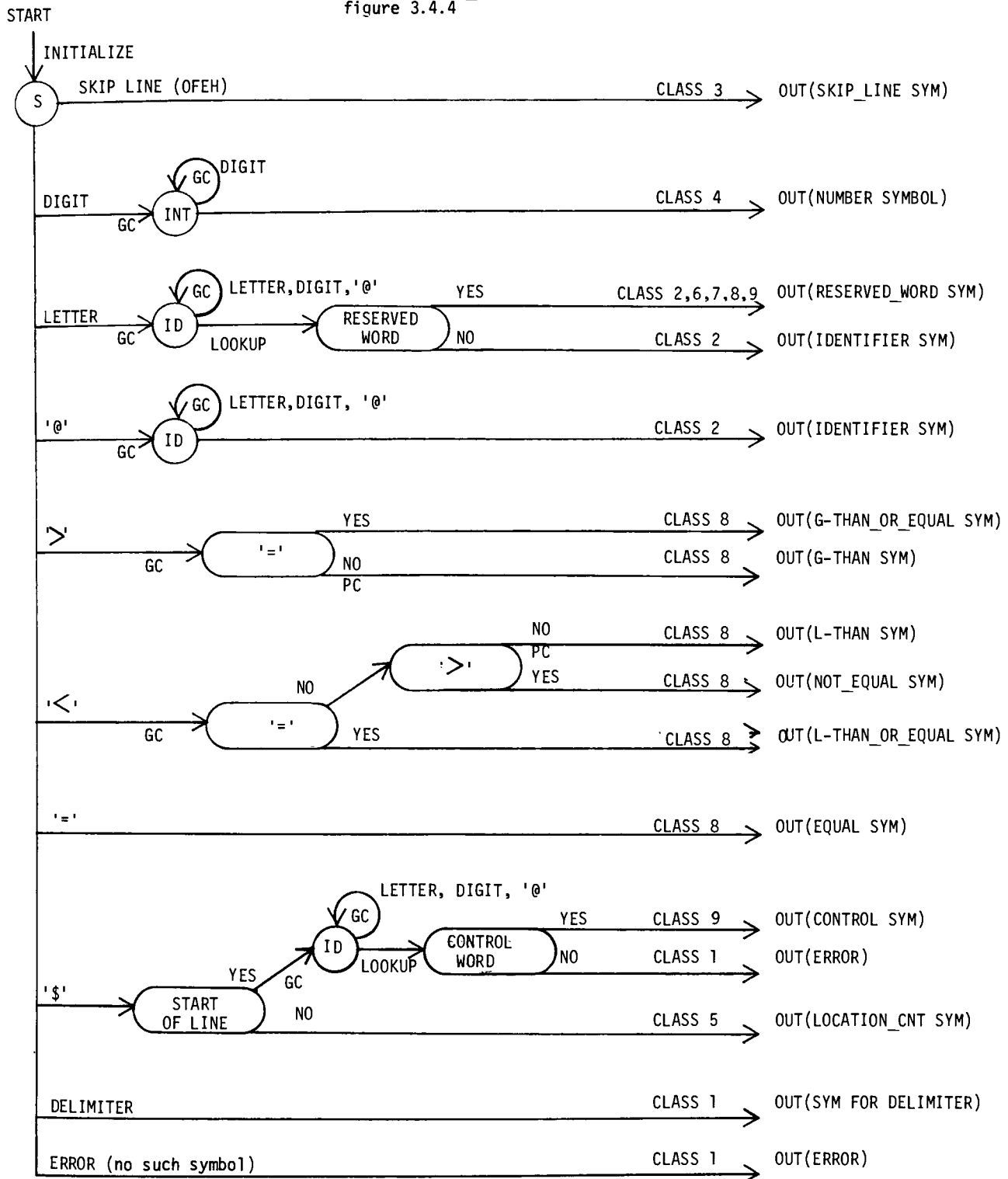
TOKEN - EQ, NE, GT, GE, LT, LE CONDITIONAL OPERATORS
 CONOITION TYPE Identifies class as conditional operator

CLASS 9



TYPE STATEMENT, CONTROL, MEMORY, DATA STORAGE
 TOKEN's
 STATEMENT TYPE
 -IF, FOR, CASE, WHILE
 CONTROL TYPE
 - CODE, LIST, NOLIST, PRINT, NOPRINT
 NOOBJECT, OEBUG, TITLE, MACRO_CODE
 MEMORY TYPE
 - CSEG, DSEG, ASEG, ORG
 DATA STORAGE TYPE
 - DS, DB, DW

figure 3.4.4



DELIMITER SYMBOLS - '.', ',', ';', ':', '+', '-', '*', '/', '_', '(', ')', carriage return, line feed

GC - Get next character

PC - Put back last character

3.4.3 Reserved Words

In order to decrease the number of searches of the RESERVED WORD TABLE (RWT), the routine SEARCH, compares the length of the string, which is to be searched, to the length of the longest possible reserved word. The maximum length of a Reserved Word is contained in the RESERVED WORD CONTROL TABLE (RWCT) (figure 3.4.5). Once it has been established that it is possible for such a reserved word to exist, the length of the string is used to index into the RWCT and get the pointer to the RWT where Reserve Words of that length can be found. If the pointer is zero (0), no such table exists and therefore no such reserved word exists. If the number of Reserved Words in a particular table is less than 30, a linear search is performed otherwise a binary search is made of the RWT. The information placed into SYMBOL BUFFER after a successful Reserved Word search can be accessed by Class 2,6,7,8 or 9 (figure 3.4.3) depending on the reserved word.

The current version of SMA80 has eight RESERVED WORD TABLES, one table for each of the eight possible lengths of the Reserved Words. Each Reserved Word Record (RWR) has six bytes which contains information about the reserved word, along with a variable length character buffer. The length of the character buffer is equal to the length of the Reserved Words found in a particular RWT.


RESERVED WORD TABLES
figure 3.4.5

RESERVED WORD
CONTROL TABLE
(RWCT)

MAXIMUM LENGTH OF A RESERVED WORD
Ptr to RWT with 1 char
Ptr to RWT with 2 char
Ptr to RWT with 3 char
Ptr to RWT with 4 char
Ptr to RWT with 5 char
Ptr to RWT with 6 char
Ptr to RWT with 7 char
Ptr to RWT with 8 char

ptr pointer.

RESERVED WORD
TABLE
(RWT)

NO. OF RESERVED WORDS	LENGTH OF CHAR FIELD
FIRST RESERVED WORD RECORD	
	
LAST RESERVED WORD RECORD	

RESERVED WORD RECORDS have
6 one byte fields plus the
the variable length character field

Pseudo Code for SEARCH
Figure 3.5.6

```
SEARCH (string length, pointer to RWCT, SYMBOL BUFFER):
  if (string length > max length of Rsv. Word) then
    return symbol_undefined;
  use string_length as index into RWCT to get the pointer, X,
    to the RWT with reserved words of string_length;
  if (X = 0) then
    return symbol_undefined
  set Low_index = 0;
  set High_index = no. of reserved words in RWT;
  if (High_index > linear search count) then
    perform binary search;
  else
    perform linear search;
  return status of search;
end SEARCH
```

Since each RWT has a different length Reserved Word Record (RWR), the size of the RWR for the RWT being searched is unknown. For this reason a simple index can not be used to access a RWR, instead the address of the RWR must be calculated and used to access the record.

RWTR_pointer = address of RWT + (Low_index * (6 +
length of the character buffer))

```
LINEAR_SEARCH:
  /* let STR_ident be identifier in symbol buffer */
  set RWR_pointer as described above;
  while (Low_index < High_index) do;
    if (STR_ident = Reserved Word) then
      do;
        copy RWR contents into symbol buffer;
        return sym_found;
      end;
    else
      do;
        increment Low_index;
        set RWR_pointer;
      end;
    end /* while */
  return sym_undefined;
end LINEAR_SEARCH;
```

RWR_pointer = address of RWT + (temp * (6 + length of the
character buffer));


```

BINARY_SEARCH:
  /* let STR_ident be identifier in symbol buffer */

  set temp = (Low_index + High_index)/2;
  set RWR_pointer;
  while (Low_index <= High_index) do;
    if (STR_ident = Reserved Word) then
      do;
        copy RWR contents into Symbol buffer;
        retrun sym_found;
      end;
    else
      if (STR_ident < Reserved Word) then
        do;
          High_index = temp - 1;
          set RWR_pointer;
        end;
      else /* STR_ident > Reserved Word) */
        do;
          Low_index = temp + 1;
          set RWR_pointer;
        end;
      end /* while */
      return sym_undefinied;
    end BINARY_SEARCH;

```

Note of interest: PLM/80 generates less code to calculate the address of the RWR as oppose to accessing the RWR as an array.

3.5 SYMBOL TABLE

During the process of assembling it is necessary to collect, store, and recall identifier names and information relating to these identifiers. This requires a data structure called the SYMBOL TABLE and the routines to manage it. The Symbol Table is an array of Identifier/Symbol Table Records. Management of the symbol table involves an efficient method of searching the table, as well as adding and removing information from it. Many methods are available for searching symbol tables. Some of more common methods are:

1. LINEAR SEARCH - This is the easiest to implement and in most cases the least efficient method of searching a symbol table. Symbol table entries are searched in the order that they were entered into the table.
2. SELF ORGANIZING TABLE - This method attempts to move the more frequently used identifiers to the top of the symbol table where they can be efficiently accessed by a linear search.
3. SEARCH TREE - In most cases this is a more efficient method than the previous two methods, especially when there are more than fifty (50) symbols in the symbol table. This method uses two pointers which connect the identifiers in the table to form a binary tree.
4. HASH TABLE - Of the methods mentioned this is the most efficient with large symbol tables. The method uses a hash

algorithm to create an index from the identifier name. This index is used to access a hash cell where the identifier might be found. If more than one identifier name produces the same index, a collision results. In this case the cell may contain a pointer to a table where identifiers with the same index can be found.

The SEARCH TREE method is used by the current version of SMA80. The SYMBOL TABLE RECORD (STR) in fig. 3.5.1, has two pointers called LCHILD and RCHILD. LCHILD contains a pointer to the STR whose ASCII identifier name is lexically less than the current STR's identifier name. RCHILD contains a pointer to the STR that has an identifier greater than the current identifier name. Since the buffer which holds the identifier name is eight bytes long, it is included in the STR.

The search proceeds through the binary tree created by these pointers based on the comparison of the identifier name that is being searched and the name in the current STR. Figure 3.5.2 contains an example of the symbol table organization.

SMA80's implementation of the Search Tree method involves four (4) routines. The four routines and their functions are:

1. SYM_INIT - This routine is called to initialize the size and location of the symbol table.

FORMAT OF SYMBOL TABLE RECORD (STR)

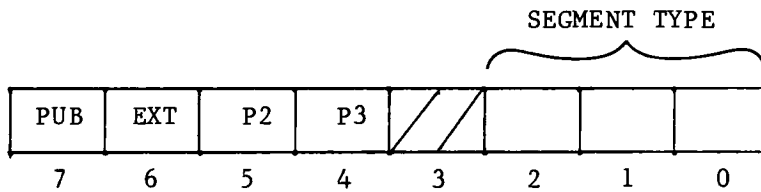
Figure 3.5.1

SYMBOL TYPE	STATUS
VALUE	
BUFFER CONTAINING IDENTIFIER NAME. 8 bytes	
LEFT CHILD POINTER	
RIGHT CHILD POINTER	

SYMBOL TYPE:

- LOCATION TAG
- EQUATE
- SET
- MACRO

STATUS:



- PUB - Identifier is declared public
- EXT - Identifier is declared external
- P2 - Set to 1 when defined during second pass
- P3 - Set to 1 when defined during third pass (prevents multiply definition)
- SEGMENT TYPE - CODE segment
- DATA segment
- ABSOLUTE segment
- EXTERNAL segment

- VALUE:
- Set to value of expression when identifier is a LOCATION TAG, EQUATE or SET identifier. Value is 16-bits.
 - Set to external number when identifier is defined EXTERNAL.
 - Set to address of macro definition buffer when defined as a MACRO.

RIGHT CHILD POINTER: Pointer to identifier numerically less than this one.

LEFT CHILD POINTER: Pointer to identifier numerically greater than this one.

2. `SYM_LOOKUP` - This routine performs a binary search of the symbol table. If the search is successful the contents of the STR is copied into the Symbol Buffer.
3. `SYM_ENTRY` - This routine fills the next available STR with the contents of the symbol buffer and takes care of the LCHILD and RCHILD pointer mapping.
4. `SYM_UPDATE` - This routine updates the contents of the STR with the contents of the symbol buffer.

One draw back to the STRING SEARCH method is the effect unbalanced trees have on the binary tree search. The worse case is when identifiers are entered into the symbol table in alphabetical order, so that each node of the tree has only one branch (fig 3.4.3). A binary search made on unbalanced trees such as that shown in figure 3.5.3 is effectively a linear search. Algorithms exist which balance the tree to avoid such situations, however they are not implemented by this version of SMA80.

3.5.2 Types of Identifiers

The four (4) types of identifiers stored in the symbol table by SMA80 are:

1. LOCATION TAG's - Identifiers used to define locations in memory,
2. EQUATE (EQU) - Identifiers used to define constants,

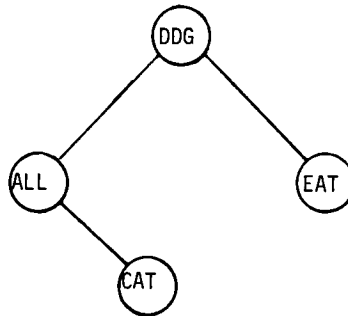
BALANCED BINARY TREE
figure 3.5.2

Identifiers entered in to SYMBOL TABLE in the following order: DOG, ALL, EAT, CAT .

SYMBOL TABLE

DOG	
LCHILD ALL	RCHILD EAT
ALL	
LCHILD 0	RCHILD CAT
EAT	
LCHILD 0	RCHILD 0
CAT	
LCHILD 0	RCHILD 0

BINARY TREE



This balanced tree takes at most 3 searches to find a match.

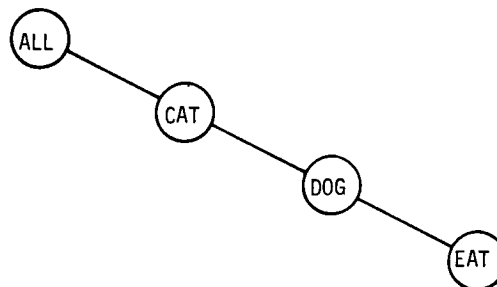
UNBALANCED BINARY TREE
figure 3.4.3

Identifiers entered into the SYMBOL TABLE in the following order: ALL, CAT, DOG, EAT.

SYMBOL TABLE

ALL	
LCHILD 0	RCHILD CAT
CAT	
LCHILD 0	RCHILD DOG
DOG	
LCHILD 0	RCHILD EAT
EAT	
LCHILD 0	RCHILD 0

BINARY TREE



This unbalanced tree takes as many as 4 searches to find a match.

3. SET - Same as EQU identifiers except SET identifiers can be redefined during assembly, and
4. MACRO - Names of Macro's.

3.5.3 Symbol Table Initialization

To efficiently use memory, the symbol table is initialized before both the first and second passes. During the first pass only MACRO, EQU and SET identifiers are entered into the symbol table. Since LOCATION TAGS are not put into the symbol table during this pass, a smaller symbol table can be used. The symbol table buffer is shared by the symbol table and the macro definition table. To make room in the symbol table buffer for the macro definition table the space available for the symbols is decreased to two thirds its normal size. The other third is used by the macro definition table.

Re-initializing the symbol table before the second pass returns that portion of the symbol table buffer previously used for macro definition to the symbol table. This larger symbol table is filled with EQU, SET and LOCATION TAG identifiers (but not the MACRO identifiers, for they were only needed by the first pass). Since the symbol table is re-initialized, LOCATION TAGs can have the same name as MACROS.

Pseudo Code for Symbol Table Management
Figure 3.5.3

```
SYM_INIT:
    set current pointer = 1;
    next entry point = 1;
    if (first pass) then
        set number of symbols = 200;
    else
        set number of symbols = 400;
end SYM_INIT;
```

```
SYM_LOOKUP:
    let identifier in symbol buffer = ident;
    let identifier in symbol table at current STR = STR_ident;

    while (next entry <> 1) and (current pointer <> 0) do
        if (ident = STR_ident) then
            do;
                copy all but identifier name and pointers from STR
                into symbol buffer;
                retrun SYM_EXIST;
            end;
        else
            if (ident < STR_ident) then
                do;
                    previous pointer = current pointer;
                    current pointer = LCHILD;
                    child = left; /* used by SYM_ENTRY */
                end;
            else /* if we get here we know (ident > STR_ident) */
                do;
                    previous pointer = current pointer;
                    current pointer = RCHILD;
                    child = right; /* used by SYM_ENTRY */
                end;
            end while;
        return SYM_UNDEFINIED
    end SYM_LOOKUP;
```

```
SYM_ENTRY: /* this routine can only be called after a
            call to SYM_LOOKUP */
    if (next entry point > no. of symbols) then
        return SYM_TABLE_OVERFLOW;
    copy identifier TYPE, STATUS, VALUE and NAME from
    symbol buffer into STR at next entry point;
    set LCHILD and RCHILD = 0;
    if (previous pointer <> 0) then /* must be first sym */
```



```
    if (child = left) then
        RCHILD of previous STR = next entry point;
    else
        LCHILD of previous STR = next entry point;
    increment next entry point;
end SYM_ENTRY;
```

```
SYM_UPDATE:  /* this routine can only be called
               after a call to SYM_LOOKUP */
    /* current pointer points to STR to be updated */
    copy TYPE, STATUS, VALUE from symbol buffer into
        STR at current pointer;
end SYM_UPDATE;
```

3.6 ARITHMETIC EXPRESSION EVALUATION

Many of the SMA80 instructions have operand fields in the form of arithmetic expressions. SMA80's expression evaluation routine, EXPRESSION, does not generate code which will calculate the expression at runtime, instead it returns the assembly-time value of the expression and fills the global data structure, EXPRESS with an error code describing any error that may have been found during the evaluation and data type (memory type) of the value. EXPRESSION is a recursive routine implemented using top-down, recursive descent parsing.

Operator Precedence
figure 3.6.1

HIGHEST	(,)
.	*, /, MOD
.	-, +
LOWEST	AND, OR, XOR

As the expression is being parsed and having its value calculated, it is checked for compatible DATA TYPE's (figure 3.6.2). Once the calculation is complete the value of the expression is checked to see if it is of the data type requested by the calling routine, if one was specified. If the calling routine requested a byte varue, a value from 0 to 255, this is also checked. If either of the above checks fails an ASCII error code is placed into the error field of the EXPRES data structure.

3.6.1 Recursive descent Parsing

The recursive descent parsing technique used by SMA80 to implement expression evaluation employs the five procedures EXPRESSION, ADDITIVE, MULTIPLICATIVE, LOGICAL and FACTOR, whose pseudo code can be found in figure 3.6.2. The operators and their priorities are shown in figure 3.6.1. The grammar for the language is,

```
E ::= A
A ::= M + A | M - A | -A | NOT A | M
M ::= A * L | A / L | A MOD L | L
L ::= A AND F | A OR F | A XOR F | F
F ::= ( A ) | NT
NT ::= identifier | instruction | location counter |
      ascii string | numeric
```

3.6.2 Data Type's

There are effectively four (4) DATA TYPEs recognized by SMA80, which are:

1. CODE - data in a code memory segment declared by a CSEG memory instruction,
2. DATA - data in a data memory segment declared by a DSEG memory instruction,
3. ABSOLUTE - which fall into various catagories:
 - a. data in a absolute memory segment declared by a ASEG memory instruction,
 - b. numeric data,
 - c. EQU and SET identifiers which are definied by ABSOLUTE data,

- d. INSTRUCTION opcodes, and
 - e. ASCII String - a string with a length of 1 character (2 characters if a word length value is acceptable). The string must be surrounded by quotes, and
4. EXTERNAL data - data declared external to the program by the EXTRN instruction. Even though External data exists as one of the above data type, exactly which one is unknown because it has been declared outside the scope of this program. External identifiers are identified by an EXTERNAL_NUMBER that is stored in the Value field of the Symbol Table Record. This value is carried along through the expression and stored in the External_number field of EXPRES.

NOTE: SMA80 has the capability of separating memory used by programs into three separate segments called CODE, DATA and ABSOLUTE memories. SMA80 has no way of knowing the relation between the CODE and DATA memories, the interactions between these memories are resolved by the LINKER. This is the reason for treating each of the memory segments as different data types.

COMPATIBLE DATA TYPE's
figure 3.6.2

Expression	Exceptions	Output Data Type
SEG - SEG	External data	ABSOLUTE
SEG (+ ! -) ABS		SEG TYPE
ABS (+ ! -) SEG		SEG TYPE
ABS (+ ! -) ABS		ABSOLUTE
SEG (* ! /) ABS	External data	SEG TYPE
ABS (* ! /) ABS	External data	SEG TYPE
ABS (* ! /) ABS		ABSOLUTE
SEG (AND !OR !XOR) ABS	External data	SEG TYPE
ABS (AND !OR !XOR) ABS	External data	SEG TYPE
ABS (AND !OR !XOR) ABS		ABSOLUTE
- ABS		ABSOLUTE
NOT(ABS)		ABSOLUTE

SEG TYPE - CODE, DATA, or EXTERNAL unless exceptions are specified.

3.5.3 Pseudo Code for EXPRESSION Modules

The expression routine consist of five modules, EXPRESSION, ARITHMETIC, MULTIPLICTIVE, LOGICAL and FACTOR. All modules are recursive, except for EXPRESSION which just invokes the expression analyzing process and checks for calling routine requests.

```
EXPRESSION (return_value, required_segment_type, data_size):
  /* The global data structure EXPRES is filled with the
    segment type, error, and external number of the
    calculated value. */

  call ADDITIVE /* returns the value and segment type */
  /* calculation is now complete */
  if (there is a required_segment_type and the segment type
    returned by ADDITIVE is not the same) then
    set EXPRES error = illegal data type (D);
  else
    if (the data_size is byte and the value > 255) then
      set EXPRES error = data overflow (O);
    set return_value = the value returned by ADDITIVE
end EXPRESSION;
```

```
ADDITIVE (data structure for return_value and segment_type);
  /* This routine handles the lowest priority operators
    '+' and '-'. It also takes care of the special case
    unary minus and NOT. */

  /* unary '-', '+' and NOT */
  if (operator = '+' or '-' or NOT) then
    do;
      save operator
      call GET_SYM;
      call MULTIPLICATIVE /* returns value and segment */
      if (segment <> ABSOLUTE) the
        set EXPRES error = illegal expression (E);
      else
        if (saved operator = '-') then
          value = 0-value /* make value negative */
        else
          if (saved operator = NOT) then
            not(value);
        end;
      else
        call MULTIPLICATIVE /* set value structure #1 */
      if (there is an error) then
        return to calling module;

    while (operator = '+' or '-') do
      save operator
      call GET_SYM;
```

```

call MULTIPLICATIVE /* set value structure #2 */
if (there is an error) then
    return to calling module;

if (saved operator = '+') then
do;
    if (value_segment_1 or value_segment_2
        = ABSOLUTE) then
        do;
            return_value = value_1 + value_2;
            set return_segment to non-ABSOLUTE segment
            otherwise set to ABSOLUTE;
        end;
    end;
else
    set EXPRES error = illegal expression (E);
else /* saved operator must be '-' */
do;
    if (one of the segments is ABSOLUTE or the two
        segments are equal but not EXTERNAL segments)
        then
        do;
            return_value = value_1 - value_2;
            set return_segment to non-ABSOLUTE segment
            otherwise set to ABSOLUTE;
        end;
    end;
else
    set EXPRES error = illegal expression (E);
end;
end /* while */;
end ADDITIVE;

```

```

MULTIPLICATIVE (data structure for return_value and segment):
/* This module handles the '*', '/' and MOD operators */

call LOGICAL /* set value structure #1 */;
if (there is an error) the
    return to calling module;

while (operator = '*' or '/' or MOD) do
    save operator;
    call GET_SYM;
    call LOGICAL /* set value structure #2 */;
    if (there is an error) then
        return to calling module;

    if (value_segment_1 or value_segment_2 = ABSOLUTE and
        value_segment_1 or value_segment_2 <> EXTERNAL seg)
        then
        do;
            if (saved operator = '*') then

```

```

        return_value = value_1 * value_2;
    else
        if (saved operator = '/') then
            return_value = value_1 / value_2;
        else /* saved operator must be MOD */
            return_value = value_1 MOD value_2;
        set return_segment equal to non-ABSOLUTE segment
        otherwise set to ABSOLUTE;
    end;
else
    do;
        set EXPRES error = illegal data type (D);
        return to calling module;
    end;
end /* while */;
end MULTIPLICATIVE;

```

```

LOGICAL (data structure for return_value and segment):
/* This module handles the AND, OR and NOT logical
   operators */

call FACTOR /* set value structure #1 */;
if (there is an error) then
    return to calling module;

while (operator = AND or OR or XOR) do
    save operator;
    call GET_SYM;
    call FACTOR /* set value structure #2 */;
    if (there is an error) then
        return to calling module;

    if (value_segment_1 or value_segment_2 = ABSOLUTE and
        value_segment_1 or value_segment_2 <> EXTERNAL seg)
    then
        do;
            if (saved operator = AND) then
                return_value = value_1 AND value_2;
            else
                if (saved operator = OR) then
                    return_value = value_1 OR value_2;
                else /* saved operator must be XOR */
                    return_value = value_1 XOR value_2;
                set return_segment equal to non-ABSOLUTE segment
                otherwise set to ABSOLUTE;
            end;
        ELSE
            do;
                set EXPRES error = illegal data type (D);
                return to calling module;
            end;
        end;
    end;
end;

```



```

        end /* while */;
end LOGICAL;

```

```

FACTOR (data structure for return_value and segment type);
/* This routine resolved the no-terminal elements of
the arithmetic expression */

if (token = identifier) then
do;
    call SYM_LOOKUP;
    if (SYM_UNDEFINIED) then
        set EXPRES error = sym_undefinied (U);
    else
        do;
            set return_segment to identifier segment type
            from symbol table;
            if (segment_type = EXTERNAL seg) then
                do;
                    set EXPRESS external number = value from
                    symbol table;
                    set return_value = 0;
                end;
            else
                return_value = value from symbol table;
            end;
        call GET_SYM;
    end;
else
    if (token = number) then
        do;
            return_value = value of number;
            return_segment = ABSOLUTE;
        end;
    else
        if (token = INSTRUCTION) then
            do;
                return_value = instruction code;
                return_segment = ABSOLUTE;
            end;
        else
            if (token = '') then
                do;
                    Set return_value = binary value of ASCII
                    string;
                    set return_segment = ABSOLUTE;
                    if (error in string) then
                        set EXPRES error = expression error 'E';
                    end;
                end;
            else
                if (token = '(') then
                    do;

```

```

        call GET_SYM;
        call ADDITIVE /* set return value and
                        segment data structure */
                        /* THIS IS A RECURSIVE CALL */
        if (token <> '(') then
            set EXPRES error = missing parenthesis
            (P);
        else /* illegal non-terminal symbol */
            end;
    else
        if (token = LOCATION_CNT) then
            do;
                return_value = location value;
                return_segment = location segment;
            end;
        else
            set EXPRES error = illegal token (I);
    end FACTOR;

```

3.7 THE THREE PASSES OVER THE SOURCE CODE

Up to this point the tools necessary for building and maintaining the input line and symbol table, the routine for analyzing arithmetic expressions, and assembly controls have been presented. This section shows how these tools are used by each of the three passes of SMA80.

3.7.1 The First Pass

In SMA80's first pass over the source code, the functions that separate SMA80 from normal assemblers are performed. This pass is responsible for,

1. generating an intermediate source code file,
2. evaluating conditional assembly statements,
3. defining and expanding Macro's, and
4. generating SMA80 source code to execute the IF-THEN-ELSE, WHILE, FOR and CASE statements.

This section will present the techniques used to implement each of these functions.

3.7.1.1 The Intermediate File

The output of the first pass is placed into an intermediate file on disk called SMA80.TMP. This file is placed on the same disk as the source code file specified in the runstring. SMA80.TMP is used as the input for the second and third passes. The intermediate file contains the,

1. code from the source file specified in the runstring,
2. code from INCLUDE files if any were specified,
3. source code generated from the structured statements, and
4. code generated by macro expansion.

Pass one appends to the front of certain lines a five (5) byte SKIP LINE HEADER (SLH) that directs the second and third passes as to what type of action to take for a particular line. These lines include,

1. the structured statements (IF, WHILE, FOR and CASE) and their associated terminal symbols (i.e., ELSE, ENDIF, ENDWHILE, ENDFOR, DO, ENDDO and ENDCASE),
2. macro definitions,
3. source code generated by the structured statements,
4. source code generated by macro expansion, and
5. the control statements processed during the first pass.

The first byte of the SLH is used to flag the second and third passes that special action is to be taken, the second byte (action code)

defines the action and the third byte contains an ASCII character describing the error found by the first pass when the line was processed, if any. The fourth and fifth bytes are unused at this time. The four action codes used and there meanings are:

1. STATEMENT_CODE - do not process line, just pass it to the print routine.
2. TAG_LINE - the location tag on the line must be processed and the remainder of the line ignored.
3. STATEMENT_CODE - the line contains SMA80 source code generated by a structured statement. The line needs to be processed.
4. MACRO_CODE - the line contains SMA80 source code generated by macro expansion. The line needs to be processed.

The SKIP STATEMENT HEADER is also used by the print routine to determine the lines to be listed. Code generated by structured statements and macro expansions are only listed if the CODE and EXPMACRO controls have been issued. If an error was logged in the error field of the SLH the print routine increments the error count and displays the error when the line is listed.

For example the source code line,

```
TAG1:      IF (COUNT1 < COUNT2 THEN ,
```

will produce a SLH with an action code (byte two of the SLH) of TAG_LINE, and the error field will contain an ASCII "P" indicating a missing a parenthesis.

The routines responsible for managing the output to the intermediate file, SMA80.TMP are:

1. COPY_SOURCE_INIT - sets the output buffer count to zero.
2. COPY_SOURCE_CHAR - this routine takes a pointer to a source code buffer and a character count and copies the source buffer to the output buffer. If the output buffer becomes full it is copied to the intermediate file on disk.
3. COPY_SOURCE - this routine takes a pointer to a source buffer and a character count. If the count is equal to 0FFFFH, the source buffer is copied into the output buffer up to and including the first line feed, otherwise the number of characters specified by the count are copied from the source buffer into the output buffer and a carriage return and line feed are appended to the end. If the output buffer becomes full it is copied to disk.
4. COPY_SOURCE_FINISH - is called at the end of the pass to copy the last of the output buffer into the intermediate file.

Beginning Of The First Pass

Once the PASS1 overlay has been read into memory, control is transferred to the routine MAIN_PASS1 which is responsible for initializing the,

1. symbol table (SYM_INIT),
2. intermediate file (FILE_INIT_PASS1),
3. location tag generator (TAG_INIT),
4. line buffer (GET_LINE_INIT), and
5. macro processor (MACRO_INIT).

The first pass uses recursive descent parsing and begins with a call to PASS1_COND. PASS1_COND checks the current symbol against five stop symbols. These five stop symbols consist of END_OF_FILE_symbol, END_symbol, and three dynamic symbols, COND1, COND2 AND COND3, that are set by the calling routine. For example the IF_PROC routine sets COND1 and COND2 to ENDIF_symbol and ELSE_symbol, and COND3 to NULL (not used). Once the current symbol matches one of the stop symbols control is returned to the calling routine, otherwise PASS1_PROC is called to determine what action is to be taken for the current symbol, if any.

The extensive use of recursive procedures in this pass makes it is necessary to determine if enough room is available on the stack before a call is made to a reentrant routine. STACK_CHECK is the

routine that checks the room available on the stack. `STACK_CHECK` is called before the reentrant call to increment the stack count and after the reentrant procedure returns in order to set the stack count to its pre-call value. If a stack overflow occurs an error message along with the current contents of the `LINE BUFFER` are displayed on the console and `SMA80` is aborted.

Pseudo Code for PASS1_COND and PASS1_PROC
Figure 3.7.1

PASS1_COND:

```
/* This routine is passed the three dynamic stop symbols and the
   new current block type.
*/
save the previous block;
set the current block to the new block type parameter;
get the next line and first symbol on the line;
WHILE (symbol is not one of the five stop symbols) do
    call PASS1_PROC to see what action to take for symbol;
    get next line and then first symbol on the line;
end /* while */
restore the previous block type;
end PASS1_COND;
```

PASS1_PROC:

```
IF (symbol is IF_symbol) then
    do;
        call STACK_CHECK /* see if room on stack for IF_PROC */
        IF_PROC;
    end;
else

IF (symbol is ELSE_symbol) then
    IF (current block is not if block) then
        copy line to intermediate file with an illegal statement
        error (I) in the SKIP LINE HEADER (SLH);
    else

IF (symbol is ENDIF_symbol) then
    IF (current block is not if block) then
        copy line to intermediate file with an illegal statement
        error (I) in the (SLH);
    else call STACK_CHECK /* reset stack count */
else

IF (symbol is WHILE_symbol) then
    do;
        call STACK_CHECK /* see if room on stack for WHILE_PROC */
        call WHILE_PROC;
    end;
else

IF (symbol is ENDWHILE_symbol) then
    IF (current block is not while block) then
        copy line to intermediate line with an illegal statement
        error in the SLH;
    else call STACK_CHECK /* reset stack count */
else
```

```

IF (symbol is FOR_symbol) then
  do;
    call STACK_CHECK /* see if room on stack for FOR_PROC */
    call FOR_PROC;
  end;
else

IF (symbol is ENDIF_symbol) then
  IF (current block is not for_block) then
    copy line into intermediate file with an illegal statement
    error in the SLH;
  else call STACK_CHECK /* reset stack count */
else

IF (symbol is CASE_symbol) then
  do;
    call STACK_CHECK /* see if room on stack for CASE_PROC */
    call CASE_PROC;
  end;
else

IF (symbol is ENDCASE_sym) then
  IF (current block is not case block) then
    copy line into intermediate file with an illegal statement
    error in the SLH;
  else call STACK_CHECK /* reset stack count */
else

IF (symbol is DO_symbol) then
  IF (current block is not case block) then
    copy line into intermediate file with an illegal statement
    error in the SLH;
  else

IF (symbol is ENDDO_symbol) then
  IF (current block is not case block) then
    copy line into intermediate file with an illegal statement
    error in the SLH;
  else

IF (symbol is CONDIF_symbol) then
  do;
    call STACK_CHECK /* see if room on stack for CONDIF_PROC */
    CONDIF_PROC;
  end;
else

IF (symbol is ELSECOND_symbol) then
  IF (current block is not condif block) then
    copy line to intermediate file with and illegal statement
    error (I) in the SKIP LINE HEADER (SLH);
  else

```

```

IF (symbol is ENDCOND_symbol) then
  IF (current block is not ifcond block) then
    copy line to intermediate file with an illegal statement
    error (I) in the (SLH);
  else call STACK_CHECK /* reset stack count */
else

IF (symbol is IDENT_symbol) then
  do;
    call STACK_CHECK /* see if room on stack for IDENT_PROC */
    call IDENT_PROC /* store MACRO, SET and EQT identifiers not
                    location tags */
  end;
else

IF (symbol is CONTROL_symbol) then
  call CNTL$PROC;
else

/* pass1 takes no action on this line */
copy line to intermediate file without a SLH;

end PASS1_PROC;

```

Symbol Table During The First Pass

The symbol table is needed during the first pass for processing conditional assembly (section 3.7.1.3) and macros (section 3.7.1.4). The SMA80 routine that defines SET and EQU identifiers, and invokes the definition and expansion of macros is IDENT_PROC. The implementation of IDENT_PROC can best be described by the pseudo code in figure 3.7.2.

Psuedo Code For IDENT_PROC
Figure 3.7.2

```
IDENT_PROC:
  call SYM_LOOKUP to see if symbol exist;
  save the contents of the symbol buffer into temp_ident;
  call GET_SYM to get the next symbol;
  if (symbol is SET_symbol) then
    do;
      get the value of the expression that follows;
      if (sym_lookup said symbol exist) then
        /* the value of a SET identifier be changed during
           assembly time */
        update symbol table with new value;
      else
        enter the new symbol and its value and status into
        the symbol table;
    end;
  else if (symbol is EQU_symbol) then
    if (sym_lookup said symbol does not exist) then
      do;
        /* an EQU identifier can not change during
           assembly time so do not chang it if it already
           exist */
        get value of expression that follows;
        if (data type of expression is absolute) then
          make new entry in the symbol table;
        else do nothing /* use of a not absolute identifier
                        conditional assembly will cause
                        an error */
      end;
    else do nothing /* EQU identifiers can not be changed once
                    they are defined */
  else if (symbol is COLON_symbol) then
    do;
      set the identifier_flag on to indicate to the
      processing of the rest of the line that the line
      starts with a location tag;
      call PASS1_PROC to see if the remaining part of
      the line after the location tag requirers
      processing "THIS IS A RECURSIVE CALL ";
    end;
  else if (symbol is MACRO_symbol) then
    call MACRO_DEFINE to define the macro;
    else if (the identifier copied into temp_ident
             exist and it is defined as being a
             macro) then
      call MACRO_EXPAND;
  if (the line has not already been copied to the intermediate file
      when control was passed to PASS1_PROC) then
    copy line to intermediate file;
end IDENT_PROC;
```

3.7.1.2 Structured Statements

The generation of code for the IF-THEN-ELSE, WHILE, FOR and CASE statements is the responsibility of the first pass. The number of levels to which structured statements can be nested is not restricted to a fixed value, rather it is restricted by the amount of available stack space. Each of the routines that processes the structured statements is reentrant and store their data on the stack. The statements require varying amounts of stack space, making the number of nested levels dependent on the arrangement of statements used. The stack space requirements for each of the structured statements is,

1. IF-THEN-ELSE - 41 bytes,
2. WHILE - 41 bytes,
3. FOR - 41 bytes,
4. CASE - 242 bytes.

These stack requirements include the four bytes needed for the call to the statement procedures and the two bytes needed for the subsequent call to PASS1_COND made by each of the procedures.

Current Block

Throughout this section the term CURRENT BLOCK is often referred to. The current block takes on one of the values:

1. null,
2. IF_block,
3. WHILE_block,
4. FOR_block,

5. CASE_block, or

6. CONDITIONAL_ASSEMBLY_block,

depending upon the type of statement whose body is currently being processed. A terminal symbol associated with a particular statement (i.e.. ELSE, ENDFOR, ENDWHILE, ENDFOR, DO, ENDDO, ELSECOND, ENDCASE) has no effect, other than producing an error, if the current block is not equal to that statement.

For example lines 2-4 of figure 3.7.3 are in the IF_block and lines 5-8 are in the WHILE_block even though the WHILE statement is nested within the IF statement. Since line 7, ENDIF, has no meaning in a WHILE block it does not terminate the IF statement. An 'I' (illegal statement) will be put into the error field of the SKIP LINE HEADER for this line.

Figure 3.7.3
Current Block Example

```
1      IF (COUNT1 < COUNT2) THEN
2          LXI H,COUNT1
3          STA COUNT2
4          WHILE (COUNT1 < .10) DO
5              LXI H, COUNT1
6              INR M
7          ENDIF
8          ENDWHILE
9      ENDIF
```

Registers

One of the more valuable resources to the assembly language programmer is the general purpose register. SMA80 uses as few of these registers as possible when generating code for the structured statements. The IF-THEN-ELSE, WHILE and FOR statements use the accumulator (A) and HL register pair, and the CASE statement uses the accumulator and both the HL and DE register pairs.

Location Tags

Any structured statement or line containing a terminal symbol associated with a structured statement can have a location tag on the front of the line.

Indentation

In an attempt to make the listing more readable the code generated by a structured statement is indented three (3) spaces in from the column in which the statement started.

The IF-THEN-ELSE Statement

The processing of the IF statement is performed by the PASS1 procedure IF_PROC, which is called from PASS1_PROC. The IF statement

implemented by SMA80 includes the optional ELSE clause. The format of the statement is,

```
IF <condition> THEN <statement(s)> ENDIF
```

or

```
IF <condition> THEN <statements(s)> ELSE <statement(s)> ENDIF.
```

The code generated by the IF statement is shown in figure 3.7.4. The implementation of the IF statement can best be described by the pseudo code in figure 3.7.5.

The routine CONDITION that generates the code to evaluate the condition is only able to compare byte values (i.e., values between 0 and 255 decimal). CONDITION is used by both the IF and WHILE statements. A pseudo code description of CONDITION can be found in figure 3.7.6.

LDC CODE DATA LINE SOURCE CODE

ISIS-II SMA-80 ASSEMBLER

SMA-80 INVOKED BY SMABO (F1:IFEXP.SMA PRINT(:LP:)) CODE

```

1  $TITLE(EXAMPLE OF IF STATEMENT)
2  COUNT1 EQU 10
3  COUNT2 EQU 20
4
5  IF (LOC1 = COUNT1) THEN
6      LDA LOC1
7      CPI COUNT1
8      JNZ @00000
9      LXI H,LOC1
10     STA COUNT2
11     JMP @00001
12 ELSE
13     @00000:
14     IF (COUNT1 = LOC1) THEN
15         LDA COUNT1
16         LXI H,LOC1
17         CMP M
18         JNZ @00002
19         RET
20     ENDIF
21     @00002:
22     EXIT LOCATION FOR NESTED IF
23     ENDIF
24     @00001:
25     LDC1 DS 1
26     EXIT LOCATION FOR OUTER IF
27     END

```

CODE TO COMPARE THE TWO EXPRESSIONS. NOTICE THE PERIOD IN FRONT
COUNT1 CAUSES THE EXPRESSION TO BE USED AS THE ACTUAL VALUE.

PROGRAMMERS CODE FOR THEN SECTION OF THE IF-THEN-ELSE STATEMENT
JUMP TO EXIT AFTER EXECUTING THE THEN CODE

START OF ELSE CODE IN CASE THE CONDITION FAILS

CODE TO COMPARE THE TWO EXPRESSIONS. NOTICE THAT COUNT1 IS USED
THE ADDRESS OF THE VALUE. THIS IF STATEMENT DOES NOT USE THE
ELSE OPTION.

PROGRAMMERS CODE

MODULE INFORMATION

CODE AREA SIZE = 1DH 29D
O PROGRAM ERROR(S)

END OF SMABO

LOCATION TAGS	MEANING
@00000	ELSE CODE FOR THE FIRST IF STATEMENT
@00001	EXIT LOCATION FOR THE NESTED IF STATEMENT
@00002	EXIT LOCATION FOR THE OUTER IF STATEMENT

Pseudo Code For IF_PROC
Figure 3.7.5

```
IF_PROC:
  call G_CHAR_CNT to get the column number that the IF symbol
    starts and set the indentation count;
  call CONDITION to evaluate the IF condition, generated code
    indented to indentation count;
  if (the next symbol is not THEN_symbol) then
    set error field of the SKIP LINE HEADER to 'T' (missing then);
  copy source line with SLH appended to it to the intermediate file;
  if (the CONDITION routine found any errors ) then
    return to PASS1_PROC;

  copy the code generated by CONDITION into the intermediate file;
  /* set stop symbols to ELSE, ENDIF and null(not used) */
  call PASS1_PROC /* get the programmers code that falls within
    the THEN segment of the IF statement */

  if (symbol is ELSE_symbol or ENDIF_symbol) then
    do;
      if (symbol is ELSE_symbol) then
        do;
          /* exit location tag points to end of IF statement */
          copy the EXIT location tag generated in the CONDITION
            routine to the ELSE location tag buffer;
          call JUMP_TAG to generate a new EXIT location tag;
          copy a jump instruction to the new EXIT location tag
            into the intermediate file. /* this jump allows
              the THEN code to jump around the ELSE code */
          copy the source line containing the ELSE to the
            intermediate file;
          copy the ELSE location tag into the intermediate file;
          /* the CONDITION routine generated code to jump to
            the EXIT location tag when the condition failed.
            By copying the old EXIT tag to the ELSE tag
            failure of the condition jumps to the else
            code*/
          /* set stop symbols to ENDIF_symbol, null, null */
          call PASS1_COND /* get the code that falls within
            ELSE segment of the IF statement */
        end;
      copy the ENDIF line to the intermediate file;
      copy the EXIT location tag to the intermediate file;
    end;
  end IF_PROC;
```

Pseudo Code For CONDITION
Figure 3.7.6

CONDITION:

```
/* This routine places the code it generated into a code buffer
   which is passed back to the calling routine
*/

call JUMP_TAG to create the EXIT location tag;
get the first expresson of the condition;
/* generate code to get the first expression */
if (the expression is a number or starts with a period '.') then
    /* the expression is the actual value and not the address
       of the value */
    call GEN_CODE to generate a "MVI A,expression" instruction;
else
    /* use the expression as the address of the data to be
       compared */
    call GEN_CODE to generate a "LDA expression" instruction;
get the next symbol;

if (symbol is conditional operator) then
    save operator;
else
    do;
        /* must be a single expression comparsion,check for true */
        call GEN_CODE to generate a "RRC" instruction to check
            bit zero (0), if high true;
        call GEN_CODE to generate a jump to the EXIT location tag if
            the value is not true;
        return to calling procedure IF_PROC or WHILE_PROC;
    end;

get the second expresson of the condition;
/* generate code to get the second value */
if (the expression is a number or starts with a period '.') then
    /* expression is value */
    call GEN_CODE to generate a "CPI expression" instruction;
else
    do;
        /* expression is address of data */
        call GEN_CODE to generate "LXI H,expression" and
            a "CMP M" instruction;
    end;
/* generate to exit if the condition is false */

/* generate a jump instruction(s) to jump to exit (or else) if the
   condition fails based on the conditional operator
*/
if (operator is EQ_symbol) then
    call GEN_CODE to generate "JNZ exit_tag"
```

```

else if (operator is NE_symbol) then
    call GEN_CODE generate "JZ exit_tag"
else if (operator is GT_symbol) then
    do;
        /* jump if less than */
        call GEN_CODE to generate "JC exit_tag";
        /* jump if equal */
        call GEN_CODE to generate "JZ exit_tag";
    end;
else if (operator is GE_symbol) then
    call GEN_CODE to generate "JC exit_tag";
else if (operator is LT_symbol) then
    do;
        /* jump if greater than or equal */
        call GEN_CODE to generate "JNC exit_tag"
    end;
else if (operator is LE_symbol) then
    call GEN_CODE to generate a
        "JZ $+3" /* jump ahead 3 bytes */
        and a "JNC exit_tag";

return to the calling routine with the generated code in the
the code buffer.;
end CONDITION;

```

The WHILE Statement

The processing of the while statement is performed by the PASS1 procedure WHILE_PROC. The format of the WHILE statement is,

```
WHILE <condition> DO <statement(s)> ENDWHILE.
```

Code is generated for the condition by the routine CONDITION, (figure 3.7.5) the same one used by the IF statement.

Figure 3.7.7 shows the code generated for the WHILE statement by WHILE_PROC (figure 3.7.8.) and CONDITION (figure 3.7.6).

LOC	CODE DATA	LINE	SOURCE CODE
-----	-----------	------	-------------

IS-II SMA-80 ASSEMBLER
 A-80 INVOKED BY: SMABO : F1:WHLEXP SMA PRINT(:LP:) CODE

000A		1	\$TITLE(EXAMPLE OF WHILE STATEMENT)
		2	COUNT1 SET 10
		3	
		4	WHILE (LOC1 <= COUNT1) DO
		5	@00001:
0000 3A	1200 C	6	LDA LOC1
0003 FE	0A	7	CPI COUNT1
0005 CA	0B00 C	8	JZ \$+3
0008 D2	1200 C	9	JNC @00000
000B 21	1200 C	10	LXI H, LOC1
000E 35		11	INR M
000F C3	0000 C	12	JMP @00001
		13	ENDWHILE
		14	@00000: EXIT LOCATION FOR WHILE STATEMENT
		15	
0012		16	LOC1: DS 1
		17	END

MODULE INFORMATION

CODE AREA SIZE = 13H 19D
 0 PROGRAM ERROR(S)

END OF SMABO

LOCATION TAGS	MEANING
@00000	EXIT LOCATION FOR WHILE STATEMENT
@00001	START OF WHILE LOOP

Pseudo Code For WHILE_PROC
Figure 3.7.8

```
WHILE_PROC:
  call G_CHAR_CNT to get the column number that the WHILE symbol
    starts and set the indentation count;
  call CONDITION to evaluate the WHILE condition, generated code
    indented to indentation count;
  if (the next symbol is not DO_symbol) then
    set error field of the SKIP LINE HEADER to 'D' (missing DO);
  copy source line with SLH appended to it to the intermediate file;
  if (the CONDITION routine found any errors ) then
    return to PASS1_PROC;

  /* the start location tag points the beginning of the loop */
  call JUMP_TAG to generate the START location tag;
  copy the START location tag into the intermediate file;
  copy the code generated by CONDITION into the intermediate file;
  /* set the stop symbols to ENDEWHILE_symbol, null and null */
  call PASS1_PROC /* get the programmers code that falls within the
    while statement block */

  if (symbol is ENDEWHILE_symbol) then
    do;
      call GEN_CODE to generate a "JMP to start_location "
      copy the ENDEWHILE line to the intermediate file;
      copy the EXIT location tag that was created by CONDITION
        to the intermediate file;
    end;
end WHILE_PROC;
```


The FOR Statement

FOR_PROC is the PASS1 procedure that directs the processing of the FOR statement. The format of the FOR statement is,

```
FOR <identifier> = <expression_1> TO  
    <expression_2> DO <statement(s)> ENDFOR,
```

or

```
FOR <identifier> = <expression_1>  
    TO<expression_2> BY <expression_3>  
    DO <statement(s)> ENDFOR.
```

Figure 3.7.9 contains an example of the code generated by FOR_PROC (figure 3.7.10) and FOR_CODE (3.7.11) for the FOR statement.

LOC CODE DATA LINE SOURCE CODE

```
SIS-II SMA-80 ASSEMBLER
MA-80 INVOKED BY: SMABO :F1 FOREXP SMA PRINT(:LP: ) CODE

1 $TITLE(EXAMPLE OF FOR STATEMENT)
2 FOR I = 1 TO 10, DO
3   LXI H, I
4   MVI M, 1 CODE TO INITIALIZE BASE VARIABLE I
5   @00000:
6   MVI A, 10
7   LXI H, I
8   CMP M CODE TO COMPARE I AGAINST THE LIMIT 10
9   JC @00001
10  JMP @00002
11  @00003:
12  LDA I
13  ADI 1H CODE TO INCREMENT I BY DEFAULT VALUE OF 1
14  JNC @00000
15  JMP @00001
16  @00002:
17  LXI H, I+1
18  STA I PROGRAMMERS CODE
19  JMP @00003
20  ENDFOR
21  @00001:
```

LOCATION TAG	MEANING
@00000	COMPARE CODE
@00001	EXIT LOCATION
@00002	PROGRAMMERS CODE
@00003	INCREMENT BASE VARIABLE CODE

LOC	CODE DATA	LINE	SOURCE CODE
		22	\$EJECT
		23	FOR I = 1 TO J BY 2 DO
0025 21	4B00 C	24 -	LXI H, I
0028 36	01	25 -	MVI M, 1
		26 -	BASE VARIABLE INITIALIZATION
002A 3A	4C00 C	27 -	LDA J
002D 21	4B00 C	28 -	LXI H, I
0030 BE		29 -	CMP M
0031 DA	4B00 C	30 -	JC @00005
0034 C3	4200 C	31 -	JMP @00006
		32 -	COMPARE BASE VARIABLE (I) TO LIMIT SPECIFIED BY THE VARIABLE J
0037 3A	4B00 C	33 -	LDA I
003A C6	02	34 -	ADI 2
003C D2	2A00 C	35 -	JNC @00004
003F C3	4B00 C	36 -	JMP @00005
		37 -	INCREMENT THE BASE VARIABLE BY THE VALUE 2 WHICH IS SPECIFIED BY THE BY OPTION
0042 21	4D00 C	38	LXI H, I+2
0045 32	4C00 C	39	STA J
0048 C3	3700 C	40 -	JMP @00007
		41	PROGRAMMERS CODE
		42 -	ENDFOR
		43	END
004B		44	I: DS 1
004C	0A	45	J: DB 10
		46	END

MODULE INFORMATION

CODE AREA SIZE = 4DH 77D
 O PROGRAM ERROR(S)

END OF SMABO

Pseudocode For FOR_PROC
Figure 3.7.10

```
FOR_PROC:
  call G_CHAR_CNT to get the column number that the IF symbol
    starts and set the indentation count;
  call FOR_CODE to evaluate the FOR statement;
  if (the next symbol is not DO_symbol) then
    set error field of the SKIP LINE HEADER to 'D' (missing DO);
  copy source line with SLH appended to it to the intermediate file;
  if (the FOR_CODE routine found any errors ) then
    return to PASS1_PROC;

  copy the code generated by FOR_CODE into the intermediate file;
  /* set the stop symbols to ENDFOR, null and null (not used) */
  call PASS1_PROC /* get programmers code that falls into body of
    the FOR statement */

  if (symbol is ENDFOR_symbol) then
    do;

      /* the increment location tag pointer to the generated code
        used to increment the base variable (expression_1) */
      call GEN_CODE to generate a "JMP increment_location"
      copy the ENDFOR line to the intermediate file;
      copy the EXIT location tag that was created by FOR_CODE
        to the intermediate file;

    end;
end FOR_PROC;
```

Pseudo Code For FOR_CODE
Figure 3.7.11

FOR_CODE:

```
/* The code generated by this routine is placed into a code buffer
   that is returned to FOR_PROC
*/

get the first expression /* base variable */
/* the first expression is used as the address of the base
   variable */
if (the expression is not a identifier or a number) then
    then return to FOR_PROC illegal expression;

get the next symbol;
if (symbol is not EQUAL_symbol) then
    return to FOR_PROC illegal symbol;

get the second expression that is either the address of the value
    or the value that the base variable is initialized to;
call GEN_CODE to generate "LXI H,base_variable" instruction;

/* generate the code to initialize the base variable */
if (the second expression is actual value) then
    call GEN_CODE to generate a "MVI A,second_expression";
else
    /* expression is address of value */
    call GEN_CODE to generate a "LDA second_expression" and a
        "MOV M,A";

call JUMP_TAG to create the START location tag;
copy the START location tag into the code buffer after the code
    to initialize the base variable;

if (the next symbol is not a TO_SYM) then
    return to FOR_PROC illegal symbol;

get the third expression that is either the address of the value
    or the actual value which represents the limit that the base
    variable is incremented to;
/* generate the code to compare the base variable to the limit */
if (third_expression is actual value) then
    call GEN_CODE to generate a "MVI A,third_expression";
else
    /* third expression is the address of the value */
    call GEN_CODE to generate a "LDA third_expression;
call GEN_CODE to generate "LXI H,base_variable" and
    "CMP M" instructions.
call JUMP_TAG to create the EXIT location tag that points to the
    end of the FOR statement;
call GEN_CODE to generate a "JC exit_location" if the limit is
```

```

    exceeded by the base variable;
call JUMP_TAG to create the CODE location tag that points to the
    start of the programmers code in the body of the FOR statement;
call GEN_CODE to generate a "JMP code_location". Jump here if the
    limit is not exceeded;

/* generate the code used to increment the base variable */
call JUMP_TAG to create the INCREMENT location tag. This tag
    points the code that increments the base variable;
copy the INCREMENT location tag to the code buffer;
call GEN_CODE to generate a "LDA base_variable". Gets the base
    variable read to be incremented;

if (symbol is BY_symbol) then
    get the forth expression which is the value or the address of
        the value that the base variable is incremented by;
    if (the expression is the actual value) then
        call GEN_CODE to generate a "ADI forth_expression". This
            increments the base variable by the forth_expression;
    else
        /* expression is address of value */
        call GEN_CODE to generate a "LXI H,forth_expression" and a
            "ADD M" to increment the base_variable;
    else
        /* the BY option is not used default increment is one (1) */
        call GEN_CODE to generate a "INR A" instruction;
        call GEN_CODE to generate a "JMP start_location"

    copy the CODE location tag to the code buffer. After this
        instruction starts the programmers code;
end FOR_CODE;

```

The CASE Statement

Many variations of the CASE statement exist. SMA80's implementation uses the value of the `case_expression` (described below), as an address where the number of the case to be executed can be found. This case number is used as an index into a jump table that contains the starting address of the specified case.

Because of its complexity the CASE statement uses more stack space during assembly and more general purpose registers at runtime than any other statement.

The format of the SMA80 CASE statement is,

```
CASE case_expression DO case_blocks ENDCASE
```

or

```
CASE case_expression TO limit_expression
```

```
DO case_blocks ENDCASE,
```

The jump table buffer, where the location tag used to access the table and the location tags for each of the cases is stored, is 186 bytes in length. Each entry in the jump table is 6 bytes long, this is the length of the of location tags created by SMA80. This buffer is within the reentrant routine and uses stack space. At the end of the CASE statement the first entry of the table (the name of the jump table) is used to tag the beginning of the jump table, all other entries are expanded into data storage instruction that point to the individual cases.

CASE JUMP TABLE
Figure 3.7.12

Jump Table Buffer
@00001
@00011
@00020
@00033

The jump table buffer is expanded to,

@00001:
DW @00011 pointer to the first state
DW @00020 " " " second state
DW @00033 " " " third state

This CASE statement contains three cases.

Figure 3.7.13 shows the code generated to evaluate the CASE statement. The pseudo code for CASE_PROC and CASE_CODE are in figures 3.7.14 and 3.7.15 respectively.

LOC	CODE DATA	LINE	SOURCE CODE
1-11 SMA-80 ASSEMBLER			
-80 INVOKED BY: SMAB0 :F1:CASEXP SMA PRINT(:LP:) CODE			
0000 2A	5200	C	1 *TITLE(EXAMPLE OF CASE STATEMENT)
0003 5D			2 CASE LOC1 DO
0004 0E	00		3 LHL D
0006 21	2200	C	4 MOV E, L
0009 19			5 MVI D, 0
000A 19			6 LXI H, @00000
000B 5E			7 DAD D
000C 23			8 DAD D
000D 4E			9 MOV E, M
000E EB			10 INX H
000F E9			11 MOV D, M
			12 XCHG
			13 PCHL
			14 - @00002:
0010 21	5200	C	15 DO
0013 32	5700	C	16 LXI H, LOC1 PROGRAMMERS CODE
0016 C3	2600	C	17 STA LOC2
			18 JMP @00001- JUMP TO EXIT
			19 ENDDO
			20 - @00003:
0019 21	5700	C	21 DO
001C 32	5200	C	22 LXI H, LOC2 PROGRAMMERS CODE
001F C3	2600	C	23 STA LOC1
			24 JMP @00001- JUMP TO EXIT
			25 ENDDO
			26 ENDCASE
0022	1000	C	27 - @00000:
0024	1900	C	28 DW @00002: JUMP TABLE
			29 DW @00003:
			30 - @00001: EXIT LOCATION FOR CASE STATEMENT
CODE TO INDEX INTO JUMP TABLE AND TRANSFER CONTROL TO PROPER CASE			
FIRST CASE			
SECOND CASE			
LOCATION TAG			
MEANING			
@00000 START OF JUMP TABLE			
@00001 EXIT LOCATION FOR CASE STATEMENT			
@00002 START OF FIRST CASE			
@00003 START OF SECOND CASE			

LOC	CODE DATA	LINE	SOURCE CODE
0026 2A	5200	31	\$EJECT
0029 5D		32	CASE LOC1 TO 2 DO
002A 0E	00	33	LHLD LOC1
002C 3F	02	34	MOV E,L
002E 0D		35	MVI D,0
002F DA	5200	36	MVI A,2
0032 21	4E00	37	CMP L
0035 19		38	JC @00005
0036 19		39	LXI H,@00004
0037 5E		40	DAD D
0038 23		41	DAD D
0039 4E		42	MOV E,M
003A ED		43	INX H
003B E9		44	MOV D,M
		45	XCHG
		46	PCHL
		47	@000006:
003C 21	5200	48	DO
003F 32	5700	49	LXI H,LOC1
0042 C3	5200	50	STA LOC2
		51	JMP @00005 JUMP TO EXIT
		52	ENDDO
		53	@000007:
		54	DO
0045 21	5700	55	LXI H,LOC2
0048 32	5200	56	STA LOC1
004B C3	5200	57	JMP @00005 JUMP TO EXIT
		58	ENDDO
		59	ENDCASE
004E	3C00	60	@000004:
0050	4500	61	DW @00006
		62	DW @00007
		63	@000005:
		64	
0052		65	LOC1: DS 5
0057		66	LOC2: DS 1
		67	END

CODE TO INDEX INTO JUMP TABLE AND TRANSFER CONTROL TO THE PROPER CASE.
 NOTICE THAT THE "TO" OPTION IS USED. IF THE CASE EXCEEDS THE VALUE (2) AS SPECIFIED IN THE CASE STATEMENT A JUMP IS MADE TO EXIT.

FIRST CASE

SECOND CASE

MODULE INFORMATION

CODE AREA SIZE = 58H 88D
 O PROGRAM ERROR(S)

END OF SMAR0

Peusdo Code For CASE_PROC
Figure 3.7.14

```
CASE$PROC:
  call G_CHAR_CNT to get the column number that the statement
    starts and set the indentation count;
  call CASE_CODE to evaluate the CASE expression
  if (the next symbol is not DO_symbol) then
    set error field of the SKIP LINE HEADER to 'D' (missing DO);
  copy source line with SLH appended to it to the intermediate file;
  if (the CONDITION routine found any errors ) then
    return to PASS1_PROC;
  copy the code generated by CASE_CODE to the intermediate file;

  while (symbol is not ENDCASE_symbol) do;
    /* set stop symbols to ENDCASE, DO and ENDDO */
    call PASS1_PROC /* get the programmers code that falls within
      the CASE staement */

    if (symbol is DO_symbol) then
      copy the line to the intermediate file;
      call JUMP_TAG to create a location tag for the start of
        this case;
      put the location tag into the jump table;
      copy location tag to intermediate file;

    else if (symbol is ENDDO_symbol) then
      if (ENDDO no associated with DO) then
        copy the line with an illegal statement error
          in the SLH to the intermediate file;
      else
        /* finished getting programmers code that
          falls into this case */
        copy ENDDO line to intermediate file;
        call GEN_CODE to generate a "JMP exit_location".
          This jumps to end of case statement;
      else

        if (symbol is ENDCASE symbol) then
          expand the jump table and copy it
            to the intermediate file;

  end CASE_PROC;
```

Pseudo Code For CASE_CODE
figure 3.8.15

```
CASE_CODE:
  call JUMP_TAG to create the EXIT location tag that points to the
    end of the case statement;
  call JUMP_TAG to create the TABLE location tag that points to the
    jump table;

  get the case expression that is used to index into the jump table;
  /* get the value of the case_expression in the DE register pair */
  if (the first expression is the actual value) then
    call GEN_CODE to generate a "LXI H,case_expression" instruction
  else
    /* expression is address of value */
    call GEN_CODE to generate a "LHLD case_expression".
  call GEN_CODE to generate "MOV E,L" AND "MVI D,0" instructions;

  if (symbol is TO_symbol) then
    get the limit_expression that is used to prevent a non-existing
      case from being accessed;
    /* generate the code to compare the case_expression to the
      limit expression;
    if (the limit expression is the actual value) then
      call GEN_CODE to generate a "MVI A,limit_expression";
    else
      call GEN_CODE to generate a "LDA limit_expression";
      call GEN_CODE to generate "CMP L" and "JC exit_location" ;

  /* generate the code to execute the proper case */
  call GEN_CODE to generate,
    "LXI H,table_location"  start address of jump table,
    "DAD D"                 add the value of the case expression
    "MOV E,M"               put the contents of the index jump
    "INX H"                 table into the DE register pair
    "MOV D,M"
    "XCHG"                  move DE register pair to HL register
    "PCHL"                  transfer control to address in HL register pair
                           which is address of case;

end CASE_CODE;
```

3.8.1.3 Conditional assembly

Conditional assembly is the process of determining if a block of code is to be included in the assembly process. SMA80's implementation of conditional assembly is very similar to the implementation of the IF-THEN-ELSE instruction previously presented.

The format of the conditional assembly statement is,

```
CONDIF <condition> THEN <statement(s)> ENDCOND
```

or

```
CONDIF <condition> then <statement(s)> ELSECOND
```

```
<statement(s)> ENDCOND,
```

where, condition := <expression conditional_operator expression> |
<expression>.

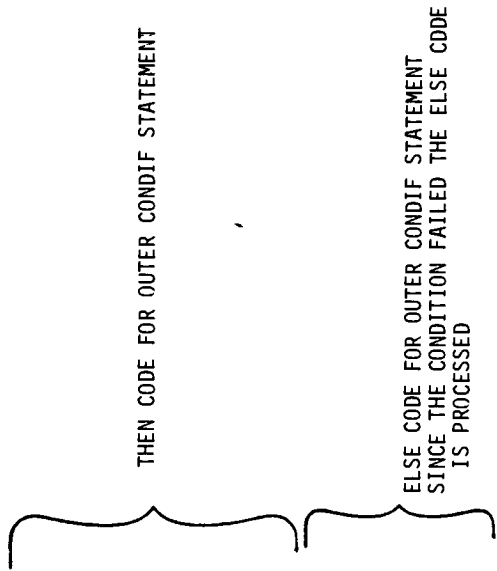
Where the IF and WHILE statements generate code to evaluate the condition at runtime, the CONDIF statement evaluates the condition at assembly time. The condition can only contain numbers, and SET and EQU identifiers. Care must be taken that the identifiers used for conditional assembly are defined before the CONDIF statement they are used in. This is necessary because the identifiers and the CONDIF statement are both processed in the same pass. Code that is not to be included in the assembly process is copied to the intermediate file with an action code of STATEMENT_LINE, in the SLH, causing passes two and three to skip the line.

Conditional assembly statements may be nested up to the point that the stack space runs out. As with the structured statements the terminal symbols of the conditional assembly statement (i.e., ELSECOND and ENDCOND) have no effect other than producing an error if they appear outside a CONDIF block. The CONDIF statement can contain a location tag that will be processed by the second and third passes.

Processing of conditional assembly is performed by the procedures CONDIF_PROC and EVALUATE_CONDITION (figure 3.7.17). The implementation of the CONDIF statement can be found in figure 3.7.16.

Figure 3.7.17

LOC	CODE DATA	LINE	SOURCE CODE
ISIS-II SMA-80 ASSEMBLER SMA-80 INVOKED BY: SMABO F1: CONEXP SMA PRINT(LP.)			
0010		1	\$TITLE(EXAMPLE OF CONDITIONAL ASSEMBLY)
0013		2	COUNT1 SET 16D
		3	COUNT2 SET 19D
		4	CONDIF (COUNT1 = COUNT2) THEN
		5	LXI H,20
		6	CONDIF (COUNT1 = 16D) THEN
		7	LXI H,COUNT2
		8	ELSECOND
		9	CONDIF (COUNT1) THEN
		10	LXI H,10
		11	ELSECOND
		12	LXI H,20
		13	ELSECOND
		14	LXI H,COUNT1
		15	ELSECOND
		16	ENDCOND
		17	CONDIF (COUNT1 < 20H) THEN
		18	LXI H,COUNT1
		19	ELSECOND
		20	STA COUNT1
		21	ENDCOND
		22	LXI H,1010G
		23	STA 101
		24	ENDCOND
		25	END
0000 21	1000		
0003 21	0802		
0006 32	6500		
MODULE INFORMATION CODE AREA SIZE = 9H 9D 0 PROGRAM ERROR(S)			
END OF SMABO			



Pseudo Code For CONDIF_PROC and CONDIF_SCAN
Figure 3.7.17

CONDIF_SCAN:

```
/* This procedure is called by CONDIF_PROC and copies the source
   that is not to be processed into the intermediate file.  Nested
   CONDIF statement are kept track of to prevent premature return
   to CONDIF_PROC.  This is accomplished by recursively calling
   itself.
   This procedure is passes an optional stop symbol that is the
   ELSECOND symbol if we should return for else processing
*/
```

```
while (not at end of source code) do;
  if (symbol is CONDIF_symbol) then
    do;
      copy line to intermediate file with SLH;
      /* this is a recursive call to keep track of the
         ENDCOND statements. The optional stop symbol is set
         to null (not used). */
      call CONDIF_SCAN
    end;
  else

    if (symbol is ENDCOND_symbol) then
      return /* if this was a recursive call it will return
              to CONDIF_SCAN otherwise it will retrun to
              CONDIF_PROC. */
    else

      if (symbol is optional stop symbol) then
        return;

  copy the line to the intermediate file with a SLH indicating that
  the second and third passes should skip this line;

end CONDIF_SCAN;
```

CONDIF_PROC:

```
/* This procedure returns a true if the condition is true and a
   false if the condition is false.  A error is also returned.*/
call EVALUATE_CONDITION;

if (condition is false) then
  /* what to skip the code up to the ELSECOND_symbol if one is
     used or up to the ENDCOND */
  call CONDIF_SCAN /* set optional stop symbol to ELSECOND */
else
  /* process the code condition is true */
  /* set stop symbols to ENDCOND, ELSECOND and null */
```



```

    call PASS1_COND /* process code until stop symbol */

if (symbol is ELSECOND_symbol) then
    copy line to intermendiate file with SLH;
if (condition status is true) then
    /* since the condition is true the cond THEN code was
       processed so skip the else code */
    call CONDIF_SCAN /* set optional stop symbol to null*/
else
    /* the condition is false so process code */
    copy line to intermediate file;
    /*set stop symbols to ENDCOND_symbol, null and null */
    call PASS1_COND /* return when ENDCOND found */

end CONDIF_PROC;

```

Pseudo Code For EVALUATE_CONDITION
Figure 3.7.17

```
EVALUATE_CONDITION:
  call EXPRESSION to get the value of the first expression;
  if (error) then
    return to CONDIF_PROC with error;

  if (symbol is not a conditional operator) then
    /* must be single expression condition check for true */
    if (value is true) then
      return true to CONDIF_PROC;
    else
      return false to CONDIF_PROC;
  else
    do;
      save conditional operator;
      call EXPRESSION to get second expression;
      if (error) then
        return false to CONDIF_PROC with error;
      use conditional operator to compare the two values;
      return a true if the condition if the true otherwise
        return a false;
    end;
end CONDIF_SCAN;
```

3.7.1.4 Macro Processor

A macro processor is a powerful tool found in most assembly languages and some high level languages. While all macro processors share the same end result, text substitution, the implementations and available features vary.

SMA80's macro processor is implemented by four routines, `MACRO_INIT`, `MACRO_DEFINE`, `MACRO_EXPAND` and `GET_MACRO_LINE`. An explanation of these routines follows. Figure 3.7.21 contains the pseudo code.

`MACRO_INIT`

As previously mentioned the symbol table buffer is shared by the symbol table and the macro definition buffer during the first pass. This routine calculates the starting address of the macro definition buffer and adjusts the maximum number of symbols that can appear in the symbol table.

`MACRO_DEFINE`

This routine fills the MACRO DEFINITION RECORD (figure 3.7.18) with the formal parameters, local symbols and macro body. It creates or updates the symbol table to reflect the fact that the identifier

is a macro, and sets the value field to the starting location of the macro definition record.

MACRO_EXPAND

The actual parameters and the generated local symbols, if any, are placed on the ACTUAL PARAMETER STACK (figure 3.7.19). This stack starts at the end of the symbol table buffer and works up toward the macro definition buffer. The nested macro level count is incremented and used as an index into the MACRO CONTROL BLOCK (figure 3.7.20). Here the addresses of the macro definition buffer and formal parameter stack, and the number of bytes of actual parameter stack used by this macro are stored. To inform GET_LINE that the next line of text is to be obtained from the macro definition buffer, the ACTIVE_MACRO_FLAG is set on.

GET_MACRO_LINE

When the ACTIVE_MACRO_FLAG is on, GET_LINE calls this routine instead of getting the next line of source code from an input file. GET_MACRO_LINE fetches the next line of source code from the macro definition buffer associated with the macro. Each identifier in the line is compared to the formal parameter list associated with the macro. If a match is not found, and the macro is not at the outer level, the formal parameter lists of the macros at the outer levels are searched for a match. If a match is found, the formal parameter

in the source line is substituted with the associated actual parameter. If an end of a macro definition is encountered the ACTUAL PARAMETER STACK used by the macro is released. If the macro is not at the outer level the line is obtained from the macro at the next level. When the end of the macro at the outer level is encountered, GET_MACRO_LINE returns a FALSE to GET_LINE indicating that macro expansion has completed.

MACRO DEFINITION BUFFER
Figure 3.7.18

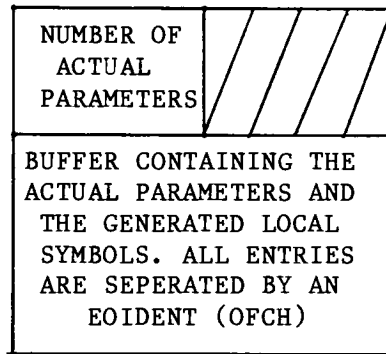
MACRO DEFINITION RECORD

POINTER TO MACRO BODY	
NUMBER OF PARAMETERS	NUMBER OF LOCAL SYM.
BUFFER CONTAINING THE FORMAL PARAMETERS AND THE NAMES OF LOCAL SYMBOLS. ALL ENTRIES ARE SEPERATED BY AN EOIDENT (OFCH)	
MACRO BODY BUFFER. THIS BUFFER CONTAINS THE BODY OF THE MACRO AND IS ENDED WITH AN EOMACRO (OFDH).	

A MACRO DEFINITION RECORD exist for each macro defined. These records are placed into the MACRO DEFINITION BUFFER which starts two thirds into the symbol table buffer.

ACTUAL PARAMETER STACK
Figure 3.7.19

ACTUAL PARAMETER RECORD



One ACTUAL PARAMETER RECORD exists for each active macro. An active macro being any macro currently being expanded. There is a limit of ten (10) active macros. ACTIVE PARAMETER RECORD'S are pushed on to the ACTUAL PARAMETER STACK which starts at the end of the symbol buffer.

MACRO CONTROL BLOCK
Figure 3.7.20

MACRO CONTROL BLOCK RECORD

POINTER TO MACRO DEFINITION RECORD
COUNT
POINTER TO ACTUAL PARAMETER RECORD
LENGTH OF ACTUAL PARAMETER RECORD

COUNT - This counter is used when expanding the macro. It keeps track of the current position in the macro definition record.
ACTUAL PARAMETER RECORD LENGTH - Used when releasing the portion of the ACTUAL PARAMETER STACK used by the macro during macro expansion.

A MACRO CONTROL BLOCK RECORD exists for each macro currently being expanded. It is indexed by the nested macro level count.

Pseudo Code For SMA80's Macro Processor
Figure 3.7.21

MACRO_INIT:

```
    calculate the start of the macro definition table;
    set macro level to 0;
    set active$macro$flag = off;
end MACRO_INIT;
```

MACRO_DEFINE:

```
    if (there is an entry in the symbol table for this identifier and
        it was not previously defined as being a macro) then
        do;
            write line to intermediate file with multiply defined error
                in SLH;
            return to calling routine;
        end;

    read the formal parameters into the formal parameter buffer;
    do while (not ENDMACRO_symbol that is associated with outer macro)
        get the next line of source code and first symbol on the line;
        if (symbol is LOCAL_symbol) then
            if (before first line of macro body) then
                read local symbols into format parameter buffer;
            else set error = illegal local symbol definition;
        else
            if (symbol is MACRO_symbol associated with a nested macro
                definition) then
                increment nested macro count;
            else
                if (symbol is ENDMACRO_symbol) then
                    if (nested macro count is not 0) then
                        decrement nested macro count;
                write line to intermediate file;
            end while;

        if (macro has previously been defined) then
            call SYM_UPDATE to update symbol table with new macro
                definition table pointer in value field;
        else
            call SYM_ENTRY to make a new entry in the symbol table;

        calculate the next available macro definition table location to be
            used for the next macro definition;
    end MACRO_DEFINE;
```

MACRO_EXPAND:

```
    /* this routine is passed the pointer into the macro definition
        table where the definition for this macro can be found
```

```

*/
if (nested macro level exceeded) then
    copy line to intermediate file with level exceeded error "N";
else
    increment nested level;
    with the macro$control$block for the current macro level set
        the buffer pointer to the macro definition buffer pointer,
        the buffer count to 0;
        and the actual parameter pointer to the address of the first
        location available on the actual parameter stack;
    read the actual parameters into the actual parameter stack;
    create the local symbols for this macro expansion and place
        them into the actual parameter stack;
    set active_macro_flag on to indicate to GET_LINE that the
        following lines of source code are obtained by a call to
        GET_MACRO_LINE;
end MACRO_EXPAND;

```

GET_MACRO_LINE:

```

    fill LINE_BUFFER with the next line from the macro definition
    buffer;
    if (end of macro definition buffer) then
        release the actual_parameter_stack used by this macro;
        decrement the nested macro count;
        if (nested macro count is zero) Then
            set active$macro$flag off;
            return a false to GETLINE indicating a line is not
            available;
        else
            fill line buffer at new level and check as was done above;

    check all identifiers against the formal parameter list. If a
    a match is found substitute the formal parameter for the
    actual parameter; /* if the macro is nested the search starts
    at the parameter lists of the current macro level and
    work down toward the outer level */
    return a value of true to GETLINE indicating that a line is
    available for processing;
end GET_MACRO_LINE;

```

3.7.2 The Second Pass

The main function of the second pass of SMA80 is to define the first level of an identifier's value. If an identifier is defined by a second identifier that is located further on in the source code the first identifier can not be defined until the value of the second identifier is known. This is why the values of identifiers are calculated in both the second and third passes. As figure 3.7.22 shows it is possible that the definition of some identifiers may not be resolved by the two passes, or for that matter by any number of passes. Identifiers of this type produce erroneous values when used.

UNRESOLVABLE IDENTIFIERS

Figure 3.7.22

A	EQU	B
B	EQU	C
C	EQU	B

The identifier A, b and C cannot be defined no matter how passes over the source code is made.

Once the PASS2 segment has been loaded into memory, control is transferred to the routine MAIN\$PASS2, where the symbol table, location counters and line buffer are initialized. Upon completion of the initialization the routine PASS2_PROC called, this is where the processing of the source code is performed. The source code

processed by the second pass comes from the intermediate file created by the first pass.

PASS2_PROC determines the action to be taken for a particular line. If a SKIP LINE HEADER is encountered the action code (the second byte of the SLH) is interrogated (section 3.7.1) to determine the action. The procedures MEMORY_PROC, DATA_PROC, EXTRN_PROC, PUBLIC_PROC and IDENT_PROC are employed by PASS2_PROC to define the value of identifiers. CNTL_PROC (section 3.2) is used to process the TITLE control which is needed at the beginning of the third pass.

PASS2 does not process errors. If an error is encountered PASS2 takes no action. The error will be found and reported by the third pass.

MEMORY_PROC

Since there are effectively three memories at assembly time, CODE, DATA and ABSOLUTE, three location counters must be maintained. These three memories are made absolute by the linker/loader for runtime execution. When an CSEG, DSEG or ASEG instruction is encountered, the procedure MEMORY_PROC is called, where the previous location counter and memory/segment type is saved and the new location counter and memory/segment type associated with the instruction is used. The ORG instruction, which also causes entry to

MEMORY_PROC, sets the current location counter to the value of the instruction operand field.

DATA_PROC

DATA_PROC is entered to process DS, DB and DW data storage instruction. The current location counter is incremented by the amount of storage required by the operand field of the instructions.

EXTRN_PROC

The EXTRN_PROC procedure enters the identifiers defined by the EXTRN instruction into the symbol table. The status field of the symbol (figure 3.5.1) is set to EXTERNAL_SEG and the value is set to a unique number called EXTERNAL NUMBER. This number is used to identify the external when creating object code for expressions that use the external identifier.

If an identifier specified in the EXTRN instruction is already locally defined, it is not entered into the symbol table. PASS3 will pick up the error.

PUBLIC_PROC

PUBLIC_PROC enters or updates symbols in the symbol table depending if they are already defined, to reflect that they are declared public. A public symbol is referenced as an external

outside of the module it is created in. Bit 7 (PUBLIC_DEF) of the symbols status is set to define an identifier public.

IDENT_PROC

IDENT_PROC enters LOCATION TAG, EQU and SET identifiers into the symbol. SET identifiers are also updated by this procedure. The pseudo code in figure 3.7.23 describes the implementation of this procedure.

Pseudo Code For IDENT_PROC
Figure 3.7.23

```
IDENT_PROC:
  /* this is a reentrant procedure it may be recursively called if
    multiple location tags appear on the same line
  */

  save the name of the identifier in the symbol table for later use;
  call SYM_LOOKUP to see if symbol exists;

  if (symbol exist and it is defined external) then
    /* can not define an external identifier */
    return to calling routine;

  get the next symbol
  if (symbol is a COLON_symbol) then
    do;
      if (this line does not have a SLH with an action code of
        get the location tag only) then
        call PASS2_PROC to process rest of line;
      /* it is necessary to look the symbol up again because
        other sym_lookup calls may have been made when
        the rest of the line was processed by the call to
        PASS2_PROC */
      call SYM_LOOKUP for saved symbol
      if (symbols status indicates bit 2(pass2_def) is set) then
        /* symbol can not be multiply defined used the first
          definition */
        return
      else
        do;
          set symbol table fields to,
            symbol type = VAR_symbol,
            status = current segment type and pass2_def;
          value = location counter;
        end;
      end;
    else
      if (symbol is EQU_symbol) then
        if (symbol is not defined or pass2_def is not set in status if
          symbol is defined) then
          do;
            call EXPRESSION to get the value of the operand
              field of the EQU instruction;
            set symbol table fields to,
              symbol type = EQU_symbol;
              status = current segment type and pass2_def;
              value = value of expression (above);
          end;
        end;
```

```

    else
        return to calling routine can not multiply define EQU ident;
    else
if (symbol is SET_symbol) then
    /* SET identifier can be redefined during assembly */
    if (symbol is undefined) then
        do;
            set symbol table fields to,
                symbol type = SET_symbol
                status = ABSOLUTE_SEG;
                value = value of expression in the operand
                    field;
        end;
    else
        /* symbol is defined make sure it was previously defined as
           a SET identifier */
        if (previously defined as identifier) then
            do;
                set symbol table value field to value of expression in
                    operand field;
            end;
        else
            return to calling routine can redefine symbol that was
                not defined other than SET identifier;
    if (the symbol exist)then
        call SYM$UPDATE to update existing symbol;
    else
        call SYM$ENTRY to enter new symbol;
end IDENT_PROC;

```


3.7.3 The Third Pass

PASS3 of SMA80 is responsible for,

1. defining the last level of identifier values,
2. generating object, and
3. generating an expanded source code listing.

Once the PASS3 overlay is loaded into memory the routine MAIN_PASS3 initializes the, listing and object files, location counters, and line buffer. After initialization, control is passed to PASS3_PROC which processes the source code.

PASS3's local procedures that are responsible for defining identifiers and maintaining the location counters are MEMORY_PROC, DATA_PROC, EXTRN_PROC, PUBLIC_PROC and IDENT_PROC. These procedures are very similar to their PASS2 counterparts, except for:

1. MEMORY_PROC -
 - a. The object code routine NEW_SEGMENT is called to clean up after the previous memory segment and to setup for the new segment.
 - b. Return any errors that were found so that the error can be included in listing and the error count can be incremented.

2. DATA_PROC -

a. The object code routine OBJECT_SPACE is called to leave the space specified by the DS instruction in the object code.

b. Rather than just calculating the space requirements for the DB and DW instructions, the values of the operand fields are calculated and included in the object code. Operand fields of the DW instruction require one word (16-bits) of object code. The values are stored in memory with the high order byte followed by the low order byte.

c. Return any errors that were found.

3. EXTRN_PROC -

a. Call the routine DCL_EXTRN_SYM to allow the object code generator to set up for the external identifier.

b. return any errors.

4. PUBLIC_PROC -

a. Check to make sure the identifier declared external were defined during the second pass. If they were not an undefined (U) error is returned.

5. IDENT_PROC -

a. As public identifiers are encountered the routine DCL_PUBLIC_SYM is called so that the object code generator can record its address. The address is used to resolve external references to this identifier during linking.

a. As identifiers are defined, bit 4 (PASS3_DEFINED) is set to prevent multiple definitions of the identifier.

b. Errors generated by multiply defined identifiers, expressions, and undefined identifiers are returned to the calling routine.

3.7.3.1 Object Code For The Instructions

Where the second pass was only concerned with the length of the object code generated by the assembly language instructions, PASS3 must actually generate the object code for the instructions. To do this the 8085 instruction set is divided into twelve (12) formats based upon the object code that is generated from them.

The notation used to describe the instruction formats includes,

single_register := A | B | C | D | E | H | L | M,

register_pair := B | D | H | SP | PSW .

Registers used in single register instructions are encoded in either in the source field (bits 0-2) or destination field (bits 3-5) of the object code. The twelve instruction format are,

1. INSTRUCTION TYPE 0 := RST <n>

Where n has a value of 0-3 and is encoded in the destination field of the object code.

Length: 1 byte.

2. INSTRUCTION TYPE 1 := <opcode>

Length: 1 byte.

3. INSTRUCTION TYPE 2 := <opcode> <single_register>

The expanded listing contains not only the source code but also,

1. the title on the top of the each page, if a title was specified,
2. the runstring that was used to invoke SMA80,
3. the location counter, object code and data associated with each instruction,
4. the line number of each line,
5. the one character error code describing the error, if any,
6. the code generated by the structured statement with a '-' character to the right of the line number if the COND control was used,
7. the code generated by macro expansion with a '+' to the right of the line number if EXPMACRO control was used,
8. the final contents of the location counters that were used, and
9. an error report that includes the number of errors and the which structured statements that are missing the appropriate END symbols, if any.

If a source line produces an error the line is always printed as long as the NOPRINT control is not used. The line will be printed even if it is within a NOLIST block or if it was generated by a structured statement or macro expansion and the CODE and EXPMACRO controls were not used.

Examples of SMA80 listings have be placed through the report.

Figure 3.7.24 displays how certain errors are reported.

MABO STRUCTURED MACRO ASSEMBLER PRINT TEST #1 _____ TITLE

LOC CODE DATA LINE SOURCE CODE

S-II SMA-80 ASSEMBLER

INVOKED BY: SMABO :F1:PRISI1.SMA PRINT(.LP:.) RUNSTRING

```

1 $TITLE(PRINT TEST #1)
2
3 IF (COUNT1 NE OFEH) THEN
4 LDA COUNT1
5 JZ @00000 } THESE TWO LINE OF CODE WERE GENERATED BY SMA80 AND ARE LISTED BECAUSE
6 CASE 100H TO 2 DO AN ERROR WAS ENCOUNTERED WHEN PROCESSING THE LINE.
7 DO
23 LXI H,COUNT2
24 ENDDO
26 DO
28 LXI H,20
29 ENDDO
31 ENDCASE
32 COUNT2: DB 0
37 MISSING ENDIF FOR THE IF STATEMENT
38
F 002E 00
002E 00

```

STATEMENT ERRORS

1 ENDIF(S) MISSING

} TELLS US THAT ENDIF IS MISSING

MODULE INFORMATION

CODE AREA SIZE = 2FH 47D } DEFAULT SEGMENT IS CODE SEGMENT. NUMBER OF BYTES USED IS REPORTED.
3 PROGRAM ERROR(S)

END OF SMABO

ERRORS

THE FIRST "U" INDICATES THAT COUNT1 WAS NOT DEFINED
THE SECOND "U" INDICATES THAT @00000 WAS NOT DEFINED BECAUSE THE ENDIF STATEMENT WAS NOT ENCOUNTERED
THE "F" INDICATES THAT THE SOURCE CODE WAS NOT ENDED BY THE END INSTRUCTION

CHAPTER 4
FUTURE ENHANCEMENTS
AND
CONCLUSION

The SMA80 project was undertaken not only for the challenge of designing such a system, but also to create a working system that will be used by others. This paper has described the first version of the Structured Macro Assembler, SMA80. While deciding on the features to be implemented in the first version and those to be implemented in future revisions it was important to keep in mind that SMA80 is an assembler and not a compiler. SMA80 does not attempt to take the place of high level languages, but instead make available some high level features when assembly language is the only alternative. Being an assembler, SMA80 must in no way restrict the programmer from utilizing the machine to its fullest.

Following is a description of some future enhancements planned for the next revision of SMA80.

Macro Processor

The current implementation of the macro processor is limited by the macro definition buffer, which shares the symbol table buffer with the symbol table during the first pass of SMA80. The symbol table buffer, which is 12288 bytes, is divided up so that 8092 bytes is used for the symbol table (room for 449 symbols), and 4096 bytes is used for the macro definition buffer. The next version of the SMA80 will still use the shared symbol table buffer, except that when the symbol table grows to the point that it collides with the macro definition buffer, or the macro definition buffer is exceeded, the macro definitions that reside in memory and all subsequent macro definitions will be written out to a disk file. This implementation avoids the relatively slow disk accesses before it becomes necessary, and at the same time removes the restriction on the size of macro definitions.

Symbol Table

As presented in section 3.5 the Search Tree technique is the implementation used by SMA80 for managing the symbol table. The disadvantage to this technique is the effect unbalanced symbol entries have on the search. To resolve this problem the symbol table will be balanced. The method most likely to be used was developed by Adelson-Velskii and Landis in 1962. The structure produced by their algorithm is called the AVL-tree. A Balance-Factor is used to

indicate whether or not the tree is balanced. Once it has been determined the tree is unbalanced the nodes of the tree are rotated to achieve balance.

Conditional Expressions

At present SMA80 is only capable of generating code for conditional expressions containing one conditional operator. An enhancement will be made so that arbitrary conditional expressions with multiple conditional and logical operators can be processed. For example,

IF (CNT1 = 23H) OR (CNT1 = 24H) THEN.

S-II PL/M-80 V3.1 COMPILATION OF MODULE INCFILE

ECT MODULE PLACED IN : F4:ELD.OBJ

PILER INVOKED BY: PLM80 ELD.PLM DATE(27-MAR-82) OBJECT(:F4:ELD.OBJ) PRINT(:F1:ELD.LST) PAGELENGTH(50)

```

1      *TITLE('STURCTURED MACRO ASSEMBLER INCLUDE FILES')
      INCFILE:
      DO,

```

```

      *INCLUDE(:F1:INC.ELD)
      /*          :F1:INC.ELD          08.02.80  */
      =
      =
      2      1 = DECLARE POINTER          LITERALLY 'ADDRESS',
      3      1 = DECLARE WORD             LITERALLY 'ADDRESS',
      4      1 = DECLARE INTEGER          LITERALLY '(2) BYTE',
      5      1 = DECLARE REAL             LITERALLY '(4) BYTE',
      6      1 = DECLARE TRUE             LITERALLY 'OFFH',
      7      1 = DECLARE FALSE            LITERALLY 'OOOH',
      8      1 = DECLARE BOOLEAN          LITERALLY 'BYTE',
      9      1 = DECLARE FOREVER          LITERALLY 'WHILE 1',
      10     1 = DECLARE ON                LITERALLY 'OFFH',
      11     1 = DECLARE OFF               LITERALLY 'OOOH',
      12     1 = DECLARE NULL              LITERALLY 'OOOH',
      13     1 = DECLARE ALL               LITERALLY 'OFFH',
      14     1 = DECLARE LF                LITERALLY 'OAH',
      15     1 = DECLARE FF                LITERALLY 'OCH',
      16     1 = DECLARE CR                LITERALLY 'ODH',
      17     1 = DECLARE CRLF              LITERALLY 'ODH,OAH',
      18     1 = DECLARE BYTE$LNG          LITERALLY 'OO1H',
      19     1 = DECLARE WORD$LNG          LITERALLY 'OO2H',
      20     1 = DECLARE EOFILE             LITERALLY 'OFFH',
      21     1 = DECLARE EOMACRO            LITERALLY 'OFDH',
      22     1 = DECLARE EOIDENT            LITERALLY 'OFCH',
      23     1 = DECLARE EOBLOCK            LITERALLY 'OFBH',
      =

```

```

$EJECT
$INCLUDE(:F1:SYSTEM.ELD)
/*
  SYSTEM.ELD      10-AUG-80
*/
/*****
/* SYMBOL TABLE SEARCH DESCRIPTORS */
    DECLARE SYM$UNDEF          LITERALLY '1';
    DECLARE SYM$ENTERED        LITERALLY '2';
    DECLARE SYM$OVERFLOW        LITERALLY '3';
    DECLARE SYM$EXSIST          LITERALLY '4';
    DECLARE SYM$UPDATED         LITERALLY '5';
/*****
/* SIZE DESCRIPTORS */
    DECLARE STACK$LENGTH        LITERALLY '2048';
    DECLARE TABLE$SIZE         LITERALLY '12288';
    DECLARE NO$OF$SYM            LITERALLY '646';
    DECLARE LINE$SIZE           LITERALLY '102';
    DECLARE LINE$PLUS            LITERALLY '107';
    DECLARE TITLE$LENGTH        LITERALLY '35';
    DECLARE SYM$LENGTH          LITERALLY '10';
    DECLARE INST$LENGTH         LITERALLY '4';
    DECLARE NUMBER$LNG          LITERALLY '17';
    DECLARE PAGE$LENGTH         LITERALLY '59';
/*****
/* SKIP FIELD TYPES */
    DECLARE SKIP$STATEMENT      LITERALLY 'OFEH';
    DECLARE STATEMENT$LINE      LITERALLY '1';
    /* macro line is a special case for a macro line may consist of
       of statement. Bit 7 if flagged for a macro line */
    DECLARE NOT$MACRO$LINE      LITERALLY '0111111B';
    DECLARE MACRO$LINE          LITERALLY '10000000B';
    DECLARE STATEMENT$CODE      LITERALLY '3';
    DECLARE TAG$LINE            LITERALLY '4';
/*****
/* MASKS FOR CODE AND DATA */

```

```

15 1 = DECLARE BYTE$TYPE          LITERALLY '00000000B';
16 1 = DECLARE WORD$TYPE         LITERALLY '10000000B';
17 1 = DECLARE INFO$TYPE         LITERALLY '10000000B';
18 1 = DECLARE DATA$FLD$LNQ     LITERALLY '00000011B';

```

/*****

/* IDENTIFIER DESCRIPTORS */

```

49 1 = DECLARE IDENT$SYM          LITERALLY '1';

```

```

50 1 = DECLARE VAR$SYM            LITERALLY '42';
51 1 = DECLARE EQU$SYM           LITERALLY '43';
52 1 = DECLARE SET$SYM           LITERALLY '44';
53 1 = DECLARE MACRO$SYM         LITERALLY '60';

```

```

54 1 = DECLARE CODE$SEQ          LITERALLY '1';
55 1 = DECLARE DATA$SEQ         LITERALLY '2';
56 1 = DECLARE ABSOLUTE$SEQ     LITERALLY '3';
57 1 = DECLARE EXTERNAL$SEQ    LITERALLY '4';
58 1 = DECLARE NULL$SEQ         LITERALLY '5';

```

```

59 1 = DECLARE PUBLIC$DEF        LITERALLY '10000000B';
60 1 = DECLARE EXTRN$DEF        LITERALLY '01000000B';
61 1 = DECLARE PASS2$DEF        LITERALLY '00100000B';
62 1 = DECLARE PASS3$DEF        LITERALLY '00010000B';
63 1 = DECLARE SEQ$DEF          LITERALLY '00000111B';

```

/* NUMERIC DESCRIPTOR */

```

64 1 = DECLARE NUMBER$SYM        LITERALLY '2';

```

/* INSTRUCTION DESCRIPTORS */

```

65 1 = DECLARE INSTRUCTION$SYM   LITERALLY '3';

66 1 = DECLARE INSTRUCTION$0     LITERALLY '0';
67 1 = DECLARE INSTRUCTION$1     LITERALLY '1';
68 1 = DECLARE INSTRUCTION$2     LITERALLY '2';
69 1 = DECLARE INSTRUCTION$3     LITERALLY '3';
70 1 = DECLARE INSTRUCTION$4     LITERALLY '4';
71 1 = DECLARE INSTRUCTION$5     LITERALLY '5';
72 1 = DECLARE INSTRUCTION$6     LITERALLY '6';
73 1 = DECLARE INSTRUCTION$7     LITERALLY '7';
74 1 = DECLARE INSTRUCTION$8     LITERALLY '8';

```

```

'5 1 = DECLARE INSTRUCTION$9 LITERALLY '9';
'6 1 = DECLARE INSTRUCTION$10 LITERALLY '10';
'7 1 = DECLARE INSTRUCTION$11 LITERALLY '11';
'8 1 = DECLARE INSTRUCTION$12 LITERALLY '12';
=
= /* REGISTER DESCRIPTORS */
=
'9 1 = DECLARE REG$SYM LITERALLY '4';
=
30 1 = DECLARE PSW$REG LITERALLY '1';
31 1 = DECLARE SP$REG LITERALLY '2';
32 1 = DECLARE AS$REG LITERALLY '3';
33 1 = DECLARE BS$REG LITERALLY '4';
34 1 = DECLARE CS$REG LITERALLY '5';
35 1 = DECLARE DS$REG LITERALLY '6';
36 1 = DECLARE ES$REG LITERALLY '7';
37 1 = DECLARE HS$REG LITERALLY '8';
38 1 = DECLARE LS$REG LITERALLY '9';
39 1 = DECLARE MS$REG LITERALLY '10';
=
90 1 = DECLARE DEST$REG LITERALLY '00111000B';
91 1 = DECLARE SRC$REG LITERALLY '0000111B';
92 1 = DECLARE R$ERROR LITERALLY '10000000B';
93 1 = DECLARE RP$ERROR LITERALLY '10000000B';
94 1 = DECLARE RP$1$REG LITERALLY '00110000B';
95 1 = DECLARE RP$2$REG LITERALLY '00010000B';
=
= /* STATEMENT INSTRUCTION DESCRIPTORS */
=
96 1 = DECLARE STATEMENT$SYM LITERALLY '5';
=
97 1 = DECLARE CONDIF$SYM LITERALLY '1';
98 1 = DECLARE WHILE$SYM LITERALLY '2';
99 1 = DECLARE FOR$SYM LITERALLY '3';
100 1 = DECLARE CASE$SYM LITERALLY '4';
101 1 = DECLARE IF$SYM LITERALLY '5';
=
= /* MEMORY DESCRIPTORS */
=
102 1 = DECLARE EXTRN$SYM LITERALLY '6';
103 1 = DECLARE PUBLIC$SYM LITERALLY '7';
=
104 1 = DECLARE MEMORY$SYM LITERALLY '8';
=
105 1 = DECLARE DSEG$SYM LITERALLY '0';
106 1 = DECLARE CSEG$SYM LITERALLY '1';

```

```

107 1      =      DECLARE ASEG$SYM      LITERALLY '2';
108 1      =      DECLARE ORG$SYM      LITERALLY '3';
109      =      =     
110      =      /* DATA STORAGE DESCRIPTORS */
111      =      DECLARE DATA$STORAGE$SYM      LITERALLY '9';
112      =     
113      =      DECLARE DS$SYM      LITERALLY '0';
114      =      DECLARE DB$SYM      LITERALLY '1';
115      =      DECLARE DW$SYM      LITERALLY '2';
116      =     
117      =      /* CONTROL DESCRIPTORS */
118      =      DECLARE CONTROL$SYM      LITERALLY '10';
119      =     
120      =      DECLARE EJECT$SYM      LITERALLY '1';
121      =      DECLARE NOLIST$SYM      LITERALLY '2';
122      =      DECLARE LIST$SYM      LITERALLY '3';
123      =      DECLARE NOOBJECT$SYM      LITERALLY '4';
124      =      DECLARE NOPRINT$SYM      LITERALLY '5';
125      =      DECLARE PRINT$SYM      LITERALLY '6';
126      =      DECLARE TITLE$SYM      LITERALLY '7';
127      =      DECLARE DEBUG$SYM      LITERALLY '8';
128      =      DECLARE INCLUDE$SYM      LITERALLY '9';
129      =      DECLARE EXPAND$MACRO$SYM      LITERALLY '10';
130      =      DECLARE STATEMENT$CODE$SYM      LITERALLY '11';
131      =     
132      =      /* LOGICAL SYMBOLS */
133      =     
134      =      DECLARE AND$SYM      LITERALLY '11';
135      =      DECLARE OR$SYM      LITERALLY '12';
136      =      DECLARE XOR$SYM      LITERALLY '13';
137      =      DECLARE NOT$SYM      LITERALLY '14';
138      =     
139      =      /* CONDITIONAL SYMBOLS */
140      =      DECLARE CONDITION$SYM      LITERALLY '15';
141      =     
142      =      DECLARE EQ$SYM      LITERALLY '51';
143      =      DECLARE NE$SYM      LITERALLY '52';
144      =      DECLARE GT$SYM      LITERALLY '53';
145      =      DECLARE LT$SYM      LITERALLY '54';
146      =      DECLARE LE$SYM      LITERALLY '55';
147      =      DECLARE GE$SYM      LITERALLY '56';

```



```

=
=
=
/* SPECIAL SYMBOLS */
=
136 1 = DECLARE COMMENT$SYM LITERALLY '21'
137 1 = DECLARE COLON$SYM LITERALLY '22'
138 1 = DECLARE PLUS$SYM LITERALLY '23'
139 1 = DECLARE MINUS$SYM LITERALLY '24'
140 1 = DECLARE MULTI$SYM LITERALLY '25'
141 1 = DECLARE DIVI$SYM LITERALLY '26'
142 1 = DECLARE MOD$SYM LITERALLY '27'
143 1 = DECLARE QUOTE$SYM LITERALLY '29'
144 1 = DECLARE COMMA$SYM LITERALLY '30'
145 1 = DECLARE LOCATION$CNT$SYM LITERALLY '31'
146 1 = DECLARE LPAREN$SYM LITERALLY '32'
147 1 = DECLARE RPAREN$SYM LITERALLY '33'
148 1 = DECLARE EOF$SYM LITERALLY '34'
149 1 = DECLARE EOL$SYM LITERALLY '35'
150 1 = DECLARE END$SYM LITERALLY '36'
151 1 = DECLARE ENDIF$SYM LITERALLY '46'
152 1 = DECLARE ENDWHILE$SYM LITERALLY '47'
153 1 = DECLARE ENDCASE$SYM LITERALLY '48'
154 1 = DECLARE ENDFOR$SYM LITERALLY '49'
155 1 = DECLARE ENDDO$SYM LITERALLY '58'
156 1 = DECLARE DO$SYM LITERALLY '37'
157 1 = DECLARE TO$SYM LITERALLY '16'
158 1 = DECLARE THEN$SYM LITERALLY '38'
159 1 = DECLARE ELSE$SYM LITERALLY '39'
160 1 = DECLARE SKIP$LINE$SYM LITERALLY '50'
161 1 = DECLARE LINE$TYPE$SYM LITERALLY '41'
162 1 = DECLARE PERIOD$SYM LITERALLY '45'
163 1 = DECLARE BY$SYM LITERALLY '57'
164 1 = DECLARE LOCAL$SYM LITERALLY '61'
165 1 = DECLARE DEPTH$SYM LITERALLY '62'
166 1 = DECLARE ENDMACRO$SYM LITERALLY '63'
167 1 = DECLARE ELSECOND$SYM LITERALLY '64'
168 1 = DECLARE ENDCOND$SYM LITERALLY '65'

/* NONEXISTENT SYMBOL TO RESET SYM$TYPE */
169 1 = DECLARE SYM$TYPE$RESET LITERALLY '100'

```

```

$EJECT
$INCLUDE(:F1:ASCII.EPD)
/* ASCII.ELD
=
=
=
70 1 = DECLARE ASCII$STATUS
= ERROR
= ACT$LNQ
=
=
71 1 = G$ASCII:
= PROCEDURE (VALUE$PTR, LENGTH, DATA$LNQ) BYTE EXTERNAL;
72 2 = DECLARE VALUE$PTR
= DECLARE LENGTH
173 2 = DECLARE DATA$LNQ
174 2 = END G$ASCII;
175 2 =
=
=
176 1 = G$STRING:
= PROCEDURE (BUFF$PTR, LENGTH, STOP$SYM) BYTE EXTERNAL;
177 2 = DECLARE BUFF$PTR
178 2 = DECLARE LENGTH
179 2 = DECLARE STOP$SYM
180 2 = END G$STRING;
=
=
181 1 = G$ASCII$LNQ:
= PROCEDURE
= END G$ASCII$LNQ;
182 2 =
=
=

```

27 APRIL 1981

STRUCTURE(
BYTE,
BYTE) EXTERNAL;POINT;
BYTE;
BYTE;

BYTE EXTERNAL;

13	1	DECLARE MATCH			
14	1	DECLARE L\$THAN			
15	1	DECLARE \$THAN			
16	1	STRING\$MATCH:			
17	2	PROCEDURE (BUFF\$1\$PTR,	BUFF\$2\$PTR,	COUNT) BYTE	EXTERNAL;
18	2	DECLARE BUFF\$1\$PTR		POINTER;	
19	2	DECLARE BUFF\$2\$PTR		POINTER;	
190	2	DECLARE COUNT		BYTE;	
		END STRING\$MATCH;			

LINE	TEXT	DATE
1	\$/EJECT	
2	\$/INCLUDE(: F1: CNTL. EPD)	26 APRIL 1981
3	\$/CNTL. EPD	
4	\$/INCLUDE(: F1: FLAGS. EVD)	26 APRIL 1981
5	\$/FLAGS. EVD	
6	DECLARE PAGE\$FLAG	BYTE EXTERNAL;
7	DECLARE PRINT\$FLAG	BYTE EXTERNAL;
8	DECLARE DEFAULT\$PRINT\$FLAG	BYTE EXTERNAL;
9	DECLARE LIST\$FLAG	BYTE EXTERNAL;
10	DECLARE LIST\$FLAG#2	BYTE EXTERNAL;
11	DECLARE OBJECT\$FLAG	BYTE EXTERNAL;
12	DECLARE MACRO\$FLAG	BYTE EXTERNAL;
13	DECLARE STRUCTURED\$FLAG	BYTE EXTERNAL;
14	DECLARE DEBUG\$FLAG	BYTE EXTERNAL;
15	DECLARE STATEMENT\$CODE\$FLAG	BYTE EXTERNAL;
16		
17	CNTL\$INIT:	
18	PROCEDURE	EXTERNAL;
19	END CNTL\$INIT;	
20		
21	CNTL\$PROC:	
22	PROCEDURE (PASS)	BYTE EXTERNAL;
23	DECLARE PASS	BYTE;
24	END CNTL\$PROC;	
25		

		\$EJECT	
		\$INCLUDE(:F1:FILEIO.EPD)	
		/*	01.01.80 */
206	1	DECLARE FILE\$CO	LITERALLY '0';
207	1	DECLARE FILE\$CI	LITERALLY '1';
208	1	CO:	
209	2	PROCEDURE (CHAR)	EXTERNAL;
210	2	DECLARE CHAR	BYTE,
211	1	END CO;	
212	2	OPEN:	
213	2	PROCEDURE (AFTN\$PTR, FILE, ACCESS, MODE, STATUS) EXTERNAL;	
214	2	DECLARE AFTN\$PTR	POINTER;
215	2	DECLARE FILE	POINTER;
216	2	DECLARE ACCESS	WORD;
217	2	DECLARE MODE	WORD;
218	1	DECLARE STATUS	POINTER;
219	1	END OPEN;	
220	1	DECLARE OPEN\$INPUT	LITERALLY '1';
221	1	DECLARE OPEN\$OUTPUT	LITERALLY '2';
222	2	DECLARE OPEN\$UPDATE	LITERALLY '3';
223	2	READ:	
224	2	PROCEDURE (AFTN, BUFFER, COUNT, ACTUAL, STATUS) EXTERNAL;	
225	2	DECLARE AFTN	WORD;
226	2	DECLARE BUFFER	POINTER;
227	2	DECLARE COUNT	WORD;
228	1	DECLARE ACTUAL	POINTER;
229	2	DECLARE STATUS	POINTER;
230	2	END READ;	
231	2	WRITE:	
232	2	PROCEDURE (AFTN, BUFFER, COUNT, STATUS) EXTERNAL;	
233	2	DECLARE AFTN	WORD;
234	1	DECLARE BUFFER	POINTER;
235	2	DECLARE COUNT	WORD;
236	2	DECLARE STATUS	POINTER;
237	2	END WRITE;	
238	1	CLOSE:	
239	2	PROCEDURE (AFTN, STATUS)	EXTERNAL;
240	2	DECLARE AFTN	WORD;
241	2	DECLARE STATUS	POINTER;

```

237 2 -      END CLOSE;
238 1 -
239 2 -      DELETE:
240 2 -      PROCEDURE (FILE, STATUS)      EXTERNAL;
241 2 -      DECLARE FILE                    POINTER;
242 2 -      DECLARE STATUS                  POINTER;
243 2 -      END DELETE;
244 1 -      EXIT:
245 2 -      PROCEDURE                        EXTERNAL;
246 2 -      END EXIT;
247 1 -      ERROR:
248 2 -      PROCEDURE (ERRNUM)               EXTERNAL;
249 2 -      DECLARE ERRNUM                   POINTER;
250 2 -      END ERROR;
251 1 -      LOAD:
252 2 -      PROCEDURE (FILE, BIAS, SWITCH, ENTRY, STATUS) EXTERNAL;
253 2 -      DECLARE (FILE, BIAS, SWITCH, ENTRY, STATUS) ADDRESS;
254 2 -      END LOAD;
255 1 -      DECLARE SYSTEM$ERROR            LITERALLY 'OFFH';
256 2 -

```

\$EJECT

\$INCLUDE(:F1:LOCNT.EVD)

/* LOCNT.EVD

27 APRIL 1981 */

51 1 = DECLARE SEGMENT\$TYPE
52 1 = DECLARE EXTERNAL\$CNT
53 1 = DECLARE LOCATION\$CNT
54 1 = DECLARE CODE\$LOC\$CNT
55 1 = DECLARE DATA\$LOC\$CNT
56 1 = DECLARE ASEG\$LOC\$CNT

BYTE EXTERNAL;
BYTE EXTERNAL;
WORD EXTERNAL;
WORD EXTERNAL;
WORD EXTERNAL;
WORD EXTERNAL;

```

$EJECT
$INCLUDE(: F1: EXPRES. EPD)
/* EXPRES. ELD                                01 APRIL 1981
*/

37 1 = DECLARE EXPRES
    = ERROR
    = BEG$TYPE
    = EXTRN$NO
    = STRUCTURE(
    = BYTE,
    = BYTE,
    = BYTE) EXTERNAL;

38 1 = EXPRESSION:
    = PROCEDURE (VALUE$PTR, SEGMENT, DATA$SIZE) EXTERNAL;
259 2 = DECLARE VALUE$PTR
260 2 = DECLARE SEGMENT
261 2 = DECLARE DATA$SIZE
262 2 = END EXPRESSION;

```



```

$EJECT
$INCLUDE(:F1:PRINT.EPD)
/*
PRINT.ELD
*/
79 1 = DECLARE LISTING
    = = LINE
    = = OBJECT
    = =
30 1 = DECLARE NUMBER$OF$ERRORS
    = =
281 1 = PRINT$FINISH:
    = = PROCEDURE
282 2 = END PRINT$FINISH;
    = =
283 1 = PRINT$INIT:
    = = PROCEDURE
284 2 = END PRINT$INIT;
    = =
285 1 = PRINT$LINE:
    = = PROCEDURE (ERROR$TYPE)
286 2 = DECLARE ERROR$TYPE
287 2 = END PRINT$LINE;
    = =
07 APRIL 1981
STRUCTURE(
BYTE,
BYTE) EXTERNAL;
WORD EXTERNAL;
EXTERNAL;
EXTERNAL;
EXTERNAL;
EXTERNAL;
BYTE;

```

```

%EJECT
%INCLUDE(:F1:GETSYM.EPD)
/* GETSYM.ELD
*/
%INCLUDE(:F1:SYMTYP.ELD)
/* SYMTYP.ELD
*/
288 1 %1 DECLARE SYM$PTR WORD EXTERNAL;
289 1 %1 DECLARE (SYM$TYPE BASED SYM$PTR) BYTE;
290 1 %1 DECLARE (INSTRUCTION BASED SYM$PTR) STRUCTURE(
    TYPE
    INST
    INST$TYPE
    CODE
    LENGTH
    CHAR (1)
    %1
291 1 %1 DECLARE (REG BASED SYM$PTR) STRUCTURE(
    TYPE
    REGISTER
    SPEC$CODE
    R$CODE
    RP$CODE
    CHAR (1)
    %1
292 1 %1 DECLARE (MEMORY BASED SYM$PTR) STRUCTURE(
    TYPE
    INST
    DUMMY (3)
    CHAR (1)
    %1
293 1 %1 DECLARE (STATEMENT BASED SYM$PTR) STRUCTURE(
    TYPE
    INST
    DUMMY (3)
    CHAR (1)
    %1
294 1 %1 DECLARE (DATA$STORAGE BASED SYM$PTR) STRUCTURE(
    TYPE
    INST
    DUMMY (3)
    CHAR (1)
    %1

```

```

=1
75 1 =1 DECLARE (CONTROL BASED SYM#PTR) STRUCTURE(
=1 TYPE BYTE,
=1 INST BYTE,
=1 DUMMY (3) BYTE,
=1 CHAR (1) BYTE);

96 1 =1 DECLARE (IDENT BASED SYM#PTR) STRUCTURE(
=1 TYPE BYTE,
=1 STATUS BYTE,
=1 VALUE WORD,
=1 MARCO#LEV BYTE,
=1 CHAR (1) BYTE);

297 1 =1 DECLARE (NUMERIC BASED SYM#PTR) STRUCTURE(
=1 TYPE BYTE,
=1 VALUE WORD,
=1 DUMMY WORD,
=1 CHAR (1) BYTE);

298 1 =1 DECLARE (CURRENT#LOC BASED SYM#PTR) STRUCTURE(
=1 TYPE BYTE,
=1 VALUE WORD,
=1 SEGMENT BYTE);

299 1 =1 DECLARE (SKIP BASED SYM#PTR) STRUCTURE(
=1 TYPE BYTE,
=1 LINE#TYPE BYTE,
=1 ERROR#TYPE BYTE,
=1 PARM1 BYTE,
=1 PARM2 BYTE);

300 1 = DECLARE LINE#BUFF#2 (10) BYTE EXTERNAL;
301 1 = DECLARE LINE#BUFF (5) BYTE AT(.LINE#BUFF#2(5));

302 1 = G#CHAR$INIT;
303 2 = PROCEDURE (NEW#CHAR$CNT) EXTERNAL;
304 2 = DECLARE NEW#CHAR$CNT BYTE;
= END G#CHAR$INIT;

305 1 = G#CHAR$CNT;
306 2 = PROCEDURE BYTE EXTERNAL;
= END G#CHAR$CNT;

307 1 = GET$CHAR;
308 2 = PROCEDURE BYTE EXTERNAL;
= END GET$CHAR;

```

```

=
09 1 = GETSYM:
10 2 = PROCEDURE
11 1 = END GETSYM; EXTERNAL;

12 2 = GET$LINE$INIT:
13 1 = PROCEDURE EXTERNAL;
14 2 = END GET$LINE$INIT;
15 1 = GET$LINE:
16 2 = PROCEDURE BYTE EXTERNAL;
17 1 = END GET$LINE;
18 1 =

```

		\$EJECT	
		\$INCLUDE(: F1: SYMBOL. EPD)	
		/* SYMBOL. EPD	01 APRIL 1981
		*/	
15	1	DECLARE NO\$SYMBOLS	
16	1	DECLARE SYM\$TBL\$PTR	WORD EXTERNAL;
			POINTER EXTERNAL;
17	1	SYM\$INIT:	
		PROCEDURE	EXTERNAL;
18	2	END SYM\$INIT;	
319	1	SYM\$LOOKUP:	
		PROCEDURE (IDENT\$PTR)	BYTE EXTERNAL;
320	2	DECLARE IDENT\$PTR	POINTER;
321	2	END SYM\$LOOKUP;	
322	1	SYM\$ENTRY:	
		PROCEDURE (IDENT\$PTR)	BYTE EXTERNAL;
323	2	DECLARE IDENT\$PTR	POINTER;
324	2	END SYM\$ENTRY;	
325	1	SYM\$UPDATE:	
		PROCEDURE (IDENT\$PTR)	BYTE EXTERNAL;
326	2	DECLARE IDENT\$PTR	POINTER;
327	2	END SYM\$UPDATE;.	

```

$EJECT
$INCLUDE(:F1:MACRO.EPD)
/* MACRO.EPD      20 MARCH 1982 */

28 1 = DECLARE ACTIVE$MACRO$FLAG      BYTE EXTERNAL;
29 1 = MACRO$INIT:                     EXTERNAL;
30 2 = PROCEDURE                       EXTERNAL;
    = END MACRO$INIT;
31 1 = MACRO$DEFINE:
32 2 = PROCEDURE (M$IDENT$PTR)        EXTERNAL;
33 2 = DECLARE M$IDENT$PTR            POINTER;
    = END MACRO$DEFINE;
34 1 = MACRO$EXPAND:
35 2 = PROCEDURE (M$BUFF$PTR)        EXTERNAL;
36 2 = DECLARE M$BUFF$PTR            POINTER;
    = END MACRO$EXPAND;
37 1 = GET$MACRO$LINE:
38 2 = PROCEDURE                     BYTE EXTERNAL;
    = END GET$MACRO$LINE;
    =

```

```

$EJECT
$INCLUDE(:F1: COPYSR.EPD)
/* COPYSR.EPD */
19 1 = COPY$SOURCE$INIT: EXTERNAL;
10 2 = PROCEDURE
    END COPY$SOURCE$INIT;
11 1 = COPY$SOURCE$FINISH: EXTERNAL;
12 2 = PROCEDURE
    END COPY$SOURCE$FINISH;
343 1 = COPY$SOURCE: EXTERNAL;
344 2 = PROCEDURE (SOURCE$PTR, LNG) POINTER;
345 2 = DECLARE SOURCE$PTR WORD;
346 2 = END COPY$SOURCE;
347 1 = COPY$CHAR: EXTERNAL;
348 2 = PROCEDURE (SOURCE$PTR, LNG) POINTER;
349 2 = DECLARE SOURCE$PTR BYTE;
350 2 = END COPY$CHAR;
=
```



```

31  =
32  =
33  =
34  =
35  =
36  =
37  =
38  =
39  =
40  =
41  =
42  =
43  =
44  =
45  =
46  =
47  =
48  =
49  =
50  =
51  =
52  =
53  =
54  =
55  =
56  =
57  =
58  =
59  =
60  =
61  =
62  =
63  =
64  =
65  =
66  =
67  =
68  =
69  =
70  =
71  =
72  =
73  =
74  =
75  =
76  =
77  =
78  =
79  =
80  =
81  =
82  =
83  =
84  =
85  =
86  =
87  =
88  =
89  =
90  =
91  =
92  =
93  =
94  =
95  =
96  =
97  =
98  =
99  =
100  =
101  =
102  =
103  =
104  =
105  =
106  =
107  =
108  =
109  =
110  =
111  =
112  =
113  =
114  =
115  =
116  =
117  =
118  =
119  =
120  =
121  =
122  =
123  =
124  =
125  =
126  =
127  =
128  =
129  =
130  =
131  =
132  =
133  =
134  =
135  =
136  =
137  =
138  =
139  =
140  =
141  =
142  =
143  =
144  =
145  =
146  =
147  =
148  =
149  =
150  =
151  =
152  =
153  =
154  =
155  =
156  =
157  =
158  =
159  =
160  =
161  =
162  =
163  =
164  =
165  =
166  =
167  =
168  =
169  =
170  =
171  =
172  =
173  =
174  =
175  =
176  =
177  =
178  =
179  =
180  =
181  =
182  =
183  =
184  =
185  =
186  =
187  =
188  =
189  =
190  =
191  =
192  =
193  =
194  =
195  =
196  =
197  =
198  =
199  =
200  =
201  =
202  =
203  =
204  =
205  =
206  =
207  =
208  =
209  =
210  =
211  =
212  =
213  =
214  =
215  =
216  =
217  =
218  =
219  =
220  =
221  =
222  =
223  =
224  =
225  =
226  =
227  =
228  =
229  =
230  =
231  =
232  =
233  =
234  =
235  =
236  =
237  =
238  =
239  =
240  =
241  =
242  =
243  =
244  =
245  =
246  =
247  =
248  =
249  =
250  =
251  =
252  =
253  =
254  =
255  =
256  =
257  =
258  =
259  =
260  =
261  =
262  =
263  =
264  =
265  =
266  =
267  =
268  =
269  =
270  =
271  =
272  =
273  =
274  =
275  =
276  =
277  =
278  =
279  =
280  =
281  =
282  =
283  =
284  =
285  =
286  =
287  =
288  =
289  =
290  =
291  =
292  =
293  =
294  =
295  =
296  =
297  =
298  =
299  =
300  =
301  =
302  =
303  =
304  =
305  =
306  =
307  =
308  =
309  =
310  =
311  =
312  =
313  =
314  =
315  =
316  =
317  =
318  =
319  =
320  =
321  =
322  =
323  =
324  =
325  =
326  =
327  =
328  =
329  =
330  =
331  =
332  =
333  =
334  =
335  =
336  =
337  =
338  =
339  =
340  =
341  =
342  =
343  =
344  =
345  =
346  =
347  =
348  =
349  =
350  =
351  =
352  =
353  =
354  =
355  =
356  =
357  =
358  =
359  =
360  =
361  =
362  =
363  =
364  =
365  =
366  =
367  =
368  =
369  =
370  =
371  =
372  =
373  =
374  =
375  =
376  =
377  =
378  =
379  =
380  =
381  =
382  =
383  =
384  =
385  =
386  =
387  =
388  =
389  =
390  =
391  =
392  =
393  =
394  =
395  =
396  =
397  =
398  =
399  =
400  =
401  =
402  =
403  =
404  =
405  =
406  =
407  =
408  =
409  =
410  =
411  =
412  =
413  =
414  =
415  =
416  =
417  =
418  =
419  =
420  =
421  =
422  =
423  =
424  =
425  =
426  =
427  =
428  =
429  =
430  =
431  =
432  =
433  =
434  =
435  =
436  =
437  =
438  =
439  =
440  =
441  =
442  =
443  =
444  =
445  =
446  =
447  =
448  =
449  =
450  =
451  =
452  =
453  =
454  =
455  =
456  =
457  =
458  =
459  =
460  =
461  =
462  =
463  =
464  =
465  =
466  =
467  =
468  =
469  =
470  =
471  =
472  =
473  =
474  =
475  =
476  =
477  =
478  =
479  =
480  =
481  =
482  =
483  =
484  =
485  =
486  =
487  =
488  =
489  =
490  =
491  =
492  =
493  =
494  =
495  =
496  =
497  =
498  =
499  =
500  =
501  =
502  =
503  =
504  =
505  =
506  =
507  =
508  =
509  =
510  =
511  =
512  =
513  =
514  =
515  =
516  =
517  =
518  =
519  =
520  =
521  =
522  =
523  =
524  =
525  =
526  =
527  =
528  =
529  =
530  =
531  =
532  =
533  =
534  =
535  =
536  =
537  =
538  =
539  =
540  =
541  =
542  =
543  =
544  =
545  =
546  =
547  =
548  =
549  =
550  =
551  =
552  =
553  =
554  =
555  =
556  =
557  =
558  =
559  =
560  =
561  =
562  =
563  =
564  =
565  =
566  =
567  =
568  =
569  =
570  =
571  =
572  =
573  =
574  =
575  =
576  =
577  =
578  =
579  =
580  =
581  =
582  =
583  =
584  =
585  =
586  =
587  =
588  =
589  =
590  =
591  =
592  =
593  =
594  =
595  =
596  =
597  =
598  =
599  =
600  =
601  =
602  =
603  =
604  =
605  =
606  =
607  =
608  =
609  =
610  =
611  =
612  =
613  =
614  =
615  =
616  =
617  =
618  =
619  =
620  =
621  =
622  =
623  =
624  =
625  =
626  =
627  =
628  =
629  =
630  =
631  =
632  =
633  =
634  =
635  =
636  =
637  =
638  =
639  =
640  =
641  =
642  =
643  =
644  =
645  =
646  =
647  =
648  =
649  =
650  =
651  =
652  =
653  =
654  =
655  =
656  =
657  =
658  =
659  =
660  =
661  =
662  =
663  =
664  =
665  =
666  =
667  =
668  =
669  =
670  =
671  =
672  =
673  =
674  =
675  =
676  =
677  =
678  =
679  =
680  =
681  =
682  =
683  =
684  =
685  =
686  =
687  =
688  =
689  =
690  =
691  =
692  =
693  =
694  =
695  =
696  =
697  =
698  =
699  =
700  =
701  =
702  =
703  =
704  =
705  =
706  =
707  =
708  =
709  =
710  =
711  =
712  =
713  =
714  =
715  =
716  =
717  =
718  =
719  =
720  =
721  =
722  =
723  =
724 
```

MODULE INFORMATION:

```

CODE AREA SIZE      = 0000H
VARIABLE AREA SIZE  = 0000H
MAXIMUM STACK SIZE  = 0000H
728 LINES READ
0 PROGRAM ERROR(S)

```

END OF PL/M-80 COMPILATION

'L/M-80 COMPILER SMA MAIN ROUTINE START POINT (W/OVERLAYS)

10-MAR-82 PAGE 1

:SIS-II PL/M-80 V3.1 COMPILATION OF MODULE MAINOV
:IO OBJECT MODULE REQUESTED
:COMPILER INVOKED BY: PLM80 MAINOV.PLM DATE(10-MAR-82) NOOBJECT PRINT(:F1:MAINOV.LST) PAGEDLENGTH(50)

1 \$TITLE('SMA MAIN ROUTINE START POINT (W/OVERLAYS)')

MAINOV;
DO;

2 1 DECLARE REV820205 BYTE;

/* WRITTEN BY JOSEPH R. GARAPPOLO 26 APRIL 1981

INCLUDE FILES - INC .ELD
FILEIO.EPD

*/
\$NOLIST

```
$EJECT
67 1  ASSEMBLY$INIT:
68 2  PROCEDURE
69 1  END ASSEMBLY$INIT;
70 2  EXTERNAL;

69 1  SRC$FILE$RESET:
70 2  PROCEDURE
71 1  END SRC$FILE$RESET;
72 2  EXTERNAL;

71 1  ASSEMBLY$FINISH:
72 2  PROCEDURE
73 1  END ASSEMBLY$FINISH;
74 2  EXTERNAL;

73 1  MAIN$PASS1:
74 2  PROCEDURE
75 1  END MAIN$PASS1;
76 2  EXTERNAL;

75 1  MAIN$PASS2:
76 2  PROCEDURE
77 1  END MAIN$PASS2;
78 2  EXTERNAL;

77 1  MAIN$PASS3:
78 2  PROCEDURE
79 1  END MAIN$PASS3;
80 1  EXTERNAL;

79 1  DECLARE PASS$NUMBER
80 1  DECLARE STATUS
81 1  DECLARE ENTRY
81 1  EXTERNAL;
81 1  WORD;
81 1  WORD;
```

```

$EJECT

82 1 /* display startup message to console */
    CALL WRITE(O, (CR,LF,' STRUCTURED MACRO ASSEMBLER V1.0',CR,LF),37,.STATUS);

83 1 /* parse runstring and open input file */
    CALL ASSEMBLY$INIT;

/* PASS1 - expand Macro's, evaluate conditional assemble and generate code
   for structured statements. This pass outputs the source code read
   from the input file(s) along with the generated code into an
   intermediate file called SMABO.TMP */

84 1 PASS$NUMBER = 1;
/* load PASS$1 overlay from disk */
85 1 CALL LOAD((':F1:PASS1'),O,O,ENTRY,.STATUS);
/* transfer control to PASS$1 */
86 1 STATUS = MAIN$PASS1;

/* PASS2 - resolves first level of identifier definitions */

87 1 PASS$NUMBER = 2;
/* load PASS$2 overlay from disk */
88 1 CALL LOAD((':F1:PASS2'),O,O,ENTRY,.STATUS);
/* setup input file */
89 1 CALL SRC$FILE$RESET;
/* transfer control to PASS$2 */
90 1 STATUS = MAIN$PASS2;

/* PASS3 - resolve second level of identifier definitions, create object code
   and object code file, create listing of file */

91 1 PASS$NUMBER = 3;
/* load PASS$3 overlay from disk */
92 1 CALL LOAD((':F1:PASS3'),O,O,ENTRY,STATUS);
/* setup input file */
93 1 CALL SRC$FILE$RESET;
/* transfer control to PASS$3 */
94 1 STATUS = MAIN$PASS3;

/* clean up */

95 1 CALL ASSEMBLY$FINISH;
96 1 END MAINOV;

```

MODULE INFORMATION:

CODE AREA SIZE	=	00CEH	206D
VARIABLE AREA SIZE	=	0006H	6D
MAXIMUM STACK SIZE	=	000BH	8D
189 LINES READ			
0 PROGRAM ERROR(S)			

END OF PL/M-80 COMPILATION

```
SIS-II PL/M-80 V3.1 COMPILATION OF MODULE ASINIF
O OBJECT MODULE REQUESTED
COMPILER INVOKED BY:  PLM80 ASINFI.PLM DATE(10-MAR-82) NOOBJECT PRINT(:FI:ASINFI.LST) PAGELENGTH(50)
```

```
1            $TITLE('STRUCTURED MACRO ASSEMBLER INITIALIZATION AND COMPLETEION')
             ASINIF:
             DO;

2    1        DECLARE REV820204                                BYTE AT(0);

             /* WRITTEN BY JOSEPH R. GARAPPOLO                07 APRIL 1981

                             This module contains routines the parse the runstring and open the
                             files needed for each of the three passes.

             INCLUDE FILES - INC        .ELD
                             SYSTEM.ELD
                             CNTL        .EPD
                             GETSYM. EPD
                             FILEIO. EPD
                             PRINT        .EPD
                             OBJECT. EPD

             */

             $NOLIST
```

```
$EJECT

278 1 COPY$BUFFS:
279 PROCEDURE (DEST$PTR, SRC$PTR, COUNT) EXTERNAL;
280 DECLARE DEST$PTR
281 DECLARE SRC$PTR
282 DECLARE COUNT
283 END COPY$BUFFS;

283 1 NUMOUT:
284 PROCEDURE (VALUE, BASE, OUTPUT, LENGTH, PAD) EXTERNAL;
285 DECLARE VALUE
286 DECLARE BASE
287 DECLARE OUTPUT
288 DECLARE LENGTH
289 DECLARE PAD
290 END NUMOUT;

290 1 OPEN$ERROR:
291 PROCEDURE (STATUS)
292 DECLARE STATUS
293 END OPEN$ERROR;

293 1 DECLARE I
294 DECLARE J
295 DECLARE STATUS
296 DECLARE WORD$TEMP
297 DECLARE BYTE$TEMP
298 DECLARE SUFFIX$INDEX
299 DECLARE TEMP$FILE (14)
300 DECLARE SRC$FILE (14)

301 1 DECLARE INPUT$FILE (5)
    AFT$SRC
    BUFF$CNT
    BUFFER (256)

302 1 DECLARE RUN$STRING (LINE$SIZE)
303 1 DECLARE OPEN$FILE (1)
304 1 DECLARE PRINT$FILE (1)
305 1 DECLARE AFT$LIST
306 1 DECLARE AFT$OBJ
307 1 DECLARE AFT$TMP

308 1 DECLARE IF$COUNT
309 1 DECLARE WHILE$COUNT
310 1 DECLARE FOR$COUNT

    EXTERNAL;
    WORD;
    EXTERNAL;
    BYTE;
    BYTE;
    WORD;
    WORD;
    WORD;
    BYTE;
    BYTE;
    BYTE;
    STRUCTURE(
    WORD,
    WORD,
    BYTE) EXTERNAL;
    BYTE PUBLIC;
    BYTE EXTERNAL;
    BYTE EXTERNAL,
    WORD PUBLIC;
    WORD PUBLIC;
    WORD PUBLIC,
    BYTE PUBLIC;
    BYTE PUBLIC;
    BYTE PUBLIC;
    BYTE PUBLIC;
```

311 1 DECLARE CASE\$COUNT BYTE PUBLIC;


```

$EJECT

312 1  ASSEMBLY$INIT:
    PROCEDURE

        PUBLIC;

/* this routine parses the run string that invokes SMABO and opens
the input file and checks for controls. If any errors are encountered
return control to the ISIS Operating System.
*/

313 2  /* read the run string into line buffer. */
    CALL READ(1, LINE$BUFF, LINE$SIZE, .WORD$TEMP, STATUS);
314 2  /* save run string for use by print routine */
    CALL COPY$BUFFS (.RUN$STRING(27), LINE$BUFF, LINE$SIZE-27);

/* open source file and check for errors */

315 2  I = 0;
316 2  DO WHILE (LINE$BUFF(I) = ' ');
317 3  I = I + 1;
318 3  END;
319 2  BYTE$TEMP = I;
320 2  /* check for source file missing */
    IF (LINE$BUFF(I) = CR)
    THEN DO;
322 3  CALL WRITE(0, (CR,LF,'MISSING FILE NAME'), 19, .STATUS);
323 3  CALL WRITE(0, (CR,LF,'EXECUTION TERMINATED',CR,LF),24,.STATUS);
324 3  CALL EXIT;
325 3  END;

/* open source code file */
326 2  CALL OPEN(.INPUT$FILE(0).AFT$SRC, .LINE$BUFF(I), 1, 0, STATUS);
/* check for open error */
327 2  IF (STATUS <> 0)
    THEN DO;
329 3  CALL WRITE(0, (CR,LF,'FILE DOESN'T EXIST'), 22, .STATUS);
330 3  CALL WRITE(0, (CR,LF,'EXECUTION TERMINATED',CR,LF),24, STATUS);
331 3  CALL EXIT;
332 3  END;

/* move source file name into temp$file buffer. Temp$file
is used to append the obj and 1st suffix onto the
source file name.
*/
333 2  DO WHILE (LINE$BUFF(I) <> ' ') AND ((I-BYTE$TEMP) <= 14) AND
    (LINE$BUFF(I) <> ' ');
334 3  TEMP$FILE(I-BYTE$TEMP) = LINE$BUFF(I),

```

```

335 3      I = I + 1;
336 3      END;
337 2      SUFFIX$INDEX = I - BYTE$TEMP + 1;
338 2      TEMP$FILE(I-BYTE$TEMP) = ' ',
339 2      DO J = (I-BYTE$TEMP) TO 12;
340 3      TEMP$FILE(J) = ' ',
341 3      END;

/* initialize control flags and check run string for controls */
342 2      DO WHILE (LINE$BUFF(I) <> ' ') AND (LINE$BUFF(I) <> CR);
343 3      I = I + 1;
344 3      END;
345 2      CALL G$CHAR$INIT(I-1);
346 2      CALL CNTL$INIT;
347 2      CALL GET$SYM;
348 2      STATUS = NULL;
349 2      DO WHILE (SYM$TYPE = CONTROL$SYM) AND (STATUS = NULL);
350 3      STATUS = CNTL$PROC(NULL);
351 3      CALL GET$SYM;
352 3      END /* WHILE */;
353 2      /* this is not an control or an end of line report error */
      IF (SYM$TYPE <> EOL$SYM) OR (STATUS <> NULL)
      THEN DO;
355 3          CALL WRITE(0, ' (CR,LF, 'UNRECOGNIZED COMMAND',22, 'STATUS);
356 3          CALL WRITE(0, ' (CR,LF, 'EXECUTION TERMINATED',CR,LF),24, 'STATUS);
357 3          CALL EXIT;
358 3          END;
359 2      END ASSEMBLY$INIT;

```

```

$EJECT
360 1 FILE$INIT$PASS1:
PROCEDURE PUBLIC;
/* see what disk the source code file is on so that SMABO.TMP can
   be put on the same disk */
361 2 IF (TEMP$FILE(0) = '.')
THEN DO;
363 3 CALL COPY$BUFFS(.SRC$FILE, .TEMP$FILE, 4);
364 3 CALL COPY$BUFFS(.SRC$FILE(4), ('SMABO.TMP '), 10);
365 3 END;
366 2 ELSE CALL COPY$BUFFS(.SRC$FILE, ('SMABO.TMP '), 14);
367 2 CALL COPY$BUFFS(.OPEN$FILE, .SRC$FILE, 14);
/* delete SMABO.TMP if it already exist */
368 2 CALL DELETE(.OPEN$FILE, STATUS);
/* open intermediate file for write */
369 2 CALL OPEN(.AFT$TMP, .OPEN$FILE, 2, 0, STATUS);
/* check for open error */
370 2 CALL OPEN$ERROR(STATUS);
371 2 END FILE$INIT$PASS1;

```

```

$EJECT
372 1      FILE$INIT$PASS3:
           PROCEDURE
           PUBLIC;

           /* open, object and print files if they are needed */

373 2      IF (PRINT$FLAG = ON)
           THEN DO;
375 3          IF (DEFAULT$PRINT$FLAG = ON)
           THEN DO,
           /* use default list file, source file.LST */
           CALL COPY$BUFFS(.OPEN$FILE, TEMP$FILE, 14);
           CALL COPY$BUFFS(.OPEN$FILE(SUFFIX$INDEX), ('LST'), 3);
           END;
           /* use the name specified in th print control */
           ELSE CALL COPY$BUFFS(.OPEN$FILE, .PRINT$FILE, 14);
           CALL DELETE(.OPEN$FILE, .STATUS);
           CALL OPEN(.AFT$LST, .OPEN$FILE, 2, 0, STATUS);
           CALL OPEN$ERROR(STATUS);
           END;
           IF (OBJECT$FLAG = ON)
           THEN DO,
           /* open the object code file */
           CALL COPY$BUFFS(.OPEN$FILE, .TEMP$FILE, 14);
           CALL COPY$BUFFS(.OPEN$FILE(SUFFIX$INDEX), ('OBJ'), 3);
           CALL DELETE(.OPEN$FILE, .STATUS);
           CALL OPEN(.AFT$OBJ, .OPEN$FILE, 2, 0, STATUS);
           CALL OPEN$ERROR(STATUS);
           END;

393 2      END FILE$INIT$PASS3;
    
```

```

$EJECT
394 1  SRC$FILE$RESET:
      PROCEDURE                                PUBLIC;

      /* this routine resets the input file */

395 2  CALL CLOSE (AFT$TMP, .STATUS);
396 2  CALL CLOSE (INPUT$FILE(0).AFT$SRC, .STATUS);
397 2  CALL OPEN ( INPUT$FILE(0).AFT$SRC, .SRC$FILE, 1, 0, STATUS);

398 2  END SRC$FILE$RESET;
```

```

$EJECT
399 1  ASSEMBLY$FINISH:
      PROCEDURE
      PUBLIC;

      /* This file cleans up SMABO before going back to ISIS */

400 2  IF (PRINT$FLAG = ON)
      THEN DO;
402 3      CALL PRINT$FINISH;
403 3      CALL CLOSE(AFT$LIST, .STATUS);
404 3      END;
405 2  IF (OBJECT$FLAG = ON)
      THEN DO;
407 3      CALL OBJECT$FINISH;
408 3      CALL CLOSE(AFT$OBJ, .STATUS);
409 3      END;
410 2  CALL CLOSE (.INPUT$FILE(O).AFT$SRC, .STATUS);
411 2  CALL DELETE (.SRC$FILE, .STATUS);

      /* write number of errors and end of assembly to console */

412 2  CALL COPY$BUFFS(.LINE$BUFF, (' PROGRAM ERROR(S)', CR, LF, LF), 26);
413 2  IF (NUMBER$OF$ERRORS = 0)
      THEN LINE$BUFF(5) = '0';
      ELSE CALL NUMOUT(NUMBER$OF$ERRORS, 10, .LINE$BUFF(2), 4, ' ');
415 2  CALL WRITE(O, .LINE$BUFF, 26, .STATUS);
416 2  CALL WRITE(O, ('SMABO COMPLETED', CR, LF), 17, .STATUS);
417 2  CALL EXIT;
418 2  END ASSEMBLY$FINISH;
419 2

420 1  END ASINIF;
    
```

MODULE INFORMATION:

```

CODE AREA SIZE      = 04A1H    1185D
VARIABLE AREA SIZE = 0094H    148D
MAXIMUM STACK SIZE = 0008H    8D
770 LINES READ
0 PROGRAM ERROR(S)
    
```

END OF PL/M-80 COMPILATION

315-11 PL/M-80 V3.1 COMPILATION OF MODULE SEGMENT
 3 OBJECT MODULE REQUESTED
 3MPILER INVOKED BY: PLM80 SEGMENT.PLM DATE(10-MAR-82) NOOBJECT PRINT(:F1:SEGMENT.LST) PAGELENGTH(50)

```

1      $TITLE('INITIALIZATION AND COMPLETION OF PROGRAM SEGMENTS AND COUNTERS')
      SEGMENT:
      DO;
  
```

```

2  1      DECLARE REV820128                                BYTE AT(0);
  
```

```

/* WRITTEN BY JOSEPH R. GARAPPOLO      20 MAY 1981
  
```

```

INCLUDE FILES - INC    ELD
                   SYSTEM.ELD
  
```

DESCRIPTION:

This module contains two routines one initializes the program location
 counters and the other one cleans up the location counter

```

*/
$NOLIST
  
```

\$EJECT

```

166 1    DECLARE SEGMENT$TYPE                    BYTE PUBLIC;
167 1    DECLARE EXTERNAL$CNT                    BYTE PUBLIC;
168 1    DECLARE LOCATION$CNT                   WORD PUBLIC;
169 1    DECLARE CODE$LOC$CNT                   WORD PUBLIC;
170 1    DECLARE DATA$LOC$CNT                  WORD PUBLIC;
171 1    DECLARE ASEG$LOC$CNT                   WORD PUBLIC;

172 1    SEGMENT$INIT.                           PUBLIC;
PROCEDURE

```

```

173 2    /* set the default segment type to code segment */
174 2    SEGMENT$TYPE = CODE$SEG;
175 2    EXTERNAL$CNT = 0;
176 2    CODE$LOC$CNT = 0;
177 2    DATA$LOC$CNT = 0;
178 2    ASEG$LOC$CNT = 0;
179 2    LOCATION$CNT = 0;

179 2    END SEGMENT$INIT;

```



```

$EJECT
180 1      SEGMENT$FINISH:
      PROCEDURE
          PUBLIC;

181 2      /* update the current location counter before exiting SMABO */
          IF (SEGMENT$TYPE = CODE$SEG)
              THEN CODE$LOC$CNT = LOCATION$CNT;
          ELSE
183 2      IF (SEGMENT$TYPE = DATA$SEG)
              THEN DATA$LOC$CNT = LOCATION$CNT;
          ELSE ASEG$LOC$CNT = LOCATION$CNT;
185 2
186 2      END SEGMENT$FINISH;
187 1      END SEGMENT;

```

MODULE INFORMATION:

```

CODE AREA SIZE      = 0041H      65D
VARIABLE AREA SIZE = 000AH      10D
MAXIMUM STACK SIZE = 0000H      0D
307 LINES READ
0 PROGRAM ERROR(S)

```

END OF PL/M-80 COMPILATION

SIS-II PL/M-80 V3.1 COMPILATION OF MODULE OPNMSG
 3 OBJECT MODULE REQUESTED
 COMPILER INVOKED BY: PLM80 OPNMSG.PLM DATE(10-MAR-82) NOOBJECT PRINT(:F1:OPNMSG.LST) PAGELENGTH(50)

```

1            $TITLE('PRINT OPEN ERROR AND FILENAME - ISIS OPEN ROUTINE ')
   OPNMSG:
   DO;

```

```

2    1       DECLARE REV820204                                BYTE AT(O);

```

```

/* WRITTEN BY JOSEPH R. GARAPPOLO

```

```

INCLUDE FILES - INC     .ELD
                 FILEIO.EPD

```

```

PARAMETERS: STATUS - status of the open file call

```

```

DESCRIPTION:

```

```

This routine checks the status of the ISIS open file call. If an
error exist it is displayed along with the file name on the console
and SMABO is temainalted with control going back to ISIS. The name
of the file opened is in OPEN$FILE.

```

```

*/
$NOLIST

```

L/M-80 COMPILER PRINT OPEN ERROR AND FILENAME - ISIS OPEN ROUTINE

10-MAR-82 PAGE 2

\$EJECT

67 1

 DECLARE OPEN\$FILE (13)

 BYTE PUBLIC;

```

$EJECT
68 1 OPEN$ERROR:
    PROCEDURE (STATUS)                PUBLIC,
69 2 DECLARE STATUS                    WORD,
70 2 DO CASE STATUS,
71 3 /* NO ACTION ON STATUS' 0, 1 AND 2 */
    ;,
74 3 /* STATUS - 3 */
    DO,
75 4 CALL WRITE(O, OPEN$FILE, 13, STATUS),
76 4 CALL WRITE(O, (CR,LF,'TO MANY OPEN FILES',CR,LF), 22, STATUS);
77 4 CALL EXIT;
78 4 END,
79 3 /* STATUS - 4 */
    DO,
80 4 CALL WRITE(O, OPEN$FILE, 13, STATUS);
81 4 CALL WRITE(O, (CR,LF,'ILLEGAL FILENAME',CR,LF), 20, STATUS);
82 4 CALL EXIT;
83 4 END,
84 3 /* STATUS - 5 */
    DO,
85 4 CALL WRITE(O, OPEN$FILE, 13, STATUS);
86 4 CALL WRITE(O, (CR,LF,'ILLEGAL DEVICE SPECIFIED',CR,LF), 28, STATUS);
87 4 CALL EXIT;
88 4 END,
89 3 /* NO ACTION ON STATUS' 6,7,8,9,10 AND 11 */
    ;,;
95 3 /* STATUS - 12 */
    DO,
96 4 CALL WRITE(O, OPEN$FILE, 13, STATUS);
97 4 CALL WRITE(O, (CR,LF,'FILE ALREADY OPEN OR DUPLICATE NAME',CR,LF), 39, STATUS);
98 4 CALL EXIT;
99 4 END,
100 3 /* STATUS - 13 */
    DO,
101 4 CALL WRITE(O, OPEN$FILE, 13, STATUS);
102 4 CALL WRITE(O, (CR,LF,'NO SUCH FILE',CR,LF), 16, STATUS);

```

```

103 4          CALL EXIT;
104 4      END;

105 3      /* STATUS - 14 */
106 4      DO;
107 4          CALL WRITE(O, . OPEN$FILE, 13, . STATUS);
108 4          CALL WRITE(O, (CR,LF,'FILE WRITE PROTECTED',CR,LF), 24, . STATUS);
109 4          CALL EXIT;
      END;

110 3      /* NO ACTION ON STATUS' 15,16,17,18,19,20 AND 21 */
      ;;;

117 3      /* STATUS - 22 */
118 4      DO;
119 4          CALL WRITE(O, . OPEN$FILE, 13, . STATUS);
120 4          CALL WRITE(O, (CR,LF,'ILLEGAL ACCESS PARAMETER',CR,LF), 28, . STATUS);
121 4          CALL EXIT;
      END;

122 3      /* STATUS - 23 */
123 4      DO;
124 4          CALL WRITE(O, OPEN$FILE, 13, . STATUS);
125 4          CALL WRITE(O, (CR,LF,'NO FILENAME SPECIFIED FOR DISK FILE',CR,LF), 39, . STATUS);
126 4          CALL EXIT;
      END;

127 3      /* NO ACTION ON STATUS 24 */
      ;

128 3      /* STATUS - 25 */
129 4      DO;
130 4          CALL WRITE(O, OPEN$FILE, 13, . STATUS);
131 4          CALL WRITE(O, (CR,LF,'ECHO FILE INCORRECT',CR,LF), 24, . STATUS);
132 4          CALL EXIT;
      END;

133 3      /* NO ACTION ON STATUS' 26 AND 27 */
      ;;

135 3      /* STATUS - 28 */
136 4      DO;
137 4          CALL WRITE(O, OPEN$FILE, 13, . STATUS);
138 4          CALL WRITE(O, (CR,LF,'NULL FILE EXTENSION',CR,LF), 23, . STATUS);
139 4          CALL EXIT;
      END;
    
```

./M-80 COMPILER PRINT OPEN ERROR AND FILENAME - ISIS OPEN ROUTINE

10-MAR-82 PAGE 5

.40 3 END /* CASE *//;

.41 2 END OPEN\$ERROR;

.42 1 END OPNMSG;

MODULE INFORMATION:

CODE AREA SIZE	=	031EH	798D
VARIABLE AREA SIZE	=	000FH	15D
MAXIMUM STACK SIZE	=	0006H	6D
226 LINES READ			
0 PROGRAM ERROR(S)			

END OF PL/M-80 COMPILATION

-II PL/M-80 V3.1 COMPILATION OF MODULE GETLIN
 CT MODULE PLACED IN : F4:GETLIN.OBJ
 ILER INVOKED BY: PLM80 GETLIN.PLM DATE(27-MAR-82) OBJECT(:F4:GETLIN.OBJ) PRINT(:F1:GETLIN.LST) DEBUG PAGELENGTH(50)

*TITLE('CONSTRUCT LINE AND MANAGE FILES')

GET\$LIN:

DO;

1 DECLARE REV820327

 BYTE AT(0);

/*

WRITTEN BY - JOSEPH R. GARAPPOLO 06 JUNE 1981

DESCRIPTION:

This module contains the routines for initializing, managing the input file(s) and for building the input line, LINE BUFFER.

For the first pass getline is initialized to handle up to five input files. One for the main source code file and four for INCLUDE files, which can be nested four level deep. During the first pass line may come from macro expansion in which case the active\$macro\$flag is set on, indicating to getline that get\$macro\$line should be called to get the current line of source code.

The second and third passes only require the Intermediate file, which contains the source code scanned by the first pass.

INCLUDE FILES - INC .ELD
 SYSTEM.ELD
 FILEIO.EPD
 MACRO .EPD

*/
 \$NOLIST

```

$EJECT
7 1  G$CHAR$INIT:
8      PROCEDURE (CHAR$CNT)
9      DECLARE CHAR$CNT
      END G$CHAR$INIT;
0 1  DECLARE LEVEL$DEPTH

1 1  DECLARE LINE$BUFF$2 (10)
2 1  DECLARE LINE$BUFF (5)
3 1  DECLARE PASS$NUMBER
4 1  DECLARE LEVEL

/* Input File Control Blocks */
235 1 DECLARE INPUT$FILE (LEVEL$DEPTH)
      AFT$SRC
      BUFF$CNT
      BUFFER (256)

236 1 DECLARE CHAR$CNT
237 1 DECLARE FILE$STATUS
238 1 DECLARE END$OF$LINE
239 1 DECLARE BUFF$SIZE

EXTERNAL;
BYTE;

LITERALLY '5';

BYTE EXTERNAL;
BYTE AT( LINE$BUFF$2(5));
BYTE EXTERNAL;
BYTE EXTERNAL;

STRUCTURE(
WORD,
WORD,
BYTE) PUBLIC;

BYTE;
BYTE;
BYTE;
WORD;
```



```

$EJECT
1  GET$LINE$INIT:
   PROCEDURE
2  DECLARE I
   PUBLIC;
   BYTE;
3  /* first pass */
4  IF (PASS$NUMBER = 1)
5  THEN DO;
6      /* initialize all five input file control blocks
7      with a buffer size of 256 bytes */
8      BUFF$SIZE = 256;
9      DO I = 0 TO LEVEL$DEPTH-1;
10         INPUT$FILE(I).BUFF$CNT = BUFF$SIZE;
11     END;
12     END;
13     /* passes two and three */
14     ELSE DO;
15         /* initialize the first input file control block only.
16         make the buffer size 1280 bytes. This larger buffer
17         uses the space no longer needed by the four additional
18         control blocks */
19         BUFF$SIZE = 1280;
20         INPUT$FILE(0).BUFF$CNT = BUFF$SIZE;
21     END;
22     FILE$STATUS = TRUE;
23 END GET$LINE$INIT;
24 END GET$LINE$INIT;

```

```

272 3 1 GET$LINE:
273 3 2 PROCEDURE
274 3 3
275 3 4
276 3 5
277 3 6
278 3 7
279 3 8
280 3 9
281 3 10
282 3 11
283 3 12
284 3 13
285 3 14
286 3 15
287 3 16
288 3 17
289 3 18
290 3 19
291 3 20
292 3 21
293 3 22
294 3 23
295 3 24
296 3 25
297 3 26
298 3 27
299 3 28
300 3 29
301 3 30
302 3 31
303 3 32
304 3 33
305 3 34
306 3 35
307 3 36
308 3 37
309 3 38
310 3 39
311 3 40
312 3 41
313 3 42
314 3 43
315 3 44
316 3 45
317 3 46
318 3 47
319 3 48
320 3 49
321 3 50
322 3 51
323 3 52
324 3 53
325 3 54
326 3 55
327 3 56
328 3 57
329 3 58
330 3 59
331 3 60
332 3 61
333 3 62
334 3 63
335 3 64
336 3 65
337 3 66
338 3 67
339 3 68
340 3 69
341 3 70
342 3 71
343 3 72
344 3 73
345 3 74
346 3 75
347 3 76
348 3 77
349 3 78
350 3 79
351 3 80
352 3 81
353 3 82
354 3 83
355 3 84
356 3 85
357 3 86
358 3 87
359 3 88
360 3 89
361 3 90
362 3 91
363 3 92
364 3 93
365 3 94
366 3 95
367 3 96
368 3 97
369 3 98
370 3 99
371 3 100
372 3 101
373 3 102
374 3 103
375 3 104
376 3 105
377 3 106
378 3 107
379 3 108
380 3 109
381 3 110
382 3 111
383 3 112
384 3 113
385 3 114
386 3 115
387 3 116
388 3 117
389 3 118
390 3 119
391 3 120
392 3 121
393 3 122
394 3 123
395 3 124
396 3 125
397 3 126
398 3 127
399 3 128
400 3 129
401 3 130
402 3 131
403 3 132
404 3 133
405 3 134
406 3 135
407 3 136
408 3 137
409 3 138
410 3 139
411 3 140
412 3 141
413 3 142
414 3 143
415 3 144
416 3 145
417 3 146
418 3 147
419 3 148
420 3 149
421 3 150
422 3 151
423 3 152
424 3 153
425 3 154
426 3 155
427 3 156
428 3 157
429 3 158
430 3 159
431 3 160
432 3 161
433 3 162
434 3 163
435 3 164
436 3 165
437 3 166
438 3 167
439 3 168
440 3 169
441 3 170
442 3 171
443 3 172
444 3 173
445 3 174
446 3 175
447 3 176
448 3 177
449 3 178
450 3 179
451 3 180
452 3 181
453 3 182
454 3 183
455 3 184
456 3 185
457 3 186
458 3 187
459 3 188
460 3 189
461 3 190
462 3 191
463 3 192
464 3 193
465 3 194
466 3 195
467 3 196
468 3 197
469 3 198
470 3 199
471 3 200
472 3 201
473 3 202
474 3 203
475 3 204
476 3 205
477 3 206
478 3 207
479 3 208
480 3 209
481 3 210
482 3 211
483 3 212
484 3 213
485 3 214
486 3 215
487 3 216
488 3 217
489 3 218
490 3 219
491 3 220
492 3 221
493 3 222
494 3 223
495 3 224
496 3 225
497 3 226
498 3 227
499 3 228
500 3 229
501 3 230
502 3 231
503 3 232
504 3 233
505 3 234
506 3 235
507 3 236
508 3 237
509 3 238
510 3 239
511 3 240
512 3 241
513 3 242
514 3 243
515 3 244
516 3 245
517 3 246
518 3 247
519 3 248
520 3 249
521 3 250
522 3 251
523 3 252
524 3 253
525 3 254
526 3 255
527 3 256
528 3 257
529 3 258
530 3 259
531 3 260
532 3 261
533 3 262
534 3 263
535 3 264
536 3 265
537 3 266
538 3 267
539 3 268
540 3 269
541 3 270
542 3 271
543 3 272
544 3 273
545 3 274
546 3 275
547 3 276
548 3 277
549 3 278
550 3 279
551 3 280
552 3 281
553 3 282
554 3 283
555 3 284
556 3 285
557 3 286
558 3 287
559 3 288
560 3 289
561 3 290
562 3 291
563 3 292
564 3 293
565 3 294
566 3 295
567 3 296
568 3 297
569 3 298
570 3 299
571 3 300
572 3 301
573 3 302
574 3 303
575 3 304
576 3 305
577 3 306
578 3 307
579 3 308
580 3 309
581 3 310
582 3 311
583 3 312
584 3 313
585 3 314
586 3 315
587 3 316
588 3 317
589 3 318
590 3 319
591 3 320
592 3 321
593 3 322
594 3 323
595 3 324
596 3 325
597 3 326
598 3 327
599 3 328
600 3 329
601 3 330
602 3 331
603 3 332
604 3 333
605 3 334
606 3 335
607 3 336
608 3 337
609 3 338
610 3 339
611 3 340
612 3 341
613 3 342
614 3 343
615 3 344
616 3 345
617 3 346
618 3 347
619 3 348
620 3 349
621 3 350
622 3 351
623 3 352
624 3 353
625 3 354
626 3 355
627 3 356
628 3 357
629 3 358
630 3 359
631 3 360
632 3 361
633 3 362
634 3 363
635 3 364
636 3 365
637 3 366
638 3 367
639 3 368
640 3 369
641 3 370
642 3 371
643 3 372
644 3 373
645 3 374
646 3 375
647 3 376
648 3 377
649 3 378
650 3 379
651 3 380
652 3 381
653 3 382
654 3 383
655 3 384
656 3 385
657 3 386
658 3 387
659 3 388
660 3 389
661 3 390
662 3 391
663 3 392
664 3 393
665 3 394
666 3 395
667 3 396
668 3 397
669 3 398
670 3 399
671 3 400
672 3 401
673 3 402
674 3 403
675 3 404
676 3 405
677 3 406
678 3 407
679 3 408
680 3 409
681 3 410
682 3 411
683 3 412
684 3 413
685 3 414
686 3 415
687 3 416
688 3 417
689 3 418
690 3 419
691 3 420
692 3 421
693 3 422
694 3 423
695 3 424
696 3 425
697 3 426
698 3 427
699 3 428
700 3 429
701 3 430
702 3 431
703 3 432
704 3 433
705 3 434
706 3 435
707 3 436
708 3 437
709 3 438
710 3 439
711 3 440
712 3 441
713 3 442
714 3 443
715 3 444
716 3 445
717 3 446
718 3 447
719 3 448
720 3 449
721 3 450
722 3 451
723 3 452
724 3 453
725 3 454
726 3 455
727 3 456
728 3 457
729 3 458
730 3 459
731 3 460
732 3 461
733 3 462
734 3 463
735 3 464
736 3 465
737 3 466
738 3 467
739 3 468
740 3 469
741 3 470
742 3 471
743 3 472
744 3 473
745 3 474
746 3 475
747 3 476
748 3 477
749 3 478
750 3 479
751 3 480
752 3 481
753 3 482
754 3 483
755 3 484
756 3 485
757 3 486
758 3 487
759 3 488
760 3 489
761 3 490
762 3 491
763 3 492
764 3 493
765 3 494
766 3 495
767 3 496
768 3 497
769 3 498
770 3 499
771 3 500
772 3 501
773 3 502
774 3 503
775 3 504
776 3 505
777 3 506
778 3 507
779 3 508
780 3 509
781 3 510
782 3 511
783 3 512
784 3 513
785 3 514
786 3 515
787 3 516
788 3 517
789 3 518
790 3 519
791 3 520
792 3 521
793 3 522
794 3 523
795 3 524
796 3 525
797 3 526
798 3 527
799 3 528
800 3 529
801 3 530
802 3 531
803 3 532
804 3 533
805 3 534
806 3 535
807 3 536
808 3 537
809 3 538
810 3 539
811 3 540
812 3 541
813 3 542
814 3 543
815 3 544
816 3 545
817 3 546
818 3 547
819 3 548
820 3 549
821 3 550
822 3 551
823 3 552
824 3 553
825 3 554
826 3 555
827 3 556
828 3 557
829 3 558
830 3 559
831 3 560
832 3 561
833 3 562
834 3 563
835 3 564
836 3 565
837 3 566
838 3 567
839 3 568
840 3 569
841 3 570
842 3 571
843 3 572
844 3 573
845 3 574
846 3 575
847 3 576
848 3 577
849 3 578
850 3 579
851 3 580
852 3 581
853 3 582
854 3 583
855 3 584
856 3 585
857 3 586
858 3 587
859 3 588
860 3 589
861 3 590
862 3 591
863 3 592
864 3 593
865 3 594
866 3 595
867 3 596
868 3 597
869 3 598
870 3 599
871 3 600
872 3 601
873 3 602
874 3 603
875 3 604
876 3 605
877 3 606
878 3 607
879 3 608
880 3 609
881 3 610
882 3 611
883 3 612
884 3 613
885 3 614
886 3 615
887 3 616
888 3 617
889 3 618
890 3 619
891 3 620
892 3 621
893 3 622
894 3 623
895 3 624
896 3 625
897 3 626
898 3 627
899 3 628
900 3 629
901 3 630
902 3 631
903 3 632
904 3 633
905 3 634
906 3 635
907 3 636
908 3 637
909 3 638
910 3 639
911 3 640
912 3 641
913 3 642
914 3 643
915 3 644
916 3 645
917 3 646
918 3 647
919 3 648
920 3 649
921 3 650
922 3 651
923 3 652
924 3 653
925 3 654
926 3 655
927 3 656
928 3 657
929 3 658
930 3 659
931 3 660
932 3 661
933 3 662
934 3 663
935 3 664
936 3 665
937 3 666
938 3 667
939 3 668
940 3 669
941 3 670
942 3 671
943 3 672
944 3 673
945 3 674
946 3 675
947 3 676
948 3 677
949 3 678
950 3 679
951 3 680
952 3 681
953 3 682
954 3 683
955 3 684
956 3 685
957 3 686
958 3 687
959 3 688
960 3 689
961 3 690
962 3 691
963 3 692
964 3 693
965 3 694
966 3 695
967 3 696
968 3 697
969 3 698
970 3 699
971 3 700
972 3 701
973 3 702
974 3 703
975 3 704
976 3 705
977 3 706
978 3 707
979 3 708
980 3 709
981 3 710
982 3 711
983 3 712
984 3 713
985 3 714
986 3 715
987 3 716
988 3 717
989 3 718
990 3 719
991 3 720
992 3 721
993 3 722
994 3 723
995 3 724
996 3 725
997 3 726
998 3 727
999 3 728
1000 3 729

```


PLM80 GETSYM. PLM DATE(27-MAR-82) OBJECT(:F4:GETSYM.OBJ) PRINT(:F1:GETSYM.LST) PAGELENGTH(50)

```
1 $TITLE('LEXICAL ANALYZER')
  GET$SYMBOL:
  DO,
2 1 DECLARE REV820327 BYTE AT(0),
/* WRITTEN BY JOSEPH R. GARAPPOLO 20 AUG 80
```

INCLUDED FILES ARE - INC .ELD
SYSTEM.ELD
LOCENT. EVD

DESCRIPTION:

This routine converts the symbols in the LINE BUFFER into TOKENS.
Additional information may be provided depending on the type of token.

*/
\$NOLIST

\$EJECT

```

16 1 NUM$IN:
17 2 PROCEDURE (NUM$PTR)
18 2 DECLARE NUM$PTR
19 2 END NUM$IN;
20 1 SEARCH:
21 2 PROCEDURE (SYM$PTR, TABLE$PTR, LNG)
22 2 DECLARE SYM$PTR
23 2 DECLARE TABLE$PTR
24 2 DECLARE LNG
25 2 END SEARCH;
26 1 DECLARE SYM$PTR
27 1 DECLARE SYM$BUFF (25)
28 1 DECLARE (SYM$TYPE BASED SYM$PTR)
29 1 DECLARE (INSTRUCTION BASED SYM$PTR)
30 1 TYPE
31 1 INST
32 1 INST$TYPE
33 1 CODE
34 1 LENGTH
35 1 CHAR (1)
36 1 DECLARE (IDENT BASED SYM$PTR)
37 1 TYPE
38 1 STATUS
39 1 VALUE
40 1 DUMMY
41 1 CHAR (SYM$LENGTH)
42 1 DECLARE (NUMERIC BASED SYM$PTR)
43 1 TYPE
44 1 VALUE
45 1 MACRO$LEV
46 1 CHAR (1)
47 1 DECLARE (CURRENT$LOC BASED SYM$PTR)
48 1 TYPE
49 1 VALUE
50 1 SEGMENT
51 1 DECLARE (SKIP BASED SYM$PTR)
52 1 TYPE
53 1 WORD EXTERNAL;
54 1 POINTER;
55 1 BYTE EXTERNAL;
56 1 POINTER;
57 1 BYTE;
58 1 WORD PUBLIC;
59 1 BYTE;
60 1 BYTE;
61 1 STRUCTURE(
62 1 BYTE,
63 1 BYTE,
64 1 WORD,
65 1 BYTE,
66 1 BYTE);
67 1 STRUCTURE(
68 1 BYTE,
69 1 WORD,
70 1 WORD,
71 1 BYTE);
72 1 STRUCTURE(
73 1 BYTE,
74 1 WORD,
75 1 BYTE);
76 1 STRUCTURE(
77 1 TYPE
78 1 VALUE
79 1 MACRO$LEV
80 1 CHAR (1)
81 1 TYPE
82 1 VALUE
83 1 SEGMENT
84 1 TYPE
85 1 STRUCTURE(
86 1 BYTE,
87 1 WORD,
88 1 BYTE);
89 1 STRUCTURE(
90 1 TYPE
91 1 VALUE
92 1 SEGMENT
93 1 TYPE
94 1 STRUCTURE(
95 1 BYTE,
96 1 WORD,
97 1 BYTE);
98 1 STRUCTURE(
99 1 TYPE
100 1 VALUE
101 1 SEGMENT
102 1 TYPE
103 1 STRUCTURE(
104 1 BYTE,
105 1 WORD,
106 1 BYTE);
107 1 STRUCTURE(
108 1 TYPE
109 1 VALUE
110 1 SEGMENT
111 1 TYPE
112 1 STRUCTURE(
113 1 BYTE,
114 1 WORD,
115 1 BYTE);
116 1 STRUCTURE(
117 1 TYPE
118 1 VALUE
119 1 SEGMENT
120 1 TYPE
121 1 STRUCTURE(
122 1 BYTE,
123 1 WORD,
124 1 BYTE);
125 1 STRUCTURE(
126 1 TYPE
127 1 VALUE
128 1 SEGMENT
129 1 TYPE
130 1 STRUCTURE(
131 1 BYTE,
132 1 WORD,
133 1 BYTE);
134 1 STRUCTURE(
135 1 TYPE
136 1 VALUE
137 1 SEGMENT
138 1 TYPE
139 1 STRUCTURE(
140 1 BYTE,
141 1 WORD,
142 1 BYTE);
143 1 STRUCTURE(
144 1 TYPE
145 1 VALUE
146 1 SEGMENT
147 1 TYPE
148 1 STRUCTURE(
149 1 BYTE,
150 1 WORD,
151 1 BYTE);
152 1 STRUCTURE(
153 1 TYPE
154 1 VALUE
155 1 SEGMENT
156 1 TYPE
157 1 STRUCTURE(
158 1 BYTE,
159 1 WORD,
160 1 BYTE);
161 1 STRUCTURE(
162 1 TYPE
163 1 VALUE
164 1 SEGMENT
165 1 TYPE
166 1 STRUCTURE(
167 1 BYTE,
168 1 WORD,
169 1 BYTE);
170 1 STRUCTURE(
171 1 TYPE
172 1 VALUE
173 1 SEGMENT
174 1 TYPE
175 1 STRUCTURE(
176 1 BYTE,
177 1 WORD,
178 1 BYTE);
179 1 STRUCTURE(
180 1 TYPE
181 1 VALUE
182 1 SEGMENT
183 1 TYPE
184 1 STRUCTURE(
185 1 BYTE,
186 1 WORD,
187 1 BYTE);
188 1 STRUCTURE(
189 1 TYPE
190 1 VALUE
191 1 SEGMENT
192 1 TYPE
193 1 STRUCTURE(
194 1 BYTE,
195 1 WORD,
196 1 BYTE);
197 1 STRUCTURE(
198 1 TYPE
199 1 VALUE
200 1 SEGMENT
201 1 TYPE
202 1 STRUCTURE(
203 1 BYTE,
204 1 WORD,
205 1 BYTE);
206 1 STRUCTURE(
207 1 TYPE
208 1 VALUE
209 1 SEGMENT
210 1 TYPE
211 1 STRUCTURE(
212 1 BYTE,
213 1 WORD,
214 1 BYTE);
215 1 STRUCTURE(
216 1 TYPE
217 1 VALUE
218 1 SEGMENT
219 1 TYPE
220 1 STRUCTURE(
221 1 BYTE,
222 1 WORD,
223 1 BYTE);
224 1 STRUCTURE(
225 1 TYPE
226 1 VALUE
227 1 SEGMENT
228 1 TYPE
229 1 STRUCTURE(
230 1 BYTE,
231 1 WORD,
232 1 BYTE);
233 1 STRUCTURE(
234 1 TYPE
235 1 VALUE
236 1 SEGMENT
237 1 TYPE
238 1 STRUCTURE(
239 1 BYTE,
240 1 WORD,
241 1 BYTE);
242 1 STRUCTURE(
243 1 TYPE
244 1 VALUE
245 1 SEGMENT
246 1 TYPE
247 1 STRUCTURE(
248 1 BYTE,
249 1 WORD,
250 1 BYTE);
251 1 STRUCTURE(
252 1 TYPE
253 1 VALUE
254 1 SEGMENT
255 1 TYPE
256 1 STRUCTURE(
257 1 BYTE,
258 1 WORD,
259 1 BYTE);
260 1 STRUCTURE(
261 1 TYPE
262 1 VALUE
263 1 SEGMENT
264 1 TYPE
265 1 STRUCTURE(
266 1 BYTE,
267 1 WORD,
268 1 BYTE);
269 1 STRUCTURE(
270 1 TYPE
271 1 VALUE
272 1 SEGMENT
273 1 TYPE
274 1 STRUCTURE(
275 1 BYTE,
276 1 WORD,
277 1 BYTE);
278 1 STRUCTURE(
279 1 TYPE
280 1 VALUE
281 1 SEGMENT
282 1 TYPE
283 1 STRUCTURE(
284 1 BYTE,
285 1 WORD,
286 1 BYTE);
287 1 STRUCTURE(
288 1 TYPE
289 1 VALUE
290 1 SEGMENT
291 1 TYPE
292 1 STRUCTURE(
293 1 BYTE,
294 1 WORD,
295 1 BYTE);
296 1 STRUCTURE(
297 1 TYPE
298 1 VALUE
299 1 SEGMENT
300 1 TYPE
301 1 STRUCTURE(
302 1 BYTE,
303 1 WORD,
304 1 BYTE);
305 1 STRUCTURE(
306 1 TYPE
307 1 VALUE
308 1 SEGMENT
309 1 TYPE
310 1 STRUCTURE(
311 1 BYTE,
312 1 WORD,
313 1 BYTE);
314 1 STRUCTURE(
315 1 TYPE
316 1 VALUE
317 1 SEGMENT
318 1 TYPE
319 1 STRUCTURE(
320 1 BYTE,
321 1 WORD,
322 1 BYTE);
323 1 STRUCTURE(
324 1 TYPE
325 1 VALUE
326 1 SEGMENT
327 1 TYPE
328 1 STRUCTURE(
329 1 BYTE,
330 1 WORD,
331 1 BYTE);
332 1 STRUCTURE(
333 1 TYPE
334 1 VALUE
335 1 SEGMENT
336 1 TYPE
337 1 STRUCTURE(
338 1 BYTE,
339 1 WORD,
340 1 BYTE);
341 1 STRUCTURE(
342 1 TYPE
343 1 VALUE
344 1 SEGMENT
345 1 TYPE
346 1 STRUCTURE(
347 1 BYTE,
348 1 WORD,
349 1 BYTE);
350 1 STRUCTURE(
351 1 TYPE
352 1 VALUE
353 1 SEGMENT
354 1 TYPE
355 1 STRUCTURE(
356 1 BYTE,
357 1 WORD,
358 1 BYTE);
359 1 STRUCTURE(
360 1 TYPE
361 1 VALUE
362 1 SEGMENT
363 1 TYPE
364 1 STRUCTURE(
365 1 BYTE,
366 1 WORD,
367 1 BYTE);
368 1 STRUCTURE(
369 1 TYPE
370 1 VALUE
371 1 SEGMENT
372 1 TYPE
373 1 STRUCTURE(
374 1 BYTE,
375 1 WORD,
376 1 BYTE);
377 1 STRUCTURE(
378 1 TYPE
379 1 VALUE
380 1 SEGMENT
381 1 TYPE
382 1 STRUCTURE(
383 1 BYTE,
384 1 WORD,
385 1 BYTE);
386 1 STRUCTURE(
387 1 TYPE
388 1 VALUE
389 1 SEGMENT
390 1 TYPE
391 1 STRUCTURE(
392 1 BYTE,
393 1 WORD,
394 1 BYTE);
395 1 STRUCTURE(
396 1 TYPE
397 1 VALUE
398 1 SEGMENT
399 1 TYPE
400 1 STRUCTURE(
401 1 BYTE,
402 1 WORD,
403 1 BYTE);
404 1 STRUCTURE(
405 1 TYPE
406 1 VALUE
407 1 SEGMENT
408 1 TYPE
409 1 STRUCTURE(
410 1 BYTE,
411 1 WORD,
412 1 BYTE);
413 1 STRUCTURE(
414 1 TYPE
415 1 VALUE
416 1 SEGMENT
417 1 TYPE
418 1 STRUCTURE(
419 1 BYTE,
420 1 WORD,
421 1 BYTE);
422 1 STRUCTURE(
423 1 TYPE
424 1 VALUE
425 1 SEGMENT
426 1 TYPE
427 1 STRUCTURE(
428 1 BYTE,
429 1 WORD,
430 1 BYTE);
431 1 STRUCTURE(
432 1 TYPE
433 1 VALUE
434 1 SEGMENT
435 1 TYPE
436 1 STRUCTURE(
437 1 BYTE,
438 1 WORD,
439 1 BYTE);
440 1 STRUCTURE(
441 1 TYPE
442 1 VALUE
443 1 SEGMENT
444 1 TYPE
445 1 STRUCTURE(
446 1 BYTE,
447 1 WORD,
448 1 BYTE);
449 1 STRUCTURE(
450 1 TYPE
451 1 VALUE
452 1 SEGMENT
453 1 TYPE
454 1 STRUCTURE(
455 1 BYTE,
456 1 WORD,
457 1 BYTE);
458 1 STRUCTURE(
459 1 TYPE
460 1 VALUE
461 1 SEGMENT
462 1 TYPE
463 1 STRUCTURE(
464 1 BYTE,
465 1 WORD,
466 1 BYTE);
467 1 STRUCTURE(
468 1 TYPE
469 1 VALUE
470 1 SEGMENT
471 1 TYPE
472 1 STRUCTURE(
473 1 BYTE,
474 1 WORD,
475 1 BYTE);
476 1 STRUCTURE(
477 1 TYPE
478 1 VALUE
479 1 SEGMENT
480 1 TYPE
481 1 STRUCTURE(
482 1 BYTE,
483 1 WORD,
484 1 BYTE);
485 1 STRUCTURE(
486 1 TYPE
487 1 VALUE
488 1 SEGMENT
489 1 TYPE
490 1 STRUCTURE(
491 1 BYTE,
492 1 WORD,
493 1 BYTE);
494 1 STRUCTURE(
495 1 TYPE
496 1 VALUE
497 1 SEGMENT
498 1 TYPE
499 1 STRUCTURE(
500 1 BYTE,
501 1 WORD,
502 1 BYTE);
503 1 STRUCTURE(
504 1 TYPE
505 1 VALUE
506 1 SEGMENT
507 1 TYPE
508 1 STRUCTURE(
509 1 BYTE,
510 1 WORD,
511 1 BYTE);
512 1 STRUCTURE(
513 1 TYPE
514 1 VALUE
515 1 SEGMENT
516 1 TYPE
517 1 STRUCTURE(
518 1 BYTE,
519 1 WORD,
520 1 BYTE);
521 1 STRUCTURE(
522 1 TYPE
523 1 VALUE
524 1 SEGMENT
525 1 TYPE
526 1 STRUCTURE(
527 1 BYTE,
528 1 WORD,
529 1 BYTE);
530 1 STRUCTURE(
531 1 TYPE
532 1 VALUE
533 1 SEGMENT
534 1 TYPE
535 1 STRUCTURE(
536 1 BYTE,
537 1 WORD,
538 1 BYTE);
539 1 STRUCTURE(
540 1 TYPE
541 1 VALUE
542 1 SEGMENT
543 1 TYPE
544 1 STRUCTURE(
545 1 BYTE,
546 1 WORD,
547 1 BYTE);
548 1 STRUCTURE(
549 1 TYPE
550 1 VALUE
551 1 SEGMENT
552 1 TYPE
553 1 STRUCTURE(
554 1 BYTE,
555 1 WORD,
556 1 BYTE);
557 1 STRUCTURE(
558 1 TYPE
559 1 VALUE
560 1 SEGMENT
561 1 TYPE
562 1 STRUCTURE(
563 1 BYTE,
564 1 WORD,
565 1 BYTE);
566 1 STRUCTURE(
567 1 TYPE
568 1 VALUE
569 1 SEGMENT
570 1 TYPE
571 1 STRUCTURE(
572 1 BYTE,
573 1 WORD,
574 1 BYTE);
575 1 STRUCTURE(
576 1 TYPE
577 1 VALUE
578 1 SEGMENT
579 1 TYPE
580 1 STRUCTURE(
581 1 BYTE,
582 1 WORD,
583 1 BYTE);
584 1 STRUCTURE(
585 1 TYPE
586 1 VALUE
587 1 SEGMENT
588 1 TYPE
589 1 STRUCTURE(
590 1 BYTE,
591 1 WORD,
592 1 BYTE);
593 1 STRUCTURE(
594 1 TYPE
595 1 VALUE
596 1 SEGMENT
597 1 TYPE
598 1 STRUCTURE(
599 1 BYTE,
600 1 WORD,
601 1 BYTE);
602 1 STRUCTURE(
603 1 TYPE
604 1 VALUE
605 1 SEGMENT
606 1 TYPE
607 1 STRUCTURE(
608 1 BYTE,
609 1 WORD,
610 1 BYTE);
611 1 STRUCTURE(
612 1 TYPE
613 1 VALUE
614 1 SEGMENT
615 1 TYPE
616 1 STRUCTURE(
617 1 BYTE,
618 1 WORD,
619 1 BYTE);
620 1 STRUCTURE(
621 1 TYPE
622 1 VALUE
623 1 SEGMENT
624 1 TYPE
625 1 STRUCTURE(
626 1 BYTE,
627 1 WORD,
628 1 BYTE);
629 1 STRUCTURE(
630 1 TYPE
631 1 VALUE
632 1 SEGMENT
633 1 TYPE
634 1 STRUCTURE(
635 1 BYTE,
636 1 WORD,
637 1 BYTE);
638 1 STRUCTURE(
639 1 TYPE
640 1 VALUE
641 1 SEGMENT
642 1 TYPE
643 1 STRUCTURE(
644 1 BYTE,
645 1 WORD,
646 1 BYTE);
647 1 STRUCTURE(
648 1 TYPE
649 1 VALUE
650 1 SEGMENT
651 1 TYPE
652 1 STRUCTURE(
653 1 BYTE,
654 1 WORD,
655 1 BYTE);
656 1 STRUCTURE(
657 1 TYPE
658 1 VALUE
659 1 SEGMENT
660 1 TYPE
661 1 STRUCTURE(
662 1 BYTE,
663 1 WORD,
664 1 BYTE);
665 1 STRUCTURE(
666 1 TYPE
667 1 VALUE
668 1 SEGMENT
669 1 TYPE
670 1 STRUCTURE(
671 1 BYTE,
672 1 WORD,
673 1 BYTE);
674 1 STRUCTURE(
675 1 TYPE
676 1 VALUE
677 1 SEGMENT
678 1 TYPE
679 1 STRUCTURE(
680 1 BYTE,
681 1 WORD,
682 1 BYTE);
683 1 STRUCTURE(
684 1 TYPE
685 1 VALUE
686 1 SEGMENT
687 1 TYPE
688 1 STRUCTURE(
689 1 BYTE,
690 1 WORD,
691 1 BYTE);
692 1 STRUCTURE(
693 1 TYPE
694 1 VALUE
695 1 SEGMENT
696 1 TYPE
697 1 STRUCTURE(
698 1 BYTE,
699 1 WORD,
700 1 BYTE);
701 1 STRUCTURE(
702 1 TYPE
703 1 VALUE
704 1 SEGMENT
705 1 TYPE
706 1 STRUCTURE(
707 1 BYTE,
708 1 WORD,
709 1 BYTE);
710 1 STRUCTURE(
711 1 TYPE
712 1 VALUE
713 1 SEGMENT
714 1 TYPE
715 1 STRUCTURE(
716 1 BYTE,
717 1 WORD,
718 1 BYTE);
719 1 STRUCTURE(
720 1 TYPE
721 1 VALUE
722 1 SEGMENT
723 1 TYPE
724 1 STRUCTURE(
725 1 BYTE,
726 1 WORD,
727 1 BYTE);
728 1 STRUCTURE(
729 1 TYPE
730 1 VALUE
731 1 SEGMENT
732 1 TYPE
733 1 STRUCTURE(
734 1 BYTE,
735 1 WORD,
736 1 BYTE);
737 1 STRUCTURE(
738 1 TYPE
739 1 VALUE
740 1 SEGMENT
741 1 TYPE
742 1 STRUCTURE(
743 1 BYTE,
744 1 WORD,
745 1 BYTE);
746 1 STRUCTURE(
747 1 TYPE
748 1 VALUE
749 1 SEGMENT
750 1 TYPE
751 1 STRUCTURE(
752 1 BYTE,
753 1 WORD,
754 1 BYTE);
755 1 STRUCTURE(
756 1 TYPE
757 1 VALUE
758 1 SEGMENT
759 1 TYPE
760 1 STRUCTURE(
761 1 BYTE,
762 1 WORD,
763 1 BYTE);
764 1 STRUCTURE(
765 1 TYPE
766 1 VALUE
767 1 SEGMENT
768 1 TYPE
769 1 STRUCTURE(
770 1 BYTE,
771 1 WORD,
772 1 BYTE);
773 1 STRUCTURE(
774 1 TYPE
775 1 VALUE
776 1 SEGMENT
777 1 TYPE
778 1 STRUCTURE(
779 1 BYTE,
780 1 WORD,
781 1 BYTE);
782 1 STRUCTURE(
783 1 TYPE
784 1 VALUE
785 1 SEGMENT
786 1 TYPE
787 1 STRUCTURE(
788 1 BYTE,
789 1 WORD,
790 1 BYTE);
791 1 STRUCTURE(
792 1 TYPE
793 1 VALUE
794 1 SEGMENT
795 1 TYPE
796 1 STRUCTURE(
797 1 BYTE,
798 1 WORD,
799 1 BYTE);
800 1 STRUCTURE(
801 1 TYPE
802 1 VALUE
803 1 SEGMENT
804 1 TYPE
805 1 STRUCTURE(
806 1 BYTE,
807 1 WORD,
808 1 BYTE);
809 1 STRUCTURE(
810 1 TYPE
811 1 VALUE
812 1 SEGMENT
813 1 TYPE
814 1 STRUCTURE(
815 1 BYTE,
816 1 WORD,
817 1 BYTE);
818 1 STRUCTURE(
819 1 TYPE
820 1 VALUE
821 1 SEGMENT
822 1 TYPE
823 1 STRUCTURE(
824 1 BYTE,
825 1 WORD,
826 1 BYTE);
827 1 STRUCTURE(
828 1 TYPE
829 1 VALUE
830 1 SEGMENT
831 1 TYPE
832 1 STRUCTURE(
833 1 BYTE,
834 1 WORD,
835 1 BYTE);
836 1 STRUCTURE(
837 1 TYPE
838 1 VALUE
839 1 SEGMENT
840 1 TYPE
841 1 STRUCTURE(
842 1 BYTE,
843 1 WORD,
844 1 BYTE);
845 1 STRUCTURE(
846 1 TYPE
847 1 VALUE
848 1 SEGMENT
849 1 TYPE
850 1 STRUCTURE(
851 1 BYTE,
852 1 WORD,
853 1 BYTE);
854 1 STRUCTURE(
855 1 TYPE
856 1 VALUE
857 1 SEGMENT
858 1 TYPE
859 1 STRUCTURE(
860 1 BYTE,
861 1 WORD,
862 1 BYTE);
863 1 STRUCTURE(
864 1 TYPE
865 1 VALUE
866 1 SEGMENT
867 1 TYPE
868 1 STRUCTURE(
869 1 BYTE,
870 1 WORD,
871 1 BYTE);
872 1 STRUCTURE(
873 1 TYPE
874 1 VALUE
875 1 SEGMENT
876 1 TYPE
877 1 STRUCTURE(
878 1 BYTE,
879 1 WORD,
880 1 BYTE);
881 1 STRUCTURE(
882 1 TYPE
883 1 VALUE
884 1 SEGMENT
885 1 TYPE
886 1 STRUCTURE(
887 1 BYTE,
888 1 WORD,
889 1 BYTE);
890 1 STRUCTURE(
891 1 TYPE
892 1 VALUE
893 1 SEGMENT
894 1 TYPE
895 1 STRUCTURE(
896 1 BYTE,
897 1 WORD,
898 1 BYTE);
899 1 STRUCTURE(
900 1 TYPE
901 1 VALUE
902 1 SEGMENT
903 1 TYPE
904 1 STRUCTURE(
905 1 BYTE,
906 1 WORD,
907 1 BYTE);
908 1 STRUCTURE(
909 1 TYPE
910 1 VALUE
911 1 SEGMENT
912 1 TYPE
913 1 STRUCTURE(
914 1 BYTE,
915 1 WORD,
916 1 BYTE);
917 1 STRUCTURE(
918 1 TYPE
919 1 VALUE
920 1 SEGMENT
921 1 TYPE
922 1 STRUCTURE(
923 1 BYTE,
924 1 WORD,
925 1 BYTE);
926 1 STRUCTURE(
927 1 TYPE
928 1 VALUE
929 1 SEGMENT
930 1 TYPE
931 1 STRUCTURE(
932 1 BYTE,
933 1 WORD,
934 1 BYTE);
935 1 STRUCTURE(
936 1 TYPE
937 1 VALUE
938 1 SEGMENT
939 1 TYPE
940 1 STRUCTURE(
941 1 BYTE,
942 1 WORD,
943 1 BYTE);
944 1 STRUCTURE(
945 1 TYPE
946 1 VALUE
947 1 SEGMENT
948 1 TYPE
949 1 STRUCTURE(
950 1 BYTE,
951 1 WORD,
952 1 BYTE);
953 1 STRUCTURE(
954 1 TYPE
955 1 VALUE
956 1 SEGMENT
957 1 TYPE
958 1 STRUCTURE(
959 1 BYTE,
960 1 WORD,
961 1 BYTE);
962 1 STRUCTURE(
963 1 TYPE
964 1 VALUE
965 1 SEGMENT
966 1 TYPE
967 1 STRUCTURE(
968 1 BYTE,
969 1 WORD,
970 1 BYTE);
971 1 STRUCTURE(
972 1 TYPE
973 1 VALUE
974 1 SEGMENT
975 1 TYPE
976 1 STRUCTURE(
977 1 BYTE,
978 1 WORD,
979 1 BYTE);
980 1 STRUCTURE(
981 1 TYPE
982 1 VALUE
983 1 SEGMENT
984 1 TYPE
985 1 STRUCTURE(
986 1 BYTE,
987 1 WORD,
988 1 BYTE);
989 1 STRUCTURE(
990 1 TYPE
991 1 VALUE
992 1 SEGMENT
993 1 TYPE
994 1 STRUCTURE(
995 1 BYTE,
996 1 WORD,
997 1 BYTE);
998 1 STRUCTURE(
999 1 TYPE
1000 1 VALUE
1001 1 SEGMENT
1002 1 TYPE
1003 1 STRUCTURE(
1004 1 BYTE,
1005 1 WORD,
1006 1 BYTE);
1007 1 STRUCTURE(
1008 1 TYPE
1009 1 VALUE
1010 1 SEGMENT
1011 1 TYPE
1012 1 STRUCTURE(
1013 1 BYTE,
1014 1 WORD,
1015 1 BYTE);
1016 1 STRUCTURE(
1017 1 TYPE
1018 1 VALUE
1019 1 SEGMENT
1020 1 TYPE
1021 1 STRUCTURE(
1022 1 BYTE,
1023 1 WORD,
1024 1 BYTE);
1025 1 STRUCTURE(
1026 1 TYPE
1027 1 VALUE
1028 1 SEGMENT
1029 1 TYPE
1030 1 STRUCTURE(
1031 1 BYTE,
1032 1 WORD,
1033 1 BYTE);
1034 1 STRUCTURE(
1035 1 TYPE
1036 1 VALUE
1037 1 SEGMENT
1038 1 TYPE
1039 1 STRUCTURE(
1040 1 BYTE,
1041 1 WORD,
1042 1 BYTE);
1043 1 STRUCTURE(
1044 1 TYPE
1045 1 VALUE
1046 1 SEGMENT
1047 1 TYPE
1048 1 STRUCTURE(
1049 1 BYTE,
1050 1 WORD,
1051 1 BYTE);
1052 1 STRUCTURE(
1053 1 TYPE
1054 1 VALUE
1055 1 SEGMENT
1056 1 TYPE
1057 1 STRUCTURE(
1058 1 BYTE,
1059 1 WORD,
1060 1 BYTE);
1061 1 STRUCTURE(
1062 1 TYPE
1063 1 VALUE
1064 1 SEGMENT
1065 1 TYPE
1066 1 STRUCTURE(
1067 1 BYTE,
1068 1 WORD,
1069 1 BYTE);
1070 1 STRUCTURE(
1071 1 TYPE
1072 1 VALUE
1073 1 SEGMENT
1074 1 TYPE
1075 1 STRUCTURE(
1076 1 BYTE,
1077 1 WORD,
1078 1 BYTE);
1079 1 STRUCTURE(
1080 1 TYPE
1081 1 VALUE
1082 1 SEGMENT
1083 1 TYPE
1084 1 STRUCTURE(
1085 1 BYTE,
1086 1 WORD,
1087 1 BYTE);
1088 1 STRUCTURE(
1089 1 TYPE
1090 1 VALUE
1091 1 SEGMENT
1092 1 TYPE
1093 1 STRUCTURE(
1094 1 BYTE,
1095 1 WORD,
1096 1 BYTE);
1097 1 STRUCTURE(
1098 1 TYPE
1099 1 VALUE
1100 1 SEGMENT
1101 1 TYPE
1102 1 STRUCTURE(
1103 1 BYTE,
1104 1 WORD,
1105 1 BYTE);
1106 1 STRUCTURE(
1107 1 TYPE
1108 1 VALUE
1109 1 SEGMENT
1110 1 TYPE
1111 1 STRUCTURE(
1112 1 BYTE,
1113 1 WORD,
1114 1 BYTE);
1115 1 STRUCTURE(
1116 1 TYPE
1117 1 VALUE
1118 1 SEGMENT
1119 1 TYPE
1120 1 STRUCTURE(
1121 1 BYTE,
1122 1 WORD,
1123 1 BYTE);
1124 1 STRUCTURE(
1125 1 TYPE
1126 1 VALUE
1127 1 SEGMENT
1128 1 TYPE
1129 1 STRUCTURE(
1130 1 BYTE,
1131 1 WORD,
1132 1 BYTE);
1133 1 STRUCTURE(
1134 1 TYPE
1135 1 VALUE
1136 1 SEGMENT
1137 1 TYPE
1138 1 STRUCTURE(
1139 1 BYTE,
1140 1 WORD,
1141 1 BYTE);
1142 1 STRUCTURE(
1143 1 TYPE
1144 1 VALUE
1145 1 SEGMENT
1146 1 TYPE
1147 1 STRUCTURE(
1148 1 BYTE,
1149 1 WORD,
1150 1 BYTE);
1151 1 STRUCTURE(
1152 1 TYPE
1153 1 VALUE
1154 1 SEGMENT
1155 1 TYPE
1156 1 STRUCTURE(
1157 1 BYTE,
1158 1 WORD,
1159 1 BYTE);
1160 1 STRUCTURE(
1161 1 TYPE
1162 1 VALUE
1163 1 SEGMENT
1164 1 TYPE
1165 1 STRUCTURE(
1166 1 BYTE,
1167 1 WORD,
1168 1 BYTE);
1169 1 STRUCTURE(
1170 1 TYPE
1171 1 VALUE
1172 1 SEGMENT
1173 1 TYPE
1174 1 STRUCTURE(
1175 1 BYTE,
1176 1 WORD,
1177 1 BYTE);
1178 1 STRUCTURE(
1179 1 TYPE
1180 1 VALUE
1181 1 SEGMENT
1182 1 TYPE
1183 1 STRUCTURE(
1184 1 BYTE,
1185 1 WORD,
1186 1 BYTE);
1187 1 STRUCTURE(
1188 1 TYPE
1189 1 VALUE
1190 1 SEGMENT
1191 1 TYPE
1192 1 STRUCTURE(
1193 1 BYTE,
1194 1 WORD,
1195 1 BYTE);
1196 1 STRUCTURE(
1197 1 TYPE
1198 1 VALUE
1199 1 SEGMENT
1200 1 TYPE
1201 1 STRUCTURE(
1202 1 BYTE,
1203 1 WORD,
1204 1 BYTE);
1205 1 STRUCTURE(
1206 1 TYPE
1207 1 VALUE
1208 1 SEGMENT
1209 1 TYPE
1210 1 STRUCTURE(
1211 1 BYTE,
1212 1 WORD,
1213 1 BYTE);
1214 1 STRUCTURE(
1215 1 TYPE
1216 1 VALUE
1217 1 SEGMENT
1218 1 TYPE
1219 1 STRUCTURE(
1220 1 BYTE,
1221 1 WORD,
1222 1 BYTE);
1223 1 STRUCTURE(
1224 1 TYPE
1225 1 VALUE
1226 1 SEGMENT
1227 1 TYPE
1228 1 STRUCTURE(
1229 1 BYTE,
1230 1 WORD,
1231 1 BYTE);
1232 1 STRUCTURE(
1233 1 TYPE
1234 1 VALUE
1235 1 SEGMENT
1236 1 TYPE
1237 1 STRUCTURE(
1238 1 BYTE,
1239 1 WORD,
1240 1 BYTE);
1241 1 STRUCTURE(
1242 1 TYPE
1243 1 VALUE
1244 1 SEGMENT
1245 1 TYPE
1246 1 STRUCTURE(
1247 1 BYTE,
1248 1 WORD,
1249 1 BYTE);
1250 1 STRUCTURE(
1251 1 TYPE
1252 1 VALUE
1253 1 SEGMENT
1254 1 TYPE
1255 1 STRUCTURE(
1256 1 BYTE,
1257 1 WORD,
1258 1 BYTE);
1259 1 STRUCTURE(
1260 1 TYPE
1261 1 VALUE
1262 1 SEGMENT
1263 1 TYPE
1264 1 STRUCTURE(
1265 1 BYTE,
1266 1 WORD,
1267 1 BYTE);
1268 1 STRUCTURE(
1269 1 TYPE
1270 1 VALUE
1271 1 SEGMENT
1272 1 TYPE
1273 1 STRUCTURE(
1274 1 BYTE,
1275 1 WORD,
1276 1 BYTE);
1277 1 STRUCTURE(
1278 1 TYPE
1279 1 VALUE
1280 1 SEGMENT
1281 1 TYPE
1282 1 STRUCTURE(
1283 1 BYTE,
1284 1 WORD,
1285 1 BYTE);
1286 1 STRUCTURE(
1287 1 TYPE
1288 1 VALUE
1289 1 SEGMENT
1290 1 TYPE
1291 1 STRUCTURE(
1292 1 BYTE,
1293 1 WORD,
1294 1 BYTE);
1295 1 STRUCTURE(
1296 1 TYPE
1297 1 VALUE
1298 1 SEGMENT
1299 1 TYPE
1300 1 STRUCTURE(
1301 1 BYTE,
1302 1 WORD,
1303 1 BYTE);
1304 1 STRUCTURE(
1305 1 TYPE
1306 1 VALUE
1307 1 SEGMENT
1308 1 TYPE
1309 1 STRUCTURE(
1310 1 BYTE,
1311 1 WORD,
1312 1 BYTE);
1313 1 STRUCTURE(
1314 1 TYPE
1315 1 VALUE
1316 1 SEGMENT
1317 1 TYPE
1318 1 STRUCTURE(
1319 1 BYTE,
1320 1 WORD,
1321 1 BYTE);
1322 1 STRUCTURE(
1323 1 TYPE
1324 1 VALUE
1325 1 SEGMENT
1326 1 TYPE
1327 1 STRUCTURE(
1328 1 BYTE,
1329 1 WORD,
1330 1 BYTE);
1331 1 STRUCTURE(
1332 1 TYPE
1333 1 VALUE
1334 1 SEGMENT
1335 1 TYPE
1336 1 STRUCTURE(
1337 1 BYTE,
1338 1 WORD,
1339 1 BYTE);
1340 1 STRUCTURE(
1341 1 TYPE
1342 1 VALUE
1343 1 SEGMENT
1344 1 TYPE
1345 1 STRUCTURE(
1346 1 BYTE,
1347 1 WORD,
1348 1 BYTE);
1349 1 STRUCTURE(
1350 1 TYPE
1351 1 VALUE
1352 1 SEGMENT
1353 1 TYPE
1354 1 STRUCTURE(
1355 1 BYTE,
1356 1 WORD,
1357 1 BYTE);
1358 1 STRUCTURE(
1359 1 TYPE
1360 1 VALUE
1361 1 SEGMENT
1362 1 TYPE
1363 1 STRUCTURE(
1364 1 BYTE,
1365 1 WORD,
1366 1 BYTE);
1367 1 STRUCTURE(
1368 1 TYPE
1369 1 VALUE
1370 1 SEGMENT
1371 1 TYPE
1372 1 STRUCTURE(
1373 1 BYTE,
1374 1 WORD,
1375 1 BYTE);
1376 1 STRUCTURE(
1377 1 TYPE
1378 1 VALUE
1379 1 SEGMENT
1380 1 TYPE
1381 1 STRUCTURE(
1382 1 BYTE,
1383 1 WORD,
1384 1 BYTE);
1385 1 STRUCTURE(
1386 1 TYPE
1387 1 VALUE
1388 1 SEGMENT
1389 1 TYPE
1390 1 STRUCTURE(
1391 1 BYTE,
1392 1 WORD,
1393 1 BYTE);
1394 1 STRUCTURE(
1395 1 TYPE
1396 1 VALUE
1397 1 SEGMENT
1398 1 TYPE
1399 1 STRUCTURE(
1400 1 BYTE,
1401 1 WORD,
1402 1 BYTE);
1403 1 STRUCTURE(
1404 1 TYPE
1405 1 VALUE
1406 1 SEGMENT
1407 1 TYPE
1408 1 STRUCTURE(
1409 1 BYTE,
1410 1 WORD,
1411 1 BYTE);
1412 1 STRUCTURE(
1413 1 TYPE
1414 1 VALUE
1415 1 SEGMENT
1416 1 TYPE
1417 1 STRUCTURE(
1418 1 BYTE,
1419 1 WORD,
1420 1 BYTE);
1421 1 STRUCTURE(
1422 1 TYPE
1423 1 VALUE
1424 1 SEGMENT
1425 1 TYPE
1426 1 STRUCTURE(
1427 1 BYTE,
1428 1 WORD,
1429 1 BYTE);
1430 1 STRUCTURE(
1431 1 TYPE
1432 1 VALUE
1433 1 SEGMENT
1434 1 TYPE
1435 1 STRUCTURE(
1436 1 BYTE,
1437 1 WORD,
1438 1 BYTE);
1439 1 STRUCTURE(
1440 1 TYPE
1441 1 VALUE
1442 1 SEGMENT
1443 1 TYPE
1444 1 STRUCTURE(
1445 1 BYTE,
1446 1 WORD,
1447 1 BYTE);
1448 1 STRUCTURE(
1449 1 TYPE
1450 1 VALUE
1451 1 SEGMENT
1452 1 TYPE
1453 1 STRUCTURE(
1454 1 BYTE,
1455 1 WORD,
1456 1 BYTE);
1457 1 STRUCTURE(
1458 1 TYPE
1459 1 VALUE
1460 1 SEGMENT
1461 1 TYPE
1462 1 STRUCTURE(
1463 1 BYTE,
1464 1 WORD,
1465 1 BYTE);
1466 1 STRUCTURE(
1467 1 TYPE
1468 1 VALUE
1469 1 SEGMENT
1470 1 TYPE
1471 1 STRUCTURE(
1472 1 BYTE,
1473 1 WORD,
1474 1 BYTE);
1475 1 STRUCTURE(
1476 1 TYPE
1477 1 VALUE
1478 1 SEGMENT
1479 1 TYPE
1480 1 STRUCTURE(
1481 1 BYTE,
1482 1 WORD,
1483 1 BYTE);
1484 1 STRUCTURE(
1485 1 TYPE
1486 1 VALUE
1487 1 SEGMENT
1488 1 TYPE
1489 1 STRUCTURE(
1490 1 BYTE,
1491 1 WORD,
1492 1 BYTE);
1493 1 STRUCTURE(
1494 1 TYPE
1495 1 VALUE
1496 1 SEGMENT
1497 1 TYPE
1498 1 STRUCTURE(
1499 1 BYTE,
1500 
```

```

72 1      LINE$TYPE
73 1      ERROR$TYPE
74 1      PARM1
75 1      PARM2
76 1      DECLARE I
77 1      DECLARE CHAR
78 1      DECLARE SYM$START
79 1      DECLARE EOF$FLAG
80 1      DECLARE ACTUAL$SYM$LNQ
81 1      DECLARE LINE$BUFF$2 (LINE$PLUS)
82 1      DECLARE LINE$BUFF (LINE$SIZE)
83 1      DECLARE CHAR$CNT
84 1      DECLARE RSV$WORD$TBL

      BYTE,
      BYTE,
      BYTE,
      BYTE);

      BYTE;
      BYTE;
      BYTE PUBLIC;
      BYTE PUBLIC;
      BYTE PUBLIC;

      BYTE PUBLIC;
      BYTE AT (.LINE$BUFF$2(5));
      BYTE;

      WORD EXTERNAL;

```

\$EJECT

/* This routine sets the index CHAR\$CNT that points into LINE BUFFER */

```
01 1  G$CHAR$INIT:          PUBLIC;
    PROCEDURE (NEW$CHAR$CNT)
02 2      DECLARE NEW$CHAR$CNT  BYTE;
03 2      CHAR$CNT = NEW$CHAR$CNT;
04 2      SYN$START = 0;
05 2      END G$CHAR$INIT;
```

/* This routine returns CHAR\$CNT */

```
206 1  G$CHAR$CNT:          BYTE PUBLIC;
    PROCEDURE
207 2      RETURN CHAR$CNT;
208 2      END G$CHAR$CNT;
```

```

00000000 $EJECT
00000001
00000002 /* This routine returns the next character in LINE BUFFER */
00000003
00000004 GET$CHAR:
00000005 PROCEDURE
00000006
00000007     BYTE PUBLIC;
00000008
00000009     DECLARE CHAR
00000010         BYTE;
00000011
00000012     CHAR$CNT = CHAR$CNT + 1;
00000013     CHAR = LINE$BUFF(CHAR$CNT);
00000014
00000015     IF (CHAR = EOF$FILE)
00000016     THEN RETURN EOF$FILE;
00000017
00000018     IF (CHAR$CNT > LAST(LINE$BUFF))
00000019     THEN RETURN CR;
00000020
00000021     IF (CHAR >= 'a') AND (CHAR <= 'z')
00000022     THEN RETURN CHAR AND ODFH;
00000023     ELSE RETURN CHAR;
00000024
00000025 END GET$CHAR;
00000026
```



```
REJECT
```

```
/* This routine puts back the last character in LINE BUFF */
```

```
1 1 PUT$BAK;  
PROCEDURE PUBLIC;
```

```
2 2 CHAR$CNT = CHAR$CNT - 1;
```

```
3 2 END PUT$BAK;
```

```
/* This routine puts back the last character into LINE BUFFER */
```

```
224 1 PUT$SYM$BAK;  
PROCEDURE PUBLIC;
```

```
225 2 CHAR$CNT = SYM$START;
```

```
226 2 END PUT$SYM$BAK;
```

*EJECT

```

17 1  GETSYM:
    PROCEDURE

    PUBLIC;

18 2  -GET$ASCII$STRING:
    PROCEDURE;
    /* This routine gets the identifier name in LINE BUFFER and places
       it into the SYMBOL BUFFER */

19 3      I = 0;
20 3      DO WHILE ((CHAR >= 'A') AND (CHAR <= 'Z'))
           OR ((CHAR >= 'O') AND (CHAR <= '9'))
           OR (CHAR = '@');
231 4      IF (I >= SYM$LENGTH)
           THEN DO;
233 5          DO WHILE (CHAR <> ' ') AND (CHAR <> CR) AND (CHAR <> EOFIL);
234 6              CHAR = GET$CHAR;
235 6              END;
236 5      ELSE DO;
237 4          INSTRUCTION.CHAR(I) = CHAR;
238 5          CHAR = GET$CHAR;
239 5          I = I + 1;
240 5      END;
241 5
242 4      END;

243 3      ACTUAL$SYM$LNQ = I;
244 3      DO WHILE I <= SYM$LENGTH;
245 4          INSTRUCTION.CHAR(I) = ' ';
246 4          I = I + 1;
247 4      END;
248 3      CALL PUTBAK;

249 3      END GET$ASCII$STRING;

250 2      CHAR = GET$CHAR;
251 2      DO WHILE CHAR = ' ';
252 3          CHAR = GET$CHAR;
253 3      END;
254 2      SYM$START = CHAR$CNT;
255 2      SYM$PTR = .SYM$BUFF;

/*      SKIP LINE SYMBOL      */

```

```

16 2      IF (CHAR = SKIP$STATEMENT)
17     THEN DO;
18         SYMTYPE = SKIP$LINE$SYM;
19         SKIP.LINE$TYPE = LINE$BUFF(1);
20         SKIP.ERROR$TYPE = LINE$BUFF(2);
21         SKIP.PARM1 = LINE$BUFF(3);
22         SKIP.PARM2 = LINE$BUFF(4);
23     END;
24 ELSE
25     RESERVED WORD OR IDENTIFIER */
26
27 4      IF (CHAR >= 'A') AND (CHAR <= 'Z')
28     THEN DO;
266         CALL GET$ASCII$STRING;
267         IF (SEARCH (SYMPTR, .RSV$WORD$TBL, ACTUAL$SYMBOL$NG) = SYM$EXIST)
268             THEN DO;
269             RETURN;
270         END;
271
272 3      /* IF ALL ABOVE TESTS FAIL SYMBOL MUST BE IDENTIFIER */
273 3      SYMTYPE = IDENT$SYM;
274
275 3      ELSE
276 2      IF (CHAR = '@')
277     THEN DO;
278         CALL GET$ASCII$STRING;
279         SYMTYPE = IDENT$SYM;
280     END;
281 ELSE
282     NUMBER SYMBOL */
283
284 2      IF (CHAR >= '0') AND (CHAR <= '9')
285     THEN DO;
286         SYMTYPE = NUMBER$SYM;
287         I = 0;
288         DO WHILE ((CHAR >= '0') AND (CHAR <= '9')) OR
289             ((CHAR >= 'A') AND (CHAR <= 'F')) AND
290             (I < NUMBER$NLG);
291             NUMERIC.CHAR(I) = CHAR;
292             I = I + 1;
293             CHAR = GET$CHAR;
294         END;
295         NUMERIC.CHAR(I) = CHAR;
296         IF (CHAR <> 'H') AND (CHAR <> 'D') AND (CHAR <> 'B') AND

```

```

00000000 (CHAR <> 'D') AND (CHAR <> 'Q')
00000001 THEN DO;
00000002     CALL PUT$BAK;
00000003     END;
00000004     ELSE I = I + 1;
00000005 DO WHILE (I < NUMBER$LNQ);
00000006     NUMERIC.CHAR(I) = ' ';
00000007     I = I + 1;
00000008 END;
00000009 NUMERIC.VALUE = NUM$IN(. NUMERIC.CHAR);
00000010 ELSE
00000011
00000012 /* PERIOD */
00000013 IF (CHAR = '.')
00000014 THEN DO;
00000015     SYM$TYPE = PERIOD$SYM;
00000016     END;
00000017 ELSE
00000018
00000019 /* LOGICAL EQUAL */
00000020 IF (CHAR = '=')
00000021 THEN DO;
00000022     SYM$TYPE = EQ$SYM;
00000023     INSTRUCTION.INST = CONDITION$SYM;
00000024     END;
00000025 ELSE
00000026
00000027 /* COMMENT SYMBOL */
00000028 IF (CHAR = ';')
00000029 THEN DO;
00000030     SYM$TYPE = COMMENT$SYM;
00000031     END;
00000032 ELSE
00000033
00000034 /* COMMA SYMBOL */
00000035 IF (CHAR = ',')
00000036 THEN DO;
00000037     SYM$TYPE = COMMA$SYM;
00000038     END;
00000039 ELSE
00000040
00000041 /* LOCATION DEFINITION */

```

```

16 2      IF (CHAR = ':')
18 3      THEN DO;
19 3          SYM$TYPE = COLON$SYM;
          END;
      ELSE
          /*
20 2          LESS$THAN OR LESS$THAN$OR$EQUAL */
          IF (CHAR = '<')
22 3          THEN DO;
23 3              CHAR = GET$CHAR;
                IF (CHAR = '=')
25 4                  THEN DO;
26 4                      SYM$TYPE = LESS$SYM;
27 3                      END;
28 3                  ELSE DO;
29 4                      SYM$TYPE = LT$SYM;
30 4                      CALL PUT$BAK;
31 3                      END;
32 3                      INSTRUCTION.INST = CONDITION$SYM;
          END;
      ELSE
          /*
333 2          GREATER$THAN OR GREATER$THAN$OR$EQUAL OR NOT$EQUAL */
          IF (CHAR = '>')
335 3          THEN DO;
336 3              CHAR = GET$CHAR;
                IF (CHAR = '=')
338 4                  THEN DO;
339 4                      SYM$TYPE = GE$SYM;
340 3                      END;
                ELSE IF (CHAR = '<')
342 4                  THEN DO;
343 4                      SYM$TYPE = NE$SYM;
344 3                      END;
                ELSE DO;
345 4                      SYM$TYPE = GT$SYM;
346 4                      CALL PUT$BAK;
347 4                      END;
                INSTRUCTION.INST = CONDITION$SYM;
          END;
      ELSE
          /*
348 3          CONTROL LINE SYMBOL OR CURRENT LOCATION COUNTER VALUE */

```

```

M-80 COMPILER      LEXICAL ANALYZER

002  IF (CHAR = '$')
003  THEN IF CHAR%CNT = 0
004  THEN DO,
005  CHAR = GET$CHAR;
006  CALL GET$ASCII$STRING;
007  IF (SEARCH (SYM$PTR, . RSV$WORD$TBL, ACTUAL$SYM$LNG)
008  = SYM$UNDEF) OR (SYM$TYPE <> CONTROL$SYM)
009  THEN SYM$TYPE = SYM$UNDEF;
010  END;
011  ELSE DO;
012  SYM$TYPE = LOCATION$CNT$SYM;
013  CURRENT$LOC.VALUE = LOCATION$CNT;
014  CURRENT$LOC.SEGMENT = SEGMENT$TYPE;
015  END;
016  ELSE
017  PLUS SYMBOL */
018  IF (CHAR = '+' )
019  THEN DO;
020  SYM$TYPE = PLUS$SYM;
021  END;
022  ELSE
023  MINUS SYMBOL */
024  IF (CHAR = '-' )
025  THEN DO;
026  SYM$TYPE = MINUS$SYM;
027  END;
028  ELSE
029  MULTI. SYMBOL */
030  IF (CHAR = '*' )
031  THEN DO;
032  SYM$TYPE = MULT$SYM;
033  END;
034  ELSE
035  DIVI. SYMBOL */
036  IF (CHAR = '/' )
037  THEN DO;
038  SYM$TYPE = DIVI$SYM;
039  END;
040  ELSE

```

```
379 2 /* QUOTE SYMBOL */
380 3 IF (CHAR = ''')
381 3 THEN DO, SYM$TYPE = QUOTE$SYM,
382 3 END,
ELSE
/* PARENTHESIS SYMBOLS */
383 2 IF (CHAR = '(')
384 3 THEN DO, SYM$TYPE = LPAREN$SYM,
385 3 END,
ELSE
386 3 IF (CHAR = ')')
387 2 THEN DO, SYM$TYPE = RPAREN$SYM,
388 3 END,
ELSE
389 3 /* END OF LINE SYMBOL */
390 3
391 2 IF (CHAR = CR)
392 3 THEN DO, SYM$TYPE = EOL$SYM,
393 3 END,
ELSE
394 3 /* END OF FILE SYMBOL */
395 2 IF (CHAR = EOF$FILE)
396 3 THEN DO, SYM$TYPE = EOF$SYM,
397 3 END,
ELSE
398 3 /* SYMBOL IS UNDEFINED */
399 2 SYM$TYPE = SYM$UNDEF,
400 2 END GET$SYM,
401 1 END GET$SYMBOL;
```

MODULE INFORMATION:

CODE AREA SIZE = 047FH 1151D
 VARIABLE AREA SIZE = 008EH 142D
 MAXIMUM STACK SIZE = 0006H 6D
 704 LINES READ
 0 PROGRAM ERROR(S)

) OF PL/M-80 COMPILATION

SIS-II PL/M-80 V3.1 COMPILATION OF MODULE CONVERSION
 O OBJECT MODULE REQUESTED
 DMPILER INVOKED BY: PLM80 CONVER.PLM DATE(10-MAR-82) NOOBJECT PRINT(.F1:CONVER.LST) PAGELENGTH(50)

```
1      $TITLE('NUMIN-ASCII TO BINARY, NUMOUT-BINARY TO ASCII CONVERSIONS')
      CONVERSION:
      DO,
```

```
2  1      DECLARE REV810712          BYTE AT(O);
      /*      MODERFIED BY JOSEPH R. GARAPPOLO    06 JUNE 1981
      INCLUDE FILES - INC      ELD
```

DESCRIPTION:

This module includes two routines, on converts binary to ascii, and the
 other converts ascii integers in hex, octal, binary and decimal to binary.

*/

\$NOLIST

```

22 1 $EJECT
23 1 DECLARE DIGITS (*)
24 1 DECLARE BASE2
25 1 DECLARE BASE8
26 1 DECLARE BASE10
27 1 DECLARE BASE16
28 1 DECLARE I
29 1 NUMIN:
30 1 PROCEDURE (BUFF$PTR)
31 1 WORD PUBLIC;
32 1
33 1 /* PARAMETERS: pointer to buffer that contains integer represented in ascii
34 1 DESCRIPTIONS:
35 1
36 1 This routine converts ASCII to binary. The ASCII may represent integers
37 1 written in hex, oct, binary and decimal depending on the character
38 1 terminating the string.
39 1 H - hex.
40 1 B - octal
41 1 Y or Q - binary
42 1 D or character other than H,B,Y or Q - decimal
43 1
44 1 */
45 1 DECLARE BUFF$PTR
46 1 DECLARE (CHAR BASED BUFF$PTR)
47 1 DECLARE ASCII$DIGIT
48 1
49 1 BASE2 = 0;
50 1 BASE8 = 0;
51 1 BASE10 = 0;
52 1 BASE16 = 0;
53 1 ASCII$DIGIT = TRUE;
54 1 DO WHILE ASCII$DIGIT;
55 1 ASCII$DIGIT = FALSE;
56 1 DO I = 0 TO LAST(DIGITS);
57 1 IF CHAR = DIGITS(I)
58 1 THEN DO;
59 1 IF (I < 2)
60 1 THEN BASE2 = BASE2 + BASE2 + I;
61 1 IF (I < 8)
62 1 THEN BASE8 = SHL(BASE8,3) + I;
63 1 IF (I < 10)
64 1 THEN BASE10 = BASE10 * 10 + I;
65 1 BASE16 = SHL(BASE16,4) + I;
66 1 BUFF$PTR = BUFF$PTR + 1;

```

```
50 5          ASCII$DIGIT = TRUE;
51 5          END;
52 4          END;
53 3          END;
54 2          /* return value depending on terminating character */
55 2          IF (CHAR = 'H')
56 2              THEN RETURN BASE16;
57 2          IF (CHAR = 'O') OR (CHAR = 'Q')
58 2              THEN RETURN BASE8;
59 2          IF (CHAR = 'Y')
60 2              THEN RETURN BASE2;
61 2          RETURN BASE10;
62 2          END NUMIN;
```

```

$EJECT
62 1  NUMOUT:
    PROCEDURE (VALUE, BASE, BUFF$PTR, WIDTH, PAD) PUBLIC;
/*  PARAMETERS:  VALUE - 16 bit value to be converted
                  BASE - radix of ASCII
                  BUFF$PTR - pointer to buffer where ASCII output is to go
                  WIDTH - width of output field
                  PAD - character to pad field not used by ASCII output

DESCRIPTION:
    This routine converts a binary number to ASCII of the specified radix.

*/
63 2  DECLARE VALUE
64 2  DECLARE BASE
65 2  DECLARE BUFF$PTR
66 2  DECLARE WIDTH
67 2  DECLARE PAD
68 2  DECLARE (CHAR BASED BUFF$PTR)(1) BYTE;
69 2  DO I = 1 TO WIDTH;
70 3  CHAR(WIDTH - I) = DIGITS(VALUE MOD BASE);
71 3  VALUE = VALUE / BASE;
72 3  END;
73 2  I = 0;
74 2  DO WHILE (CHAR(I) = '0') AND (I < WIDTH-1);
75 3  CHAR(I) = PAD;
76 3  I = I + 1;
77 3  END;
78 2  END NUMOUT;
79 1  END CONVERSION;

```

MODULE INFORMATION:

CODE AREA SIZE	= 01A1H	417D
VARIABLE AREA SIZE	= 0013H	19D
MAXIMUM STACK SIZE	= 0006H	6D

L/M-80 COMPILER NUMIN-ASCII TO BINARY, NUMOUT-BINARY TO ASCII CONVERSIONS

10-MAR-82 PAGE 5

151 LINES READ
0 PROGRAM ERROR(S)

END OF PL/M-80 COMPILATION

SIS-II PL/M-80 V3.1 COMPILATION OF MODULE TABLESEARCH
 O OBJECT MODULE REQUESTED

OMPILER INVOKED BY: PLM80 SEARCH.PLM DATE(10-MAR-82) NOOBJECT PRINT(:F1:SEARCH.LST) PAGELNGTH(50)

1 \$TITLE('SEARCH THROUGH STATIC TABLE')

TABLE\$SEARCH:

DO,

2 1 DECLARE REV810610 BYTE AT(0);

/*

WRITTEN BY JOSEPH R. GARAPPOLO 4 AUG 80

INCLUDED FILE - :F1:INC.ELD

- :F1:SYSTEM.ELD

- :F1:SMATCH.EPD

DESCRIPTION:

This module contains the routines used to perform reserved word searches.
 SEARCH performs a linear or binary search depending on the number of
 reserved words in the particular reserved word record.

SMABO has eight (8) reserved word tables, one for each of the eight
 different length reserved words can have. The field which holds the
 name of the reserved word varies depending on the length of the reserved
 words found in a particular table. For this reason the records in a
 reserved word table can not be accessed by a simple index, the address
 of the record is calculated and used to reference the record.

*/

\$NOLIST

```

174 1 1      DECLARE LINEAR$SEARCH$LIMIT      LITERALLY '30';
175 1 1      SEARCH:
176 2 2      PROCEDURE(SYM$PTR, CNTL$TBL$PTR, LNG) BYTE PUBLIC;
177 2 2      DECLARE SYM$PTR      POINTER;
178 2 2      DECLARE CNTL$TBL$PTR  POINTER;
179 2 2      DECLARE LNG      BYTE;
180 2 2      DECLARE (CNTL$TBL BASED CNTL$TBL$PTR)(1) POINTER;
181 2 2      DECLARE (SYM$BUFF BASED SYM$PTR)  STRUCTURE(
182 2 2      ENTRY$1      WORD,
183 2 2      ENTRY$2      WORD,
184 2 2      ENTRY$3      WORD,
185 2 2      CHAR (1)    BYTE);
186 2 2      DECLARE RSV$TBL$PTR      POINTER;
187 2 2      DECLARE (TBL$DESC BASED RSV$TBL$PTR) STRUCTURE(
188 2 2      NO$OF$RECORDS  BYTE,
189 2 2      CHAR$LNG      BYTE);
190 2 2      DECLARE (STATIC$TBL BASED RSV$TBL$PTR) STRUCTURE(
191 2 2      ENTRY$1      WORD,
192 2 2      ENTRY$2      WORD,
193 2 2      ENTRY$3      WORD,
194 2 2      CHAR (1)    BYTE);
195 2 2      DECLARE TABLE$BASE      POINTER;
196 2 2      DECLARE LOW$INDEX      BYTE;
197 2 2      DECLARE HIGH$INDEX     BYTE;
198 2 2      DECLARE CHAR$LNG      BYTE;
199 2 2      DECLARE MATCH$STATUS     BYTE;
200 2 2      DECLARE J

```

```

$EJECT
190 2      BINARY$SEARCH:
      PROCEDURE
191 3      DO WHILE (LOW$INDEX <= HIGH$INDEX);
      /* calculate address of reserved word record which is to
      checked for a match */
192 4      J = ((LOW$INDEX + HIGH$INDEX) / 2);
193 4      RSV$TBL$PTR = TABLE$BASE + (J * ((SIZE(STATIC$TBL) - 1) + CHAR$LNG));
194 4      MATCH$STATUS = STRING$MATCH(.SYN$BUFF.CHAR, .STATIC$TBL.CHAR,
      CHAR$LNG);
195 4      IF (MATCH$STATUS = 0$THAN)
      THEN LOW$INDEX = J + 1;
      ELSE
197 4      IF (MATCH$STATUS = L$THAN)
      THEN HIGH$INDEX = J - 1;
      ELSE DO; /* we have a match */
200 5          SYN$BUFF.ENTRY$1 = STATIC$TBL.ENTRY$1;
201 5          SYN$BUFF.ENTRY$2 = STATIC$TBL.ENTRY$2;
202 5          SYN$BUFF.ENTRY$3 = STATIC$TBL.ENTRY$3;
203 5          RETURN SYN$EXSIST;
204 5      END;
205 4      END;

/* symbol does not exist */
206 3      RETURN SYN$UNDEF;
207 3      END BINARY$SEARCH;

```



```

$EJECT
208 2 LINEAR$SEARCH:
    PROCEDURE
        BYTE;
209 3 DO WHILE (LOW$INDEX <= HIGH$INDEX);
210 4 /* calculate address of reserved word record */
    RSV$TBL$PTR = TABLE$BASE + (LOW$INDEX * (SIZE(STATIC$TBL) - 1)
    + CHAR$LNG));
211 4 MATCH$STATUS = STRING$MATCH(.SYM$BUFF CHAR, STATIC$TBL CHAR,
    CHAR$LNG);
212 4 IF (MATCH$STATUS = MATCH)
    THEN DO,
214 5     SYM$BUFF.ENTRY$1 = STATIC$TBL.ENTRY$1;
215 5     SYM$BUFF.ENTRY$2 = STATIC$TBL.ENTRY$2;
216 5     SYM$BUFF.ENTRY$3 = STATIC$TBL.ENTRY$3;
217 5     RETURN SYM$EXSIST,
218 5     END;
219 4 ELSE LOW$INDEX = LOW$INDEX + 1;
220 4 END;

/* no match */
221 3 RETURN SYM$UNDEF;
222 3 END LINEAR$SEARCH;

/* BEGIN SEARCH */
223 2 IF (LNG > CNTL$TBL(0))
    THEN RETURN SYM$UNDEF;
225 2 RSV$TBL$PTR = CNTL$TBL(LNG);
226 2 LOW$INDEX = 1;
227 2 HIGH$INDEX = TBL$DESC.NO$OF$RECORDS;
228 2 CHAR$LNG = TBL$DESC.CHAR$LNG;
229 2 TABLE$BASE = RSV$TBL$PTR + 2;
230 2 IF (HIGH$INDEX > LINEAR$SEARCH$LIMIT)
    THEN RETURN BINARY$SEARCH;
    ELSE RETURN LINEAR$SEARCH;
232 2
233 2 END SEARCH;
234 1 END TABLE$SEARCH,

```

MODULE INFORMATION:

CODE AREA SIZE	= 018EH	398D
VARIABLE AREA SIZE	= 000EH	14D
MAXIMUM STACK SIZE	= 0006H	6D
402 LINES READ		
0 PROGRAM ERROR(S)		

END OF PL/M-80 COMPILATION

8-II PL/M-80 V3.1 COMPILATION OF MODULE RBVWORDTABLE

ECT MODULE PLACED IN : F4:RBVTBL.OBJ

PIPER INVOKED BY: PLM80 RBVTBL.PLM DATE(28-MAR-82) OBJECT(:F4:RBVTBL.OBJ) PRINT(:F1:RBVTBL.LST) PAGELENGTH(50)

1 RBVWORD\$TABLE:
DO,

2 1 DECLARE REV820130 BYTE AT (0),

/* WRITTEN BY JOSEPH R. GARAPPOLO 28 APRIL 1981

INCLUDE FILES - INC .ELD
SYSTEM.ELD

DESCRIPTION:

This module contains the static tables for the Reserved Words

*/
\$NOLIST

```
$EJECT
      1 1  DECLARE MAX$NUM$CHARS      LITERALLY '08',
/* Reserved Word Control Table
   This table contains the pointers to the reserved word tables with
   the different length Reserved Words.
*/
      2 1  DECLARE RSV$WORD$TBL (*)   POINTER PUBLIC DATA(
      MAX$NUM$CHARS,
      .RSV$WORD$LN$1,
      .RSV$WORD$LN$2,
      .RSV$WORD$LN$3,
      .RSV$WORD$LN$4,
      .RSV$WORD$LN$5,
      .RSV$WORD$LN$6$TO$8,
      .RSV$WORD$LN$6$TO$8,
      .RSV$WORD$LN$6$TO$8);
```

/* Reserved Word Table for RW's with one character #/

```

3 1 DECLARE RSV$WORD$LN$1 (*)
      NULL, NULL, NULL, NULL, NULL, NULL,
      REG$SYM, A$REQ, 10000000B, 00111111B, 10000000B, 'A',
      REG$SYM, B$REQ, 00000000B, 00000000B, 00000000B, 'B',
      REG$SYM, C$REQ, 10000000B, 00001001B, 10000000B, 'C',
      REG$SYM, D$REQ, 00010000B, 00001010B, 00010000B, 'D',
      REG$SYM, E$REQ, 10000000B, 00011011B, 10010000B, 'E',
      REG$SYM, H$REQ, 10000000B, 00100100B, 10100000B, 'H',
      REG$SYM, L$REQ, 10000000B, 00101101B, 10100000B, 'L',
      REG$SYM, M$REQ, 10000000B, 00110110B, 10000000B, 'M'

```

\$EJECT

/* Reserved Word Table for RW's with length of two */

```

74 1  DECLARE REV$WORD$LN$2 (*)      BYTE DATA(31,2,
NULL      ,NULL      ,NULL      ,NULL      ,NULL
BY$SYM    ,0          ,0          ,0          ,0
INSTRUCTION$SYM ,8          ,0DCH      ,0          ,0
INSTRUCTION$SYM ,9          ,0FCH      ,3          ,3
INSTRUCTION$SYM ,14         ,0F4H      ,3          ,3
INSTRUCTION$SYM ,19         ,0CCH      ,3          ,3
DATA$STORAGE$SYM ,DB$SYM    ,000H      ,0          ,0
INSTRUCTION$SYM ,24         ,0F3H      ,1          ,1
DO$SYM      ,00          ,0          ,0          ,0
DATA$STORAGE$SYM ,DB$SYM    ,0          ,0          ,0
DATA$STORAGE$SYM ,DW$SYM    ,0          ,0          ,0
INSTRUCTION$SYM ,25         ,0FDH      ,1          ,1
EG$SYM      ,CONDITION$SYM ,0          ,0          ,0
Q$SYM       ,CONDITION$SYM ,0          ,0          ,0
QT$SYM      ,CONDITION$SYM ,0          ,0          ,0
STATEMENT$SYM ,IF$SYM     ,0          ,0          ,0
INSTRUCTION$SYM ,27         ,0DBH      ,2          ,2
INSTRUCTION$SYM ,30         ,0DAH      ,3          ,3
INSTRUCTION$SYM ,31         ,0FAH      ,3          ,3
INSTRUCTION$SYM ,35         ,0F2H      ,3          ,3
INSTRUCTION$SYM ,38         ,0CAH      ,3          ,3
LE$SYM       ,CONDITION$SYM ,0          ,0          ,0
LT$SYM       ,CONDITION$SYM ,0          ,0          ,0
NE$SYM       ,CONDITION$SYM ,0          ,0          ,0
OR$SYM       ,0          ,0          ,0          ,0
INSTRUCTION$SYM ,54         ,0DSH      ,1          ,1
INSTRUCTION$SYM ,58         ,0FSH      ,1          ,1
INSTRUCTION$SYM ,61         ,0FOH      ,1          ,1
INSTRUCTION$SYM ,66         ,0CSH      ,1          ,1
REG$SYM      ,SP$REG     ,10000000B ,00110000B
TO$SYM       ,0          ,0          ,0          ,0
          , , 'BY',
          , , 'CC',
          , , 'CM',
          , , 'CP',
          , , 'CZ',
          , , 'DB',
          , , 'DI',
          , , 'DO',
          , , 'DS',
          , , 'DW',
          , , 'EI',
          , , 'EG',
          , , 'GE',
          , , 'GT',
          , , 'IF',
          , , 'IN',
          , , 'JC',
          , , 'JM',
          , , 'JP',
          , , 'JZ',
          , , 'LE',
          , , 'LT',
          , , 'NE',
          , , 'OR',
          , , 'RC',
          , , 'RM',
          , , 'RP',
          , , 'RZ',
          , , 'SP',
          , , 'TO',

```

\$EJECT

/* Reserved Word Table for RW's with three characters */

```

5 1  DECLARE RV$WORD$LN$3 (*)      BYTE DATA(65,3,
NULL                                ,0
INSTRUCTION$SYM ,01              ,09
INSTRUCTION$SYM ,02              ,09
INSTRUCTION$SYM ,03              ,03
INSTRUCTION$SYM ,04              ,09
INSTRUCTION$SYM ,05              ,03
AND$SYM         ,0               ,0
INSTRUCTION$SYM ,06              ,09
INSTRUCTION$SYM ,10              ,01
INSTRUCTION$SYM ,11              ,01
INSTRUCTION$SYM ,12              ,03
INSTRUCTION$SYM ,13              ,10
INSTRUCTION$SYM ,15              ,10
INSTRUCTION$SYM ,16              ,09
INSTRUCTION$SYM ,17              ,10
INSTRUCTION$SYM ,18              ,10
INSTRUCTION$SYM ,20              ,01
INSTRUCTION$SYM ,21              ,04
INSTRUCTION$SYM ,22              ,02
INSTRUCTION$SYM ,23              ,04
END$SYM         ,0               ,0
EGU$SYM         ,0               ,0
STATEMENT$SYM   ,FOR$SYM        ,0
INSTRUCTION$SYM ,26              ,01
INSTRUCTION$SYM ,28              ,02
INSTRUCTION$SYM ,29              ,04
INSTRUCTION$SYM ,32              ,10
INSTRUCTION$SYM ,33              ,10
INSTRUCTION$SYM ,34              ,10
INSTRUCTION$SYM ,36              ,10
INSTRUCTION$SYM ,37              ,10
INSTRUCTION$SYM ,39              ,10
INSTRUCTION$SYM ,42              ,11
INSTRUCTION$SYM ,43              ,06
INSTRUCTION$SYM ,44              ,08
INSTRUCTION$SYM ,45              ,01
NOT$SYM         ,0               ,0
INSTRUCTION$SYM ,46              ,03
MEMORY$SYM      ,ORG$SYM        ,0
INSTRUCTION$SYM ,47              ,0
INSTRUCTION$SYM ,48              ,09
INSTRUCTION$SYM ,50              ,07
                                ,0
                                ,0CEH ,2
                                ,08BH ,1
                                ,050H ,1
                                ,0C6H ,2
                                ,0A0H ,1
                                ,0      ,0
                                ,0E6H ,2
                                ,02FH ,1
                                ,03FH ,1
                                ,08BH ,1
                                ,0C4H ,3
                                ,0ECH ,3
                                ,0FEH ,2
                                ,0E4H ,3
                                ,0      ,0
                                ,027H ,1
                                ,009H ,1
                                ,005H ,1
                                ,00BH ,1
                                ,0      ,0
                                ,0      ,0
                                ,0      ,0
                                ,076H ,1
                                ,005H ,1
                                ,003H ,1
                                ,0C3H ,3
                                ,0D2H ,3
                                ,0C2H ,3
                                ,0EAH ,3
                                ,0E2H ,3
                                ,03AH ,3
                                ,001H ,3
                                ,040H ,1
                                ,006H ,2
                                ,000H ,1
                                ,0      ,0
                                ,0B0H ,1
                                ,0      ,0
                                ,0F6H ,2
                                ,0D3H ,2
                                ,0C1H ,1
                                ,
                                ,
                                ,ACI',
                                ,ADC',
                                ,ADD',
                                ,ADI',
                                ,ANA',
                                ,AND',
                                ,ANI',
                                ,CMA',
                                ,CMC',
                                ,CMP',
                                ,CNZ',
                                ,CPE',
                                ,CPI',
                                ,CPO',
                                ,CIF',
                                ,DAA',
                                ,DAD',
                                ,DCR',
                                ,DCX',
                                ,END',
                                ,EGU',
                                ,FOR',
                                ,HLT',
                                ,INR',
                                ,INX',
                                ,JMP',
                                ,JNC',
                                ,JNZ',
                                ,JPE',
                                ,JPO',
                                ,LDA',
                                ,LXI',
                                ,MOV',
                                ,MVI',
                                ,NOP',
                                ,NOT',
                                ,ORA',
                                ,ORG',
                                ,ORI',
                                ,OUT',
                                ,POP',

```

,'XRI');

\$EJECT

/* Reserved Word Table for RW's with four characters */

```

76 1  DECLARE RSV$WORD$LN$4 (*)      BYTE DATA(18,4,
    NULL      ,NULL      ,NULL      ,NULL
MEMORY$SYN   ,ASEQ$SYN   ,0        ,0
INSTRUCTION$SYN ,07      ,10      ,0CDH ,3
STATEMENT$SYN ,CASE$SYN   ,0        ,0
CONTROL$SYN    ,STATEMENT$CODE$SYN,0        ,0
MEMORY$SYN     ,CSEQ$SYN   ,0        ,0
MEMORY$SYN     ,DSEQ$SYN   ,0        ,0
ELSE$SYN       ,0         ,0        ,0
INSTRUCTION$SYN ,40      ,09      ,00AH ,1
INSTRUCTION$SYN ,41      ,10      ,02AH ,3
CONTROL$SYN     ,LIST$SYN   ,0        ,0
INSTRUCTION$SYN ,49      ,01      ,0E9H ,1
INSTRUCTION$SYN ,51      ,07      ,0C5H ,1
INSTRUCTION$SYN ,69      ,10      ,022H ,3
INSTRUCTION$SYN ,71      ,01      ,0F9H ,1
INSTRUCTION$SYN ,73      ,05      ,002H ,1
THEN$SYN        ,0         ,0        ,0
INSTRUCTION$SYN ,77      ,01      ,0EBH ,1
INSTRUCTION$SYN ,80      ,01      ,0E3H ,1
    , ,
    , 'ASEQ',
    , 'CALL',
    , 'CASE',
    , 'CODE',
    , 'CSEQ',
    , 'DSEQ',
    , 'ELSE',
    , 'LDAX',
    , 'LHLD',
    , 'LIST',
    , 'PCHL',
    , 'PUSH',
    , 'SHLD',
    , 'SPHL',
    , 'STAX',
    , 'THEN',
    , 'XCHG',
    , 'XTHL')

```

/* Reserved Word Table for RW's with five characters */

DECLARE RSV\$WORD\$LNQ\$5 (*)

DECLARE RSV\$WORD\$LNQ\$5 (*)		BYTE DATA(12,5,			
NULL	, NULL	, NULL	, NULL	, ,	, ,
CONTROL\$SYM	, DEBUG\$SYM	, 0	, 0	, 'DEBUG',	, 'DEBUG',
DEPTH\$SYM	, 0	, 0	, 0	, 'DEPTH',	, 'DEPTH',
CONTROL\$SYM	, EJECT\$SYM	, 0	, 0	, 'EJECT',	, 'EJECT',
ENDDO\$SYM	, 0	, 0	, 0	, 'ENDDO',	, 'ENDDO',
ENDIF\$SYM	, 0	, 0	, 0	, 'ENDIF',	, 'ENDIF',
EXTRN\$SYM	, EXTRN\$SYM	, 0	, 0	, 'EXTRN',	, 'EXTRN',
LOCAL\$SYM	, 0	, 0	, 0	, 'LOCAL',	, 'LOCAL',
MACRO\$SYM	, 0	, 0	, 0	, 'MACRO',	, 'MACRO',
CONTROL\$SYM	, PRINT\$SYM	, 0	, 0	, 'PRINT',	, 'PRINT',
CONTROL\$SYM	, TITLE\$SYM	, 0	, 0	, 'TITLE',	, 'TITLE',
STATEMENT\$SYM	, WHILE\$SYM	, 0	, 0	, 'WHILE',	, 'WHILE',

*EJECT

/* Reserved Word Table for RW's with six to eight characters */

```

3 1 DECLARE RSV$WORD$LN$NO$6$TO$8 (*)
    NULL , NULL , NULL , NULL , NULL , NULL , NULL , NULL , NULL , NULL ,
    STATEMENT$SYM , CONDIF$SYM , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 ,
    ELSECOND$SYM , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 ,
    ENDCASE$SYM , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 ,
    ENDCOND$SYM , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 ,
    ENDFOR$SYM , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 ,
    ENDMACRO$SYM , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 ,
    ENDMHILE$SYM , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 ,
    CONTROL$SYM , EXPAND$MACRO$SYM , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 ,
    CONTROL$SYM , INCLUDE$SYM , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 ,
    CONTROL$SYM , NOLIST$SYM , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 ,
    CONTROL$SYM , NOOBJECT$SYM , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 ,
    CONTROL$SYM , NOPRINT$SYM , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 ,
    PUBLIC$SYM , PUBLIC$SYM , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 ,
    , 'CONDIF' ,
    , 'ELSECOND' ,
    , 'ENDCASE' ,
    , 'ENDCOND' ,
    , 'ENDFOR' ,
    , 'ENDMACRO' ,
    , 'ENDMHILE' ,
    , 'EXPMACRO' ,
    , 'INCLUDE' ,
    , 'NOLIST' ,
    , 'NOBJECT' ,
    , 'NOPRINT' ,
    , 'PUBLIC' ,
    ,

```

179 1 END RSV\$WORD\$TABLE;

MODULE INFORMATION:

```

CODE AREA SIZE      = 050EH      1294D
VARIABLE AREA SIZE = 0000H      0D
MAXIMUM STACK SIZE = 0000H      0D
479 LINES READ
0 PROGRAM ERROR(S)

```

END OF PL/M-80 COMPILATION

SIS-II PL/M-80 V3.1 COMPILATION OF MODULE SYMTABLEMANAGER
D OBJECT MODULE REQUESTED
COMPILER INVOKED BY: PLM80 SYMBOL.PLM DATE(10-MAR-82) NOOBJECT PRINT(.F1:SYMBOL.LST) PAGELENGTH(50)

```
1      $TITLE('SYMBOL TABLE MANAGEMENT ')  
      SYM$TABLE$MANAGER:  
      DO,  
  
2  1      DECLARE REV810720          BYTE AT(0);
```

```
/* DESCRIPTION:
```

This module contains the routines responsible for managing the symbol table.

SYM\$LOOKUP - looks symbol up in the symbol table.
SYM\$ENTRY - enters symbols into the symbol table.
SYB\$UPDATE - updates existing entries

```
FILES INCLUDED - SYSTEM.ELD  
                - INC.ELD  
                - SMATCH.EPD
```

```
*/
```

```

$EJECT
$NOLIST

174 1 COPY$BUFFS:
175 2 PROCEDURE (DEST$PTR, SRC$PTR, COUNT) EXTERNAL;
176 2 DECLARE DEST$PTR POINTER;
177 2 DECLARE SRC$PTR POINTER;
178 2 DECLARE COUNT WORD;
178 2 END COPY$BUFFS;

179 1 DECLARE CHAR BYTE;
180 1 DECLARE PREV$PTR WORD;
181 1 DECLARE CURR$PTR WORD;
182 1 DECLARE NEW$PTR WORD;

183 1 DECLARE CHILD BYTE;
184 1 DECLARE LEFT LITERALLY'1';
185 1 DECLARE RIGHT LITERALLY'2';

186 1 DECLARE SYM$AREA (2048) BYTE;

187 1 DECLARE SYM$TBL$PTR POINTER;

188 1 DECLARE (SYM$TABLE BASED SYM$TBL$PTR) (1) STRUCTURE(
    TYPE BYTE,
    STATUS BYTE,
    VALUE WORD,
    MACRO$LEV BYTE,
    SYM (SYM$LENGTH) BYTE,
    LCHILD WORD,
    RCHILD WORD);

```

```

189  1  $EJECT
      SYM$INIT:
      PROCEDURE
      PUBLIC;

190  2  SYM$TBL$PTR = SYM$AREA;

191  2  NEW$PTR = 1;

192  2  END SYM$INIT;
    
```

```

$EJECT

193 1  SYM$ENTRY:
    PROCEDURE (IDENT$PTR)          BYTE PUBLIC;

194 2  DECLARE IDENT$PTR          POINTER;

195 2  DECLARE (IDENT BASED IDENT$PTR) STRUCTURE(
    TYPE                          BYTE,
    STATUS                        BYTE,
    VALUE                         WORD,
    MACRO$LEV                    BYTE,
    SYM (SYM$LENGTH)             BYTE);

196 2  IF NEW$PTR > NO$OF$SYM
    THEN RETURN SYM$OVERFLOW;

198 2  CALL COPY$BUFFS (.SYM$TABLE(NEW$PTR).SYM, IDENT.SYM, SYM$LENGTH);
199 2  SYM$TABLE(NEW$PTR).TYPE = IDENT.TYPE;
200 2  SYM$TABLE(NEW$PTR).STATUS = IDENT.STATUS;
201 2  SYM$TABLE(NEW$PTR).VALUE = IDENT.VALUE;
202 2  SYM$TABLE(NEW$PTR).MACRO$LEV = 0;
203 2  SYM$TABLE(NEW$PTR).LCHILD = 0;
204 2  SYM$TABLE(NEW$PTR).RCHILD = 0;
205 2  IF PREV$PTR <> 0
    THEN IF CHILD = LEFT
        THEN SYM$TABLE(PREV$PTR).LCHILD = NEW$PTR;
        ELSE SYM$TABLE(PREV$PTR).RCHILD = NEW$PTR;
    NEW$PTR = NEW$PTR + 1;

210 2  RETURN SYM$ENTERED;

211 2  END SYM$ENTRY;

```

[illegible]

241 2 RETURN SYM\$UNDEF;

242 2 END SYM\$LOOKUP;

```

$EJECT
243 1      SYM$UPDATE;
      PROCEDURE (IDENT$PTR)      BYTE PUBLIC;

244 2      DECLARE IDENT$PTR      POINTER;

245 2      DECLARE (IDENT BASED IDENT$PTR)
      TYPE                        STRUCTURE (
      STATUS                      BYTE,
      VALUE                       BYTE,
      MACRO$LEV                   WORD,
      SYM (SYM$LENGTH)            BYTE,
      )
246 2      SYM$TABLE(CURR$PTR).TYPE = IDENT.TYPE;
247 2      SYM$TABLE(CURR$PTR).STATUS = IDENT.STATUS;
248 2      SYM$TABLE(CURR$PTR).VALUE = IDENT.VALUE;
249 2      SYM$TABLE(CURR$PTR).MACRO$LEV = IDENT.MACRO$LEV;
250 2      RETURN SYM$UPDATED;

251 2      END SYM$UPDATE;

252 1      END SYM$TABLE$MANAGER;

```

MODULE INFORMATION:

```

CODE AREA SIZE      = 02C5H      709D
VARIABLE AREA SIZE = 0B11H      2065D
MAXIMUM STACK SIZE = 0004H       4D
435 LINES READ
0 PROGRAM ERROR(S)

```

END OF PL/M-80 COMPILATION

S-II PL/M-80 V3.1 COMPILATION OF MODULE EXPRESEVAL

JECT MODULE PLACED IN : F4:EXPRES.OBJ

IPILER INVOKED BY: PLM80 EXPRES.PL M DATE(27-MAR-82) OBJECT(:F4:EXPRES.OBJ) PRINT(:F1:EXPRES.LST) PAGELENGTH(50)

*TITLE(' EXPRESSION EVALUATION FOR STRUCTURED MACRO ASSEMBLER')

EXPRES\$EVAL:

DO;

2 1 DECLARE REV810730

BYTE AT(0);

/* WRITTEN BY JOSEPH R. GARAPPOLO

AUGUST 27, 1980

This routine calculates the value of an arithmetic expression.

operator precedence - highest (,)

AND , OR , XOR

* , / , MOD

lowest + , - , unary - , NOT

FILES INCLUDED - INC .ELD
SYSTEM.ELD
GETSYM.EPD
SYMBOL.EPD
ASCII .EPD

*/

\$NOLIST

\$EJECT

23	1	DECLARE EXPRES ERROR SEG\$TYPE EXTRN\$NO	STRUCTURE(BYTE, BYTE, BYTE) PUBLIC;
24	1	DECLARE STRING\$LNG	BYTE;

```

$EJECT

15 1      ADDITIVE:
      PROCEDURE (VALUE$PTR)          REENTRANT;

      /* This routine cares for the + and - operators and also the special
      case unary minus and NOT.
      This is a reentrant routine.
      */

16 2      DECLARE VALUE$PTR          POINTER;
17 2      DECLARE (RET$VALUE BASED VALUE$PTR) STRUCTURE(
      VAL
      SEQ
      WORD,
      BYTE);

228 2      DECLARE VALUE
      VAL
      SEQ
      STRUCTURE(
      WORD,
      BYTE);

229 2      DECLARE VALUE1
      VAL
      SEQ
      STRUCTURE(
      WORD,
      BYTE);

230 2      DECLARE PREV$OP          BYTE;

231 2      /* care for the special case unary + and -, and NOT */
      IF (SYM$TYPE = PLUS$SYM) OR (SYM$TYPE = MINUS$SYM) OR (SYM$TYPE = NOT$SYM)
      THEN DO;
          /* save operator */
          PREV$OP = SYM$TYPE;
          CALL GETSYM;
          /* call next level of precedence */
          CALL MULTIPLICATIVE(.VALUE);
          /* return value must be absolute */
          IF (VALUE.SEG <> ABSOLUTE$SEQ)
          THEN EXPRES.ERROR = 'E';
          ELSE IF (PREV$OP = MINUS$SYM)
          THEN VALUE.VAL = 0 - VALUE.VAL;
          ELSE
              IF (PREV$OP = NOT$SYM)
              THEN VALUE.VAL = OFFFHH - VALUE.VAL;
      END;
      /* not a unary operator go to next level of precedence */
      ELSE CALL MULTIPLICATIVE(.VALUE);
      /* if error return */
      IF (EXPRES.ERROR <> NULL)

```

```

216 3      THEN DO;
217 3          RET$VALUE.VAL = VALUE.VAL;
218 3          RET$VALUE.SEG = VALUE.SEG;
219 3          RETURN;
220 3      END;
221 3
222 3      /* perform addition or subtraction */
223 3      DO WHILE ((SYM$TYPE = PLUS$SYM) OR (SYM$TYPE = MINUS$SYM));
224 3          PREV$OP = SYM$TYPE;
225 3          CALL GETSYM;
226 3          CALL MULTIPLICATIVE(.VALUE1);
227 3
228 3          /* return if error */
229 3          IF (EXPRES.ERROR <> NULL)
230 3              THEN DO;
231 3                  RET$VALUE.VAL = VALUE.VAL;
232 3                  RET$VALUE.SEG = VALUE.SEG;
233 3                  RETURN;
234 3              END;
235 3
236 3          IF (PREV$OP = PLUS$SYM)
237 3              THEN DO;
238 3                  /* check data types */
239 3                  IF (VALUE.SEG = ABSOLUTE$SEG) OR
240 3                      (VALUE1.SEG = ABSOLUTE$SEG)
241 3                      THEN DO;
242 3                          VALUE.VAL = VALUE.VAL + VALUE1.VAL;
243 3                          IF (VALUE.SEG = ABSOLUTE$SEG)
244 3                              THEN VALUE.SEG = VALUE1.SEG;
245 3                      END;
246 3                  /* illegal data type */
247 3                  ELSE EXPRES.ERROR = 'D';
248 3              END;
249 3          ELSE DO;
250 3              /* check data types */
251 3              IF (VALUE1.SEG = ABSOLUTE$SEG) OR (VALUE.SEG = ABSOLUTE$SEG)
252 3              OR ((VALUE.SEG = VALUE1.SEG) AND (VALUE.SEG <> EXTERNAL$SEG))
253 3              THEN DO;
254 3                  VALUE.VAL = VALUE.VAL - VALUE1.VAL;
255 3                  IF (VALUE.SEG <> VALUE1.SEG)
256 3                      THEN DO;
257 3                          IF (VALUE.SEG = ABSOLUTE$SEG)
258 3                              THEN VALUE.SEG = VALUE1.SEG;
259 3                      END;
260 3                  /* illegal data type */
261 3                  ELSE EXPRES.ERROR = 'D';
262 3              END;
263 3          ELSE DO;
264 3              /* check data types */
265 3              IF (VALUE1.SEG = ABSOLUTE$SEG) OR (VALUE.SEG = ABSOLUTE$SEG)
266 3              OR ((VALUE.SEG = VALUE1.SEG) AND (VALUE.SEG <> EXTERNAL$SEG))
267 3              THEN DO;
268 3                  VALUE.VAL = VALUE.VAL * VALUE1.VAL;
269 3                  IF (VALUE.SEG <> VALUE1.SEG)
270 3                      THEN DO;
271 3                          IF (VALUE.SEG = ABSOLUTE$SEG)
272 3                              THEN VALUE.SEG = VALUE1.SEG;
273 3                      END;
274 3                  ELSE VALUE.SEG = ABSOLUTE$SEG;
275 3              END;
276 3          /* illegal data type */
277 3          END;
278 3          /* illegal data type */
279 3          END;
280 3          /* illegal data type */

```

```

31 4          ELSE EXPRES.ERROR = 'D';
32 4          END;
33 3          END;

34 2          /* set up return value and segment(data type) */
35 2          RET$VALUE.VAL = VALUE.VAL;
36 2          RET$VALUE.SEG = VALUE.SEG;

          END ADDITIVE;

```

```

$EJECT
17 1      MULTIPLICITIVE:
      PROCEDURE (VALUE$PTR)                      REENTRANT;

      /* This routine cares for the * and / MOD operators.
      */

18 2      DECLARE VALUE$PTR                      POINTER;
19 2      DECLARE (RET$VALUE BASED VALUE$PTR) STRUCTURE(
      VAL                                      WORD,
      SEG                                      BYTE);

290 2      DECLARE VALUE                      STRUCTURE(
      VAL                                      WORD,
      SEG                                      BYTE);

291 2      DECLARE VALUE1$PTR                      POINTER;
292 2      DECLARE (VALUE1 BASED VALUE1$PTR) STRUCTURE(
      VAL                                      WORD,
      SEG                                      BYTE);

293 2      DECLARE PREV$OP                      BYTE;

294 2      /* go to next level of precedence */
      CALL LOGICAL(. VALUE);

295 2      /* return if error */
      IF (EXPRES.ERROR <> NULL)
      THEN DO,

297 3          RET$VALUE.VAL = VALUE.VAL;
298 3          RET$VALUE.SEG = VALUE.SEG;
299 3          RETURN;
300 3      END;

301 2      /* do multiply , divides or MOD operations */
      DO WHILE (SYM$TYPE = MULT$SYM) OR (SYM$TYPE = DIVI$SYM) OR
      (SYM$TYPE = MOD$SYM);
302 3          PREV$OP = SYM$TYPE;
303 3          CALL GETSYM;
304 3          CALL LOGICAL(. VALUE1);

305 3          IF (EXPRES.ERROR <> NULL)
      THEN DO;
307 4              RET$VALUE.VAL = VALUE.VAL;
308 4              RET$VALUE.SEG = VALUE.SEG;
309 4              RETURN;

```



```

300 4      END;

311 3      /* check data types */
      IF ((VALUE.SEG = ABSOLUTE$SEG) OR (VALUE1.SEG = ABSOLUTE$SEG))
      AND ((VALUE.SEG <> EXTERNAL$SEG) AND (VALUE1.SEG <> EXTERNAL$SEG))
      THEN DO;

313 4          IF (VALUE.SEG = ABSOLUTE$SEG)
      THEN VALUE.SEG = VALUE1.SEG;

315 4          IF (PREV$OP = MULT$SYM)
      THEN VALUE.VAL = VALUE.VAL * VALUE1.VAL;
      ELSE

317 4          IF (PREV$OP = DIVI$SYM)
      THEN VALUE.VAL = VALUE.VAL / VALUE1.VAL;
      ELSE VALUE.VAL = VALUE.VAL MOD VALUE1.VAL;

319 4      END;
320 4      /* illegal data type */
      ELSE EXPRES.ERROR = 'D';

321 3      END;
322 3

323 2      /* set up return value and segment */
324 2      RET$VALUE.VAL = VALUE.VAL;
      RET$VALUE.SEG = VALUE.SEG;

325 2      END MULTIPLICATIVE;

```

```

$EJECT

16 1 LOGICAL:
PROCEDURE (VALUE$PTR) REENTRANT;

/* cares for the AND, OR and XOR logical operators */

17 2 DECLARE VALUE$PTR POINTER;
18 2 DECLARE (RET$VALUE BASED VALUE$PTR) STRUCTURE(
VAL WORD,
SEG BYTE);

19 2 DECLARE VALUE STRUCTURE(
VAL WORD,
SEG BYTE);

330 2 DECLARE VALUE1$PTR POINTER;
331 2 DECLARE (VALUE1 BASED VALUE1$PTR) STRUCTURE(
VAL WORD,
SEG BYTE);

332 2 DECLARE PREV$OP BYTE;

333 2 /* go to highest level of precedence */
CALL FACTOR(.VALUE);

334 2 /* return if error */
IF (EXPRES.ERROR <> NULL)
THEN DO,
RET$VALUE.VAL = VALUE.VAL;
RET$VALUE.SEG = VALUE.SEG;
RETURN;
END;

336 3 RET$VALUE.VAL = VALUE.VAL;
337 3 RET$VALUE.SEG = VALUE.SEG;
338 3 RETURN;
339 3 END;

340 2 /* perform logical operation */
DO WHILE ((SYM$TYPE = AND$SYM) OR (SYM$TYPE = OR$SYM)
OR (SYM$TYPE = XOR$SYM));
PREV$OP = SYM$TYPE;
CALL GETSYM;
/* get nonterminal value */
CALL FACTOR(.VALUE1);

341 3 /* return if error */
342 3 IF (EXPRES.ERROR <> NULL)
343 3 THEN DO;
RET$VALUE.VAL = VALUE.VAL;
RET$VALUE.SEG = VALUE.SEG;
344 3
346 4 RET$VALUE.VAL = VALUE.VAL;
347 4 RET$VALUE.SEG = VALUE.SEG;

```

```
18 4      RETURN;
19 4      END;

10 3      /* check data types */
      IF ((VALUE.SEG = ABSOLUTE$SEG) OR (VALUE1.SEG = ABSOLUTE$SEG))
      AND ((VALUE.SEG <> EXTERNAL$SEG) AND (VALUE1.SEG <> EXTERNAL$SEG))
      THEN DO;
12 4          IF (VALUE.SEG <> ABSOLUTE$SEG)
      14 4              THEN VALUE.SEG = VALUE1.SEG;
              IF (PREV$OP = AND$SYM)
              THEN VALUE.VAL = VALUE.VAL AND VALUE1.VAL;
              ELSE
16 4                  IF (PREV$OP = OR$SYM)
      358 4                      THEN VALUE.VAL = VALUE.VAL OR VALUE1.VAL;
      359 4                      ELSE VALUE.VAL = VALUE.VAL XOR VALUE1.VAL;
      360 3      END;
      361 3      ELSE EXPRES.ERROR = 'D';
      END;

362 2      /* set up return value and segment */
363 2      RET$VALUE.VAL = VALUE.VAL;
      RET$VALUE.SEG = VALUE.SEG;

364 2      END LOGICAL;
```

```

$EJECT

365 1 FACTOR:
PROCEDURE (VALUE$PTR)                      REENTRANT;

/* this routine returns the value of the nonterminal symbol */

366 2 DECLARE VALUE$PTR                      POINTER;
367 2 DECLARE (RET$VALUE BASED VALUE$PTR) STRUCTURE(
         VAL                      WORD,
         SEG                      BYTE);

368 2 DECLARE TEMP$VAL$PTR                      POINTER;
369 2 DECLARE (TEMP$VAL BASED TEMP$VAL$PTR) STRUCTURE(
         VAL                      WORD,
         SEG                      BYTE);

370 2 /* identifier type */
IF (SYM$TYPE = IDENT$SYM)
THEN DO;

372 3 /* get data from symbol table */
IF (SYM$LOOKUP (SYM$PTR) = SYM$UNDEF) AND
((IDENT.STATUS AND PASS2$DEF) <> PASS2$DEF) AND
((IDENT.STATUS AND SEQ$DEF) <> EXTERNAL$SEQ)
/* symbol undefined */
THEN EXPRES.ERROR = 'U';
ELSE DO;

374 3 RET$VALUE.SEG = IDENT.STATUS AND SEQ$DEF;
375 4 IF (RET$VALUE.SEG = EXTERNAL$SEQ)
376 4 THEN DO;

         /* external identifier set
         external number in EXPRES and
         set value to 0 */
         EXPRES.EXTRNSND = IDENT.VALUE;
         RET$VALUE.VAL = 0;
         END;
         ELSE RET$VALUE.VAL = IDENT.VALUE;
         CALL GET$SYM;
         END;
         END;
         ELSE

378 5
379 5
380 5
381 4
382 4
383 4
384 3 ELSE

         /* number */
385 2 IF (SYM$TYPE = NUMBER$SYM)
         THEN DO;

387 3 RET$VALUE.VAL = NUMERIC.VALUE;
388 3 RET$VALUE.SEG = ABSOLUTE$SEG;

```

```

39 3          CALL GETSYM;
70 3          END;
          ELSE

/* data can be the opcode of an instruction */
71 2          IF (SYM$TYPE = INSTRUCTION$SYM)
          THEN DO;
73 3              RET$VALUE.VAL = INSTRUCTION.CODE;
74 3              RET$VALUE.SEG = ABSOLUTE$SEG;
75 3              CALL GET$SYM;
76 3          END;
          ELSE

/* data can be binary value of ascii string of length 1(byte) or 2(word) */
377 2          IF (SYM$TYPE = QUOTE$SYM)
          THEN DO;
379 3              EXPRES.ERROR = G$ASCII(VALUE$PTR, STRING$LNQ, STRING$LNQ);
400 3              IF (EXPRES.ERROR)
                  THEN EXPRES.ERROR = ASCII$STATUS.ERROR;
                  ELSE EXPRES.ERROR = 'E';
402 3              RET$VALUE.SEG = ABSOLUTE$SEG;
403 3              CALL GETSYM;
404 3          END;
          ELSE

/* current location count */
406 2          IF (SYM$TYPE = LOCATION$CNT$SYM)
          THEN DO;
408 3              RET$VALUE.VAL = CURRENT$LOC.VALUE;
409 3              RET$VALUE.SEG = CURRENT$LOC.SEGMENT;
410 3              CALL GET$SYM;
411 3          END;
          ELSE

/* if paren. the recursive call back to ADDITIVE */
412 2          IF (SYM$TYPE = LPAREN$SYM)
          THEN DO;
414 3              CALL GETSYM;
415 3              CALL ADDITIVE(VALUE$PTR);
416 3              IF (SYM$TYPE = RPAREN$SYM)
                  THEN DO;
418 4                  CALL GETSYM;
419 4                  END;
420 3              ELSE EXPRES.ERROR = 'P';
421 3          END;
          ELSE EXPRES.ERROR = 'E';
422 2

```

13 2

END FACTOR;

\$EJECT

```

24 1  EXPRESSION:
    PROCEDURE (RET$VALUE$PTR, SEGMENT, DATA$SIZE) PUBLIC;

/* PARAMETERS: ret$value$ptr - pointer to location where value is to be
    placed.
    segment - data type requested by calling routine.
    data$size - requesting routine requested data size.
                1 - byte data
                2 - word data

This is the routine which invokes the analyzing of the expression.
After the expression is analyzed it check for the requested data size
and data type (segment).

```

*/

```

425 2  DECLARE RET$VALUE$PTR      POINTER;
426 2  DECLARE SEGMENT          BYTE;
427 2  DECLARE DATA$SIZE      BYTE;

428 2  DECLARE (RET$VALUE BASED RET$VALUE$PTR) WORD;

429 2  DECLARE VALUE
      VAL
      SEQ
      STRUCTURE(
      WORD,
      BYTE);

EXPRES.ERROR = NULL;
IF (DATA$SIZE = BYTE$LNQ)
/* used by ascii string value */
THEN STRING$LNQ = 1;
ELSE STRING$LNQ = 2;
/* invoke expression analysis */
CALL ADDITIVE(.VALUE);
/* set up return expression segment type */
EXPRES.SEG$TYPE = VALUE.SEG;

IF (EXPRES.ERROR = NULL)
/* check for user requested segment(data type) if one is
requested */
THEN IF (SEGMENT <> NULL) AND (EXPRES.SEG$TYPE <> SEGMENT)
THEN EXPRES.ERROR = 'D';
ELSE
/* check for used requested data size */
IF (DATA$SIZE = BYTE$LNQ) AND (VALUE.VAL > 255)
THEN EXPRES.ERROR = 'O';

```

```
/* set return value */  
RET$VALUE = VALUE.VAL;
```

```
2 2      END EXPRESSION;
```

```
3 1      END EXPRES$EVAL;
```

RULE INFORMATION:

```
CODE AREA SIZE      = 073FH 1855D  
VARIABLE AREA SIZE = 000BH 11D  
MAXIMUM STACK SIZE = 0013H 19D  
841 LINES READ  
0 PROGRAM ERROR(S)
```

```
END OF PL/M-80 COMPILATION
```


SIS-II PL/M-80 V3.1 COMPILATION OF MODULE CNTL
O OBJECT MODULE REQUESTED
COMPILER INVOKED BY: PLM80 CNTL.PLM DATE(10-MAR-82) NOOBJECT PRINT(:F1:CNTL.LST) PAGELNGTH(50)

```
1          $TITLE('CONTROL FLAGS')
          CNTL:
          DO;

2  1      DECLARE REV820125                BYTE AT(0);

          /* WRITTEN BY JOSEPH R. GARAPPOLO    26 APRIL 1981
```

DESCRIPTION:

This routine checks what pass SMABO is currently in and tests if any action need be taken in this pass for a particular CONTROL. If a control does not exist or no action is to be taken for a particular control in this pass an error is returned

```
INCLUDE FILES - INC .ELD
                SYSTEM.ELD
                ASCII .EPD
                GETSYM. EPD
                FILEIO. EPD
```

```
*/
$NOLIST
```

```

$EJECT
251 1 CLEAR$BUFF;
252 2 PROCEDURE (BUFF$PTR, LENGTH)
253 2 DECLARE BUFF$PTR
254 2 DECLARE LENGTH
254 2 END CLEAR$BUFF;
255 1 OPEN$ERROR;
256 2 PROCEDURE (STATUS)
257 2 DECLARE STATUS
257 2 END OPEN$ERROR;
258 1 DECLARE INPUT$FILE (5)
      AFT$SRC
      BUFF$CNT
      BUFFER (256)
259 1 DECLARE OPEN$FILE (14)
260 1 DECLARE PAGE$FLAG
261 1 DECLARE PRINT$FLAG
262 1 DECLARE DEFAULT$PRINT$FLAG
263 1 DECLARE LIST$FLAG
264 1 DECLARE LIST$FLAG$2
265 1 DECLARE OBJECT$FLAG
266 1 DECLARE MACRO$FLAG
267 1 DECLARE STRUCTURED$FLAG
268 1 DECLARE TITLE$FLAG
269 1 DECLARE TITLE$FLAG$2
270 1 DECLARE DEBUG$FLAG
271 1 DECLARE STATEMENT$CODE$FLAG
272 1 DECLARE LEVEL
273 1 DECLARE TITLE
      BUFF (TITLE$LENGTH)
      LNG
274 1 DECLARE PRINT$FILE (14)
275 1 DECLARE STATUS

```

EXTERNAL;
POINTER;
BYTE;

EXTERNAL,
WORD;

STRUCTURE(
WORD,
WORD,
BYTE) EXTERNAL;

BYTE EXTERNAL;

BYTE PUBLIC;
BYTE PUBLIC;
BYTE PUBLIC;
BYTE PUBLIC;
BYTE PUBLIC;
BYTE PUBLIC;
BYTE PUBLIC;
BYTE PUBLIC;
BYTE PUBLIC;
BYTE PUBLIC;
BYTE PUBLIC;
BYTE PUBLIC;
BYTE PUBLIC;

STRUCTURE(
BYTE,
BYTE) PUBLIC;

BYTE PUBLIC;

BYTE;

\$EJECT

/* This routine initialized the control flags */

```

276 1  CNTL$INIT:
      PROCEDURE

277 2  TITLE.LNG = 0;
278 2  PAGE$FLAG = OFF;
279 2  PRINT$FLAG = ON;
280 2  DEFAULT$PRINT$FLAG = ON;
281 2  LIST$FLAG = ON;
282 2  LIST$FLAG$2 = ON;
283 2  OBJECT$FLAG = ON;
284 2  MACRO$FLAG = OFF;
285 2  STRUCTURED$FLAG = ON;
286 2  TITLE$FLAG = OFF;
287 2  TITLE$FLAG$2 = OFF;
288 2  DEBUG$FLAG = OFF;
289 2  STATEMENT$CODE$FLAG = OFF;
290 2  LEVEL = 0;
291 2  END CNTL$INIT;

      PUBLIC;

```

```

$EJECT
292 1 CNTL$PROC:
    PROCEDURE (PASS)
        BYTE PUBLIC;
293 2 DECLARE PASS
294 2 DECLARE ERROR$TYPE
        BYTE;
        BYTE;
295 2 COMMON$CNTL$1:
    PROCEDURE
        BYTE;

/* no listing until LIST control is encountered ? */
296 3 IF (CONTROL.INST = NOLIST$SYM)
    THEN LIST$FLAG = OFF;
    ELSE
/* start listing again ? */
298 3 IF (CONTROL.INST = LIST$SYM)
    THEN DO;
        LIST$FLAG$2 = ON;
        LIST$FLAG = ON;
        END;
/* no match return control undefined */
300 4 ELSE RETURN 'U';
301 4
302 4
303 3 RETURN NULL;
304 3
305 3 END COMMON$CNTL$1;
306 2 COMMON$CNTL$2:
    PROCEDURE
        BYTE;
/* do not produce object code ? */
307 3 IF (CONTROL.INST = NOOBJECT$SYM)
    THEN OBJECT$FLAG = OFF;
    ELSE
/* include code generated by structured statements in
    listing ? */
309 3 IF (CONTROL.INST = STATEMENT$CODE$SYM)
    THEN STATEMENT$CODE$FLAG = ON;
    ELSE
/* include code generated by Macro expansion in listing ? */
311 3 IF (CONTROL.INST = EXPAND$MACRO$SYM)
    THEN MACRO$FLAG = ON;
    ELSE
/* include symbol names in object code file ? */
313 3 IF (CONTROL.INST = DEBUG$SYM)
    THEN DEBUG$FLAG = ON;
    ELSE

```

```

L/M-80 COMPILER      CONTROL FLAGS

315  3      /* do not produce a listing ? */
          IF (CONTROL.INST = NOPRINT$SYM)
          THEN PRINT$FLAG = OFF;
          ELSE

317  3      /* change the listing file to PRINT(new file name) ? */
          IF (CONTROL.INST = PRINT$SYM)
          THEN DO;

319  4          CALL GET$SYM;
320  4          IF (SYM$TYPE = LPAREN$SYM)
          THEN DO;

322  5              /* indicate not to use default listing
323  5              file and get new file name */
324  5              DEFALT$PRINT$FLAG = OFF;
325  5              CALL CLEAR$BUFF(.PRINT$FILE, 14);
326  5              RETURN G$STRING(.PRINT$FILE, 14, '');
327  4          END;
          ELSE RETURN 'P';
          END;

          RETURN NULL,

329  3      END COMMON$CNTL$2;

330  2      ERROR$TYPE = NULL;

331  2      /* PASS of NULL is used when parsing the runstring */
          IF (PASS = NULL)
          THEN DO;

333  3          ERROR$TYPE = COMMON$CNTL$1;
334  3          IF (ERROR$TYPE = 'U')
          THEN ERROR$TYPE = COMMON$CNTL$2;

336  3          END;
          ELSE

337  2      /* first pass over the source code */
          IF (PASS = 1)
          THEN DO;

339  3          IF (CONTROL.INST = INCLUDE$SYM)
          THEN DO;

341  4              /* get include file name and open it and
342  4              increment the file level */
343  4              CALL GET$SYM;
344  4              CALL CLEAR$BUFF(.OPEN$FILE, 14);
345  4              STATUS = G$STRING(.OPEN$FILE, 14, '');
              LEVEL = LEVEL + 1,
              IF (LEVEL = 5)
              THEN CALL OPEN$ERROR(3),

```



```

376 3          ELSE ERROR$TYPE = COMMON$CNTL$1;
377 3          END;

          RETURN ERROR$TYPE;

379 2          END CNTL$PROC;

380 1          END CNTL;

```

MODULE INFORMATION:

```

CODE AREA SIZE      = 021EH      542D
VARIABLE AREA SIZE = 0042H      66D
MAXIMUM STACK SIZE = 000BH       8D
690 LINES READ
0 PROGRAM ERROR(S)

```

END OF PL/M-80 COMPILATION

```

;--II PL/M-80 V3.1 COMPILATION OF MODULE PASS1
ECT MODULE PLACED IN : F4: PASS1. OBJ
;ILER INVOKED BY: PLM80 PASS1. PLM DATE(27-MAR-82) OBJECT(: F4: PASS1. OBJ) PRINT(: F1: PASS1. LST) DEBUG PAGELENGTH(50)

```

```

1      $TITLE('STRUCTURED MACRO ASSEMBLER PASS1 ')
      PASS1:
      DO;

```

```

2      1      DECLARE REV820327          BYTE AT(0);

```

```

/*      WRITTEN BY JOSEPH R. GARAPPOLO      10 OCT 1981

```

The first pass of SMABO is responsible for,

1. evaluation conditional assembly
2. define and expand Macro's
3. generate code for structure statements.
4. generate an intermediate file that includes intermediate code generated by PASS1 and the source code from the main source file and INCLUDE file(s) if any are used.

```

INCLUDE FILES - INC .ELD
                SYSTEM.ELD
                GETSYM.EPD
                EXPRES.EPD
                SYMBOL.EPD
                CNTL .EPD
                MACRO .EPD
                CONDIT.EPD
                COPYSR.EPD

```

```

*/
$NOLIST

```


\$EJECT

```

287 1 FILE$INIT$PASS1:
288 2 PROCEDURE
289 2 END FILE$INIT$PASS1;
      EXTERNAL;

290 1 EVALUATE$CONDITION:
291 2 PROCEDURE (ERROR$PTR)
292 2 DECLARE ERROR$PTR
293 2 END EVALUATE$CONDITION;
      BYTE EXTERNAL;
      POINTER;

294 1 COPY$BUFFS:
295 2 PROCEDURE (DEST$PTR, SRC$PTR, LNG)
296 2 DECLARE DEST$PTR
297 2 DECLARE SRC$PTR
298 2 DECLARE LNG
299 2 END COPY$BUFFS;
      EXTERNAL;
      POINTER;
      WORD;

300 1 JUMP$TAG$INIT:
301 2 PROCEDURE
302 2 END JUMP$TAG$INIT;
      EXTERNAL;

303 1 JUMP$TAG:
304 2 PROCEDURE (TAG$PTR)
305 2 DECLARE TAG$PTR
306 2 END JUMP$TAG;
      EXTERNAL;
      POINTER;

307 1 DECLARE NULL$BLOCK
308 1 DECLARE IF$BLOCK
309 1 DECLARE WHILE$BLOCK
310 1 DECLARE FOR$BLOCK
311 1 DECLARE CASE$BLOCK
312 1 DECLARE CONDIF$BLOCK
313 1 DECLARE BLOCK$TYPE
314 1 DECLARE IF$COUNT
315 1 DECLARE WHILE$COUNT
316 1 DECLARE FOR$COUNT
317 1 DECLARE CASE$COUNT
318 1 DECLARE ERROR$TYPE
319 1 DECLARE STATUS
320 1 DECLARE COND$BUFF$PTR
321 1 DECLARE COND$LNG
322 1 DECLARE IDENTIFIER$FLAG
      LITERALLY '0';
      LITERALLY '1';
      LITERALLY '2';
      LITERALLY '3';
      LITERALLY '4';
      LITERALLY '5';

      BYTE;

      BYTE EXTERNAL;
      BYTE EXTERNAL;
      BYTE EXTERNAL;
      BYTE EXTERNAL;

      BYTE;
      BYTE;

      POINTER;
      WORD;
      BYTE;

```

```

307          $EJECT
308
309      1      PASS1$COND:
310      PROCEDURE (COND1, COND2, COND3, BLOCK) REENTRANT;
311
312      /* This reentrant routine loops calling the pass1 parser routine PASS1$PROC,
313         until one of the stop symbols is found. COND1, COND2 and COND3 are
314         defined by the calline routine.
315      */
316
317      DECLARE BLOCK
318      DECLARE PREVIOUS$BLOCK
319      DECLARE COND1
320      DECLARE COND2
321      DECLARE COND3
322
323      PREVIOUS$BLOCK = BLOCK$TYPE;
324      BLOCK$TYPE = BLOCK;
325      STATUS = GET$LINE;
326      CALL GET$SYM;
327      ERROR$TYPE = NULL;
328      DO WHILE (SYM$TYPE <> EOF$SYM) AND (SYM$TYPE <> END$SYM) AND
329              (SYM$TYPE <> COND1) AND (SYM$TYPE <> COND2) AND (SYM$TYPE <> COND3)
330              AND (STATUS = TRUE);
331              ERROR$TYPE = NULL;
332              STATUS = PASS1$PROC;
333              STATUS = GET$LINE;
334              CALL GET$SYM;
335      END;
336      BLOCK$TYPE = PREVIOUS$BLOCK;
337
338      END PASS1$COND;

```

```

$EJECT
} 1      SKIP$TAG:
    PROCEDURE (ERROR$TYPE, STMT$TYPE);

} 2      DECLARE STMT$TYPE      BYTE;
} 2      DECLARE ERROR$TYPE    BYTE;
} 2      DECLARE MACRO$TAG      BYTE;

/* This routine generates the intermediate code header for structure
   statement and fills in the error field that is used by the print
   routine.
   This header is copied into line$buff$2 which is five bytes long
   and attached to the front of line$buffer.
*/

329      IF (ACTIVE$MACRO$FLAG)
331      THEN MACRO$TAG = MACRO$LINE;
        ELSE MACRO$TAG = NULL;

332      IF (STMT$TYPE = NULL) AND NOT(ACTIVE$MACRO$FLAG)
        THEN DO;
334          CALL COPY$SOURCE (.LINE$BUFF, OFFFFH);
335          RETURN;
336          END;

337      LINE$BUFF$2(1) = STMT$TYPE OR MACRO$TAG;
338      LINE$BUFF$2(0) = SKIP$STATEMENT;
339      IF (IDENTIFIER$FLAG = ON) AND (STMT$TYPE <> NULL)
        THEN DO;
        /* there is an location tag on this line that must
           be obtained by passes two and three. */
        LINE$BUFF$2(1) = TAG$LINE OR MACRO$TAG;
        IDENTIFIER$FLAG = OFF;
        END;
        /* this line is to be skipped by passes two and three */
        LINE$BUFF$2(2) = ERROR$TYPE;
341      LINE$BUFF$2(3) = NULL;
342      LINE$BUFF$2(4) = NULL;
343      LINE$BUFF$2(5) = NULL;
        /* copy to intermediate starting at line$buff$2 */
344      CALL COPY$SOURCE (.LINE$BUFF$2, OFFFFH);
345
346      END SKIP$TAG;

```

```

$EJECT
? 1 IF$PROC:
PROCEDURE
    REENTRANT;
/* This routine parses and generated code for the IF-THEN-ELSE
statement.
    IF <condition> THEN <statement> ENDIF
or
    IF <condition> THEN <statement> ELSE <statement> ENDIF
*/
350 2 DECLARE EXIT$JUMP          (16) BYTE;
351 2 DECLARE ELSE$JUMP         (16) BYTE;
352 2 DECLARE LINE$INDENT
/* set up line indentation for generated code */
353 2 LINE$INDENT = $CHAR$CNT + 2;
354 2 CALL GET$SYM;
/* generate code for IF condition */
355 2 ERROR$TYPE = CONDITION(.EXIT$JUMP, .COND$BUFF$PTR, .COND$LNQ, LINE$INDENT);
356 2 CALL GET$SYM;
357 2 IF (SYM$TYPE <> THEN$SYM) AND (ERROR$TYPE = NULL)
    THEN ERROR$TYPE = 'T';
359 2 CALL SKIP$TAG(ERROR$TYPE, STATEMENT$LINE);
/* return if error while generating code */
360 2 IF (ERROR$TYPE <> NULL) AND (ERROR$TYPE <> 'T')
    THEN RETURN;
/* copy the generated conditional code to the intermediate file */
362 2 CALL COPY$SOURCE(COND$BUFF$PTR, COND$LNQ-2);
/* increment the number of IF's currently activated */
363 2 IF$COUNT = IF$COUNT + 1;
/* get the nonterminal statement until ENDIF or ELSE */
364 2 CALL PASS1$COND (ELSE$SYM, ENDIF$SYM, NULL, IF$BLOCK);
365 2 IF (SYM$TYPE = ELSE$SYM) OR (SYM$TYPE = ENDIF$SYM)
    THEN DO;
367 3 IF (SYM$TYPE = ELSE$SYM)
    THEN DO;
/* if ELSE generate code to take care fo ELSE */
369 4 CALL COPY$BUFFS(.ELSE$JUMP, .EXIT$JUMP, 16);
370 4 CALL JUMP$TAG(.EXIT$JUMP);
371 4 CALL COPY$CHAR(.SKIP$STATEMENT, STATEMENT$CODE, 0, 0, 0, ' '), LINE$INDENT+5);
372 4 CALL COPY$SOURCE(.EXIT$JUMP, 14);
373 4 CALL SKIP$TAG(NULL, STATEMENT$LINE);

```

```
1 4      CALL COPY$CHAR(, (SKIP$STATEMENT, STATEMENT$CODE, 0, 0, 0), 5);
2 4      CALL COPY$SOURCE(, ELSE$JUMP(8), 7);
3 4      /* find ENDIF */
4 4      CALL PASS1$COND(ENDIF$SYM, NULL, NULL, IF$BLOCK);
5 4      END;
6 4      /* setup exit jump and clean up for IF routine exiting */
7 4      CALL SKIP$TAG(NULL, STATEMENT$LINE);
8 3      CALL COPY$CHAR(, (SKIP$STATEMENT, STATEMENT$CODE, 0, 0, 0), 5);
9 3      CALL COPY$SOURCE(, EXIT$JUMP(8), 7);
10 3      IF$COUNT = IF$COUNT - 1;
11 3      END;
12 3      END IF$PROC;
13 2
```

```

$EJECT

1 1 WHILE$PROC:
PROCEDURE
    REENTRANT;

/* This routine parses and generates code for the WHILE statement.
   WHILE <condition> DO <statement> ENDWHILE
*/
2 2 DECLARE START$JUMP
2 2 DECLARE EXIT$JUMP (16) BYTE;
2 2 DECLARE LINE$INDENT (16) BYTE;
    BYTE;

/* set up index for generated code - make it look nice */
388 LINE$INDENT = 0;
389 CALL GET$SYM;
390 ERROR$TYPE = CONDITION(.EXIT$JUMP, COND$BUFF$PTR, COND$LN, LINE$INDENT);
391 CALL GET$SYM;
392 IF (SYM$TYPE <> DO$SYM) AND (ERROR$TYPE = NULL)
    THEN ERROR$TYPE = 'D';
394 CALL SKIP$TAG(ERROR$TYPE, STATEMENT$LINE);

/* if error other than 'D' return */
395 IF (ERROR$TYPE <> NULL) AND (ERROR$TYPE <> 'D')
    THEN RETURN;

CALL JUMP$TAG(.START$JUMP);
397 CALL COPY$CHAR(.SKIP$STATEMENT, STATEMENT$CODE, 0, 0, 0, 5);
398 CALL COPY$SOURCE(.START$JUMP(8), 7);
399 /* copy generated conditional code into intermediate file */
400 CALL COPY$SOURCE(COND$BUFF$PTR, COND$LN-2);
401 /* increment number of active WHILE statements */
    WHILE$COUNT = WHILE$COUNT + 1;
402 /* get nonterminal statement of WHILE statement */
403 CALL PASS1$COND(ENDWHILE$SYM, NULL, NULL, WHILE$BLOCK);
    IF (SYM$TYPE = ENDWHILE$SYM)
    THEN DO;
405 /* since it is and ENDWHILE symbol clean up */
    WHILE$COUNT = WHILE$COUNT - 1;
406 /* copy a jump to start into intermediate file */
407 CALL COPY$CHAR(.SKIP$STATEMENT, STATEMENT$CODE, 0, 0, 0, ' '), LINE$INDENT+5);
408 CALL COPY$SOURCE(.START$JUMP, 14);
    CALL SKIP$TAG(NULL, STATEMENT$LINE);
409 /* copy the exit location tag into intermediate file */
    CALL COPY$CHAR(.SKIP$STATEMENT, STATEMENT$CODE, 0, 0, 0, 5);
410 CALL COPY$SOURCE(.EXIT$JUMP(8), 7);

```

```
3
1 2      END WHILE$PROC;
      END;
```

```

$EJECT
3 1  FOR$PROC:
    PROCEDURE
        REENTRANT;
    /* This parsed and generated code for the FOR statement.
    FOR <. [identifier i number]] = <. [identifier i number]]>
        TO <. [identifier i number]]>
        <statement> ENDFOR
    or
    FOR <. [identifier i number]] = <. [identifier i number]]>
        TO <. [identifier i number]]>
        BY <. [identifier i number]]>
        <statement> ENDFOR
    */
414 2  DECLARE INC$JUMP
415 2  DECLARE EXIT$JUMP
416 2  DECLARE LINE$INDENT
        (16) BYTE;
        (16) BYTE;
        BYTE;
    /* set up line indentation for generated code */
417 2  LINE$INDENT = G$CHAR$CNT + 1;
418 2  CALL GET$SYM;
    /* generate code to evaluate FOR statement */
419 2  ERROR$TYPE = FOR$CODE(. INC$JUMP, . EXIT$JUMP, . COND$BUFF$PTR, . COND$NLG, LINE$INDENT);
420 2  IF (SYM$TYPE <> DO$SYM) AND (ERROR$TYPE = NULL)
421 2  THEN ERROR$TYPE = 'D';
422 2  CALL SKIP$TAG(ERROR$TYPE, STATEMENT$LINE);
    /* return if error otherthan 'D' */
423 2  IF (ERROR$TYPE <> NULL) AND (ERROR$TYPE <> 'D')
424 2  THEN RETURN;
    /* copy generated code to intermediate file */
425 2  CALL COPY$SOURCE(COND$BUFF$PTR, COND$NLG-2);
    /* increment active FOR statement count */
426 2  FOR$COUNT = FOR$COUNT + 1;
    /* get nonterminal statement in FOR statement */
427 2  CALL PASS1$COND (ENDFOR$SYM, NULL, NULL, FOR$BLOCK);
428 2  IF (SYM$TYPE = ENDFOR$SYM)
429 2  THEN DO;
        /* we found the ENDFOR symbol. Generate code for jump
        to increment portion of code and exit location tag
        and clean up */
        FOR$COUNT = FOR$COUNT - 1;
        CALL COPY$CHAR(. (SKIP$STATEMENT, STATEMENT$CODE, O.O.O, ' '), LINE$INDENT+5);
430 3
431 3

```



```

2 3      CALL COPY$SOURCE(. INC$JUMP, 14);
3 3      CALL SKIP$TAG(NULL, STATEMENT$LINE);
4 3      CALL COPY$CHAR(. (SKIP$STATEMENT, STATEMENT$CODE, O.O.O.), 5);
5 3      CALL COPY$SOURCE(. EXIT$JUMP(8), 7);
6 3      END,
7 2      END FOR$PROC;

```

*EJECT

```

18 1  JUMP$TABLE$FILL:
19 2  PROCEDURE (TAG$PTR, JUMP$TABLE$PTR, TABLE$CNT$PTR);
20 2  DECLARE TAG$PTR          POINTER;
21 2  DECLARE JUMP$TABLE$PTR    POINTER;
22 2  DECLARE TABLE$CNT$PTR    POINTER;

/* PARAMETERS: tag$ptr - pointer to location tag.
jump$table$ptr - pointer to jump table buffer
table$cnt$ptr - pointer to next empty element
of the jump table.

This routine copies the just the location tags of the jump
table entries into the jump table for the CASE statement
*/
442 2  DECLARE (JUMP$TABLE BASED JUMP$TABLE$PTR) (256) BYTE;
443 2  DECLARE (TABLE$INDEX BASED TABLE$CNT$PTR) BYTE;

/* each tag is six(6) bytes long */
444 2  CALL COPY$BUFFS (.JUMP$TABLE(TABLE$INDEX*6), TAG$PTR , 6);
445 2  TABLE$INDEX = TABLE$INDEX + 1;

446 2  END JUMP$TABLE$FILL;

447 1  JUMP$TABLE$COPY:
448 2  PROCEDURE (JUMP$TABLE$PTR, TABLE$CNT$PTR, TBL$JUMP$PTR, INDENT);
449 2  DECLARE JUMP$TABLE$PTR    POINTER;
450 2  DECLARE TABLE$CNT$PTR    POINTER;
451 2  DECLARE TBL$JUMP$PTR      POINTER;
452 2  DECLARE INDENT            BYTE;

/* This routine expands entries in the jump table buffer into code.
Jump table buffer is expanded to.

JUMP$TABLE$TAG:
DW CASE$1$TAG
DW CASE$2$TAG
etc. until all cases are in table
*/
452 2  DECLARE TAG$COUNT          BYTE;
453 2  DECLARE (JUMP$TABLE BASED JUMP$TABLE$PTR) (186) BYTE;
454 2  DECLARE (TABLE$INDEX BASED TABLE$CNT$PTR) BYTE;

```

```

/* The first entry in the jump table in the location tag for
the jump table. Copy the tag into the intermediate file */
CALL COPY$CHAR (. (BKIP$STATEMENT, STATEMENT$CODE, 0, 0, 0), 5);
CALL COPY$SOURCE (TBLS$JUMP$PTR, 7);
TAG$COUNT = 0;

```

```

/* creat DW statements for all other entries in table */
DO WHILE (TAG$COUNT < TABLE$INDEX);
  CALL COPY$CHAR (. (SKIP$STATEMENT, STATEMENT$CODE, 0, 0, 0), 5);
  CALL COPY$CHAR (. (' '), INDENT);
  CALL COPY$CHAR (. ('DW '), 4);
  CALL COPY$SOURCE (. JUMP$TABLE(TAG$COUNT*6), 6);
  TAG$COUNT = TAG$COUNT + 1;
END;

```

```
END JUMP$TABLE$COPY;
```

```
CASE$PROC:
PROCEDURE
```

```
REENTRANT;
```

```
/* This routine parsed and generated code for the CASE statement.
```

```

CASE <Identifier ! number> DO <statement> ENDDO ... ENDCASE
or
CASE <Identifier ! number> TO <Identifier!number>
DO <statement> ENDDO ... ENDCASE

```

```
*/
```

```

DECLARE TABLE$JUMP          (16) BYTE;
DECLARE DO$BLOCK$JUMP        (16) BYTE;
DECLARE EXIT$JUMP            (16) BYTE;
DECLARE LINE$INDENT          BYTE;
DECLARE DO$COUNT            BYTE;
DECLARE JUMP$TABLE (186)
DECLARE TABLE$INDEX         BYTE;
DECLARE END$INDENT           BYTE;

```

```

/* set up indentation for generated code */
LINE$INDENT = G$CHAR$CNT;
CALL GET$SYM;

```

```

/* generate code to evaluate CASE statement */
ERROR$TYPE = CASE$CODE(. TABLE$JUMP, . EXIT$JUMP, . COND$BUFF$PTR, . COND$LN$G, LINE$INDENT);
IF (SYM$TYPE <> DO$SYM) AND (ERROR$TYPE = NULL)
  THEN ERROR$TYPE = 'D';
CALL SKIP$TAG(ERROR$TYPE, STATEMENT$LINE);

```

```

1      2      /* return if error is otherthan 'D' */
2      IF (ERROR$TYPE <> NULL) AND (ERROR$TYPE <> 'D')
3          THEN RETURN;

4      3      /* set up for DO blocks which are within CASE statements */
5      4      TABLE$INDEX = 0;
6      5      DO$COUNT = 0;
7      6      /* copy generated code for CASE statement into intermediate file */
8      7      CALL COPY$SOURCE(COND$BUFFPTR, COND$LNQ-2);
9      8      /* increment active CASE statement count */
10     9      CASE$COUNT = CASE$COUNT + 1;
11     10     DO WHILE (SYM$TYPE <> ENDCASE$SYM) AND (SYM$TYPE <> EOF$SYM) AND
12     11     (SYM$TYPE <> ENDSYM);
13     12     /* look for DO blocks that indicate cases */
14     13     CALL PASS1$COND (ENDCASE$SYM, DO$SYM, ENDDO$SYM, CASE$BLOCK);
15     14     IF (SYM$TYPE = DO$SYM)
16     15     THEN DO;
17     16     IF (DO$COUNT = ON)
18     17     /* two DO's without a ENDDO */
19     18     THEN ERROR$TYPE = 'I';
20     19     ELSE DO; /* creat location tag for DO block(case) */
21     20     CALL JUMP$TAG (. DO$BLOCK$JUMP);
22     21     CALL COPY$CHAR (. (SKIP$STATEMENT, STATEMENT$CODE,
23     22     0, 0, 0), 5);
24     23     CALL COPY$SOURCE (. DO$BLOCK$JUMP(8), 7);
25     24     /* copy tag name into jump table */
26     25     CALL JUMP$TABLE$FILL (. DO$BLOCK$JUMP(8),
27     26     . JUMP$TABLE, . TABLE$INDEX);
28     27     DO$COUNT = ON;
29     28     ERROR$TYPE = NULL;
30     29     END;
31     30     CALL SKIP$TAG(ERROR$TYPE, STATEMENT$LINE);
32     31     END;
33     32     ELSE
34     33     IF (SYM$TYPE = ENDDO$SYM)
35     34     THEN DO;
36     35     /* ENDDO without DO */
37     36     IF (DO$COUNT = OFF)
38     37     THEN ERROR$TYPE = 'I';
39     38     ELSE DO;
40     39     ENDS$INDENT = G$CHAR$CNT - 1;
41     40     ERROR$TYPE = NULL;
42     41     CALL COPY$CHAR (. (SKIP$STATEMENT, STATEMENT$CODE,
43     42     0, 0, 0), 5);
44     43     CALL COPY$CHAR (. (' '), ENDS$INDENT);
45     44     /* set up jump to exit */
46     45     CALL COPY$SOURCE (. EXIT$JUMP, 14);
47     46     END;

```

```

003 3 5 DO$COUNT = OFF;
004 4 5 END;
005 3 4 CALL SKIP$TAG(ERROR$TYPE, STATEMENT$LINE);
006 4 4 END;
007 3 2 END;
008 3 2 IF (SYM$TYPE = ENDCASE$SYM)
009 3 2 THEN DO;
010 3 3 CALL SKIP$TAG(NULL, STATEMENT$LINE);
011 3 3 /* expand jump table and copy to intermediate file */
012 3 3 CALL JUMP$TABLE$COPY(.JUMP$TABLE, .TABLE$INDEX, .TABLE$JUMP(8), LINE$INDENT);
013 3 3 CALL COPY$CHAR(. (SKIP$STATEMENT, STATEMENT$CODE, 0, 0, 0), 5);
014 3 3 /* copy exit tag to intermediate file */
015 3 3 CALL COPY$SOURCE(.EXIT$JUMP(8), 7);
016 3 3 CASE$COUNT = CASE$COUNT - 1;
017 3 3 END;
018 3 2 END CASE$PROC;

```

```

7 1      $EJECT
8      CONDIF$SCAN:
9      PROCEDURE (STOP$SYM)      REENTRANT,
10      /* this procedure scan the input code for the proper ENDCOND
11      skipping line between.
12      */
13      DECLARE STOP$SYM      BYTE;
14      STATUS = GET$LINE;
15      CALL GET$SYM;
16      /* make sure not end of source code */
17      DO WHILE (SYM$TYPE <> EOF$SYM) AND (SYM$TYPE <> END$SYM) AND
18      (STATUS);
19      /* clear identifier$flag don't want to used identifier field */
20      IDENTIFIER$FLAG = OFF;
21      /* increment nested level if CONDIF statement */
22      IF (SYM$TYPE = STATEMENT$SYM) AND (STATEMENT.INST = CONDIF$SYM)
23      THEN DO;
24      CALL SKIP$TAG (NULL, STATEMENT$LINE);
25      CALL CONDIF$SCAN (NULL);
26      END;
27      ELSE
28      /* return if ENDCOND is associated as first level or else
29      decrement nested level count */
30      IF (SYM$TYPE = ENDCOND$SYM)
31      THEN DO;
32      RETURN;
33      END;
34      ELSE
35      /* if symbol is stop symbol at current level return else
36      no action */
37      IF (SYM$TYPE = STOP$SYM)
38      THEN DO;
39      RETURN;
40      END;
41      CALL SKIP$TAG (NULL, STATEMENT$LINE);
42      STATUS = GET$LINE;
43      CALL GET$SYM;
44      END;
45      END CONDIF$SCAN;
46      CONDIF$PROC:
47      PROCEDURE
48      /* This routine parses conditional assembly statement
49      CONDIF <condition> THEN <statement> ENDCOND

```



```

3 4      THEN DO;      /* restore previous block */
1 4          CALL SKIP$TAG(O, STATEMENT$LINE);
2 3          END;
3 2          END;
          END CONDIF$PROC;

```



```

587 2          $EJECT
588 2          IDENT$PROC:
589 2          PROCEDURE
590 2          BYTE REENTRANT;
591 2          /* This routine fill the symbol table with SET, EGT and MACOR (to be done)
592 2          identifiers. Location tag identifier are skipped.
593 2          */
594 2          DECLARE STATUS
595 2          DECLARE TEMP$IDENT
596 2          TYPE
597 2          STATUS
598 2          VALUE
599 2          DUMMY
600 2          CHAR (SYMSLENGTH)
601 2          BYTE;
602 2          STRUCTURE(
603 2          BYTE,
604 2          WORD,
605 2          BYTE,
606 2          BYTE);
607 2          DECLARE STATEMENT$LINE$TYPE
608 2          BYTE;
609 2          /* this flag is used to determine if what follows the location tag
610 2          is a structure statement. Initialize off */
611 2          STATEMENT$LINE$TYPE = OFF;
612 2          /* this is a reentrant routine we must save identifier name
613 2          that is in symbol buffer */
614 2          CALL COPY$BUFFERS(.TEMP$IDENT.CHAR, .IDENT.CHAR, SYMSLENGTH);
615 2          TEMP$IDENT.TYPE = NULL;
616 2          /* look the symbol up */
617 2          STATUS = SYM$LOOKUP(.TEMP$IDENT);
618 2          CALL GET$SYM;
619 2          IF (SYM$TYPE = SET$SYM)
620 2          THEN DO;
621 2          /* set symbols are entered into the symbol and can be
622 2          updated during assembly */
623 2          CALL GET$SYM;
624 2          /* get value */
625 2          CALL EXPRESSION(.TEMP$IDENT.VALUE, ABSOLUTE$SEQ, WORD$NLNG);
626 2          TEMP$IDENT.STATUS = ABSOLUTE$SEQ;
627 2          IF (STATUS = SYM$EXIST)
628 2          /* if it is already in symbol table only update it */
629 2          THEN STATUS = SYM$UPDATE(.TEMP$IDENT);
630 2          ELSE STATUS = SYM$ENTRY(.TEMP$IDENT);
631 2          END;
632 2          ELSE
633 2          IF (SYM$TYPE = EQU$SYM)
634 2          THEN IF (STATUS <> SYM$EXIST)
635 2          THEN DO;

```

```

5 3      /* EGU identifiers can not be redefined during
        assembly */
6 3      CALL GET$SYM;
        /* get value */
7 3      CALL EXPRESSION(, TEMP$IDENT, VALUE, ABSOLUTE$SEQ, WORD$LNQ);
        /* do not enter into symbol table if error exist */
        IF (EXPRES.ERROR = NULL)
9 4      THEN DO;
            TEMP$IDENT.STATUS = ABSOLUTE$SEQ;
            /* if it is already in symbol table
              only update it */
            STATUS = SYM$ENTRY(, TEMP$IDENT);
            END;
        ELSE,
612 3
613 2
614 2      IF (SYM$TYPE = COLON$SYM)
        THEN DO;
            /* this flag tells next statement a location tag is on the
              same line. If the next statement is a structured statement
              the intermediate code before the line will reflect that
              the identifier must be parsed by passes two and three */
            IDENTIFIER$FLAG = ON;
            CALL GET$SYM;
            STATEMENT$LINE$TYPE = PASS1$PROC;
            IDENTIFIER$FLAG = OFF;
            END;
        ELSE
621 2      IF (SYM$TYPE = MACRO$SYM)
        THEN DO;
            /* define macro - a macro may be redefined */
            CALL MACRO$DEFINE (, TEMP$IDENT);
            RETURN ON; /* indicate line has be copied to intermediate buffer */
            END;
        ELSE
626 2      IF (TEMP$IDENT.TYPE = MACRO$SYM)
        THEN DO;
            /* this identifier is a macro so expand it */
            CALL MACRO$EXPAND (TEMP$IDENT.VALUE);
            RETURN ON; /* line has been copied to intermediate file */
            END;
        /* flag is on the line has already been copied to the intermediate file */
        IF (STATEMENT$LINE$TYPE = OFF)
        THEN DO;
            CALL SKIP$TAG (NULL, NULL);
            RETURN ON;
            END;
633 3
634 3
635 3

```

6 2 ELSE RETURN OFF;

7 2 END IDENT\$PROC;

```

$EJECT
638 1 PASS1$PROC:
    PROCEDURE
        BYTE REENTRANT;
639 2 IF (SYM$TYPE = STATEMENT$SYM) AND (STATEMENT.INST = IF$SYM)
        THEN CALL IF$PROC;
        ELSE
641 2 IF (SYM$TYPE = ELSE$SYM)
        THEN DO;
643 3 IF (BLOCK$TYPE <> IF$BLOCK)
        THEN CALL SKIP$TAG('I', STATEMENT$LINE);
        ELSE RETURN ON;
645 3
646 3 END;
647 2 ELSE
        IF (SYM$TYPE = ENDIF$SYM)
        THEN DO;
649 3 IF (BLOCK$TYPE <> IF$BLOCK)
        THEN CALL SKIP$TAG('I', STATEMENT$LINE);
        ELSE RETURN ON;
651 3
652 3 END;
653 2 ELSE
        IF (SYM$TYPE = STATEMENT$SYM) AND (STATEMENT.INST = WHILE$SYM)
        THEN CALL WHILE$PROC;
        ELSE
655 2 IF (SYM$TYPE = ENDWHILE$SYM)
        THEN DO;
657 3 IF (BLOCK$TYPE <> WHILE$BLOCK)
        THEN CALL SKIP$TAG('I', STATEMENT$LINE);
        ELSE RETURN ON;
659 3
660 3 END;
661 2 ELSE
        IF (SYM$TYPE = STATEMENT$SYM) AND (STATEMENT.INST = FOR$SYM)
        THEN CALL FOR$PROC;
        ELSE
663 2 IF (SYM$TYPE = ENDFOR$SYM)
        THEN DO;
665 3 IF (BLOCK$TYPE <> FOR$BLOCK)
        THEN CALL SKIP$TAG('I', STATEMENT$LINE);
        ELSE RETURN ON;
667 3
668 3 END;
669 2 ELSE
        IF (SYM$TYPE = STATEMENT$SYM) AND (STATEMENT.INST = CASE$SYM)
        THEN CALL CASE$PROC;
        ELSE
671 2 IF (SYM$TYPE = DO$SYM)
        THEN DO;

```

```

3 3      IF (BLOCK$TYPE <> CASE$BLOCK)
5 3          THEN CALL SKIP$TAG('I', STATEMENT$LINE);
      END;
6 2      ELSE
8 3      IF (SYM$TYPE = ENDDO$SYM)
10 3          THEN DO;
11 2          IF (BLOCK$TYPE <> CASE$BLOCK)
13 3              THEN CALL SKIP$TAG('I', STATEMENT$LINE);
      END;
14 2      ELSE
15 3      IF (SYM$TYPE = ENDCASE$SYM)
17 3          THEN DO;
18 3          IF (BLOCK$TYPE <> CASE$BLOCK)
20 3              THEN CALL SKIP$TAG('I', STATEMENT$LINE);
      END;
21 3      ELSE
22 3      IF (SYM$TYPE = STATEMENT$SYM) AND (STATEMENT.INST = CONDIF$SYM)
24 3          THEN CALL CONDIF$PROC;
      ELSE
25 3      IF (SYM$TYPE = ELSECOND$SYM)
27 3          THEN DO;
28 3          IF (BLOCK$TYPE <> CONDIF$BLOCK)
30 3              THEN CALL SKIP$TAG('I', STATEMENT$LINE);
      ELSE RETURN ON;
      END;
31 3      ELSE
32 3      IF (SYM$TYPE = ENDCOND$SYM)
34 3          THEN DO;
35 3          IF (BLOCK$TYPE <> CONDIF$BLOCK)
37 3              THEN CALL SKIP$TAG('I', STATEMENT$LINE);
      ELSE RETURN ON;
      END;
38 3      ELSE
39 3      IF (SYM$TYPE = IDENT$SYM)
41 3          THEN DO;
42 3          IF (BLOCK$TYPE <> CONDIF$BLOCK)
44 3              THEN CALL SKIP$TAG('I', STATEMENT$LINE);
      ELSE RETURN ON;
      END;
45 3      ELSE
46 3      IF (SYM$TYPE = IDENT$SYM)
48 3          THEN DO;
49 3          RETURN IDENT$PROC;
      END;
50 3      ELSE
51 3      IF (SYM$TYPE = CONTROL$SYM)
53 3          THEN DO;
54 3          STATUS = CNTL$PROC(1);
55 3          IF (STATUS <> 'U')
56 3              /* if not undefined tell other passes to
57 3                  skip this line
58 3              */
59 3          THEN CALL SKIP$TAG(STATUS, STATEMENT$LINE);
      ELSE CALL SKIP$TAG(NULL, NULL);
60 3
61 3
62 3
63 3
64 3
65 3
66 3
67 3
68 3
69 3
70 2
71 3
72 3
73 3
74 2
75 3
76 3
77 3
78 3
79 3
80 3
81 3
82 3
83 3
84 3
85 3
86 3
87 3
88 3
89 3
90 3
91 3
92 3
93 3
94 2
95 3
96 3
97 3
98 3
99 3
100 2
101 3
102 3
103 3
104 2
105 3
106 3
107 3
108 3
109 3
110 3
111 3
112 3
113 3
114 3
115 3
116 3
117 3
118 3
119 3
120 3
121 3
122 3
123 3
124 3
125 3
126 3
127 3
128 3
129 3
130 3
131 3
132 3
133 3
134 3
135 3
136 3
137 3
138 3
139 3
140 3
141 3
142 3
143 3
144 3
145 3
146 3
147 3
148 3
149 3
150 3
151 3
152 3
153 3
154 3
155 3
156 3
157 3
158 3
159 3
160 3
161 3
162 3
163 3
164 3
165 3
166 3
167 3
168 3
169 3
170 3
171 3
172 3
173 3
174 3
175 3
176 3
177 3
178 3
179 3
180 3
181 3
182 3
183 3
184 3
185 3
186 3
187 3
188 3
189 3
190 3
191 3
192 3
193 3
194 3
195 3
196 3
197 3
198 3
199 3
200 3
201 3
202 3
203 3
204 3
205 3
206 3
207 3
208 3
209 3
210 3
211 3
212 3
213 3
214 3
215 3
216 3
217 3
218 3
219 3
220 3
221 3
222 3
223 3
224 3
225 3
226 3
227 3
228 3
229 3
230 3
231 3
232 3
233 3
234 3
235 3
236 3
237 3
238 3
239 3
240 3
241 3
242 3
243 3
244 3
245 3
246 3
247 3
248 3
249 3
250 3
251 3
252 3
253 3
254 3
255 3
256 3
257 3
258 3
259 3
260 3
261 3
262 3
263 3
264 3
265 3
266 3
267 3
268 3
269 3
270 3
271 3
272 3
273 3
274 3
275 3
276 3
277 3
278 3
279 3
280 3
281 3
282 3
283 3
284 3
285 3
286 3
287 3
288 3
289 3
290 3
291 3
292 3
293 3
294 3
295 3
296 3
297 3
298 3
299 3
300 3
301 3
302 3
303 3
304 3
305 3
306 3
307 3
308 3
309 3
310 3
311 3
312 3
313 3
314 3
315 3
316 3
317 3
318 3
319 3
320 3
321 3
322 3
323 3
324 3
325 3
326 3
327 3
328 3
329 3
330 3
331 3
332 3
333 3
334 3
335 3
336 3
337 3
338 3
339 3
340 3
341 3
342 3
343 3
344 3
345 3
346 3
347 3
348 3
349 3
350 3
351 3
352 3
353 3
354 3
355 3
356 3
357 3
358 3
359 3
360 3
361 3
362 3
363 3
364 3
365 3
366 3
367 3
368 3
369 3
370 3
371 3
372 3
373 3
374 3
375 3
376 3
377 3
378 3
379 3
380 3
381 3
382 3
383 3
384 3
385 3
386 3
387 3
388 3
389 3
390 3
391 3
392 3
393 3
394 3
395 3
396 3
397 3
398 3
399 3
400 3
401 3
402 3
403 3
404 3
405 3
406 3
407 3
408 3
409 3
410 3
411 3
412 3
413 3
414 3
415 3
416 3
417 3
418 3
419 3
420 3
421 3
422 3
423 3
424 3
425 3
426 3
427 3
428 3
429 3
430 3
431 3
432 3
433 3
434 3
435 3
436 3
437 3
438 3
439 3
440 3
441 3
442 3
443 3
444 3
445 3
446 3
447 3
448 3
449 3
450 3
451 3
452 3
453 3
454 3
455 3
456 3
457 3
458 3
459 3
460 3
461 3
462 3
463 3
464 3
465 3
466 3
467 3
468 3
469 3
470 3
471 3
472 3
473 3
474 3
475 3
476 3
477 3
478 3
479 3
480 3
481 3
482 3
483 3
484 3
485 3
486 3
487 3
488 3
489 3
490 3
491 3
492 3
493 3
494 3
495 3
496 3
497 3
498 3
499 3
500 3
501 3
502 3
503 3
504 3
505 3
506 3
507 3
508 3
509 3
510 3
511 3
512 3
513 3
514 3
515 3
516 3
517 3
518 3
519 3
520 3
521 3
522 3
523 3
524 3
525 3
526 3
527 3
528 3
529 3
530 3
531 3
532 3
533 3
534 3
535 3
536 3
537 3
538 3
539 3
540 3
541 3
542 3
543 3
544 3
545 3
546 3
547 3
548 3
549 3
550 3
551 3
552 3
553 3
554 3
555 3
556 3
557 3
558 3
559 3
560 3
561 3
562 3
563 3
564 3
565 3
566 3
567 3
568 3
569 3
570 3
571 3
572 3
573 3
574 3
575 3
576 3
577 3
578 3
579 3
580 3
581 3
582 3
583 3
584 3
585 3
586 3
587 3
588 3
589 3
590 3
591 3
592 3
593 3
594 3
595 3
596 3
597 3
598 3
599 3
600 3
601 3
602 3
603 3
604 3
605 3
606 3
607 3
608 3
609 3
610 3
611 3
612 3
613 3
614 3
615 3
616 3
617 3
618 3
619 3
620 3
621 3
622 3
623 3
624 3
625 3
626 3
627 3
628 3
629 3
630 3
631 3
632 3
633 3
634 3
635 3
636 3
637 3
638 3
639 3
640 3
641 3
642 3
643 3
644 3
645 3
646 3
647 3
648 3
649 3
650 3
651 3
652 3
653 3
654 3
655 3
656 3
657 3
658 3
659 3
660 3
661 3
662 3
663 3
664 3
665 3
666 3
667 3
668 3
669 3
670 3
671 3
672 3
673 3
674 3
675 3
676 3
677 3
678 3
679 3
680 3
681 3
682 3
683 3
684 3
685 3
686 3
687 3
688 3
689 3
690 3
691 3
692 3
693 3
694 3
695 3
696 3
697 3
698 3
699 3
700 3
701 3
702 3
703 3
704 3
705 3
706 3
707 3
708 3
709 3
710 3
711 3
712 3
713 3
714 3
715 3
716 3
717 3
718 3
719 3
720 3
721 3
722 3
723 3
724 3
725 3
726 3
727 3
728 3
729 3
730 3
731 3
732 3
733 3
734 3
735 3
736 3
737 3
738 3
739 3
740 3
741 3
742 3
743 3
744 3
745 3
746 3
747 3
748 3
749 3
750 3
751 3
752 3
753 3
754 3
755 3
756 3
757 3
758 3
759 3
760 3
761 3
762 3
763 3
764 3
765 3
766 3
767 3
768 3
769 3
770 3
771 3
772 3
773 3
774 3
775 3
776 3
777 3
778 3
779 3
780 3
781 3
782 3
783 3
784 3
785 3
786 3
787 3
788 3
789 3
790 3
791 3
792 3
793 3
794 3
795 3
796 3
797 3
798 3
799 3
800 3
801 3
802 3
803 3
804 3
805 3
806 3
807 3
808 3
809 3
810 3
811 3
812 3
813 3
814 3
815 3
816 3
817 3
818 3
819 3
820 3
821 3
822 3
823 3
824 3
825 3
826 3
827 3
828 3
829 3
830 3
831 3
832 3
833 3
834 3
835 3
836 3
837 3
838 3
839 3
840 3
841 3
842 3
843 3
844 3
845 3
846 3
847 3
848 3
849 3
850 3
851 3
852 3
853 3
854 3
855 3
856 3
857 3
858 3
859 3
860 3
861 3
862 3
863 3
864 3
865 3
866 3
867 3
868 3
869 3
870 3
871 3
872 3
873 3
874 3
875 3
876 3
877 3
878 3
879 3
880 3
881 3
882 3
883 3
884 3
885 3
886 3
887 3
888 3
889 3
890 3
891 3
892 3
893 3
894 3
895 3
896 3
897 3
898 3
899 3
900 3
901 3
902 3
903 3
904 3
905 3
906 3
907 3
908 3
909 3
910 3
911 3
912 3
913 3
914 3
915 3
916 3
917 3
918 3
919 3
920 3
921 3
922 3
923 3
924 3
925 3
926 3
927 3
928 3
929 3
930 3
931 3
932 3
933 3
934 3
935 3
936 3
937 3
938 3
939 3
940 3
941 3
942 3
943 3
944 3
945 3
946 3
947 3
948 3
949 3
950 3
951 3
952 3
953 3
954 3
955 3
956 3
957 3
958 3
959 3
960 3
961 3
962 3
963 3
964 3
965 3
966 3
967 3
968 3
969 3
970 3
971 3
972 3
973 3
974 3
975 3
976 3
977 3
978 3
979 3
980 3
981 3
982 3
983 3
984 3
985 3
986 3
987 3
988 3
989 3
990 3
991 3
992 3
993 3
994 3
995 3
996 3
997 3
998 3
999 3

```

```
0 3
1 2      END;
2 3      ELSE DO,
3 3          CALL SKIP$TAG (NULL, NULL);
4 2      END;
      RETURN ON;
5 2      END PASS1$PROC;
```


IS-II PL/M-80 V3.1 COMPILATION OF MODULE CONDITIONALASSEMBLY
 OBJECT MODULE REQUESTED
 MPILER INVOKED BY: PLM80 CONASM.PLM DATE(10-MAR-82) NOOBJECT PRINT(:F1: CONASM.LST) PAGELENGTH(50)

\$TITLE('EVALUATE CONDITIONAL ASSEMBLY EXPRESSION')
 CONDITIONAL\$ASSEMBLY:
 DO;

2 1 DECLARE REV820309 BYTE AT (0);

/*

WRITTEN BY JOSEPH R. GARAPPOLO 09 MAY 1982

DESCRIPTION:

This module contains the routines necessary for evaluating the conditional part of the conditional assembly statement. The error and a value of TRUE or FALSE is returned to the calling routine COND\$ASSEMBLY.

PARAMETERS : error\$ptr - pointer to the location where the error from the expression is placed, if an error is encountered

INCLUDE FILES - INC .ELD
 SYSTEM.ELD
 GETSYM.EPD
 EXPRES.EPD

*/
 \$NOLIST


```

99 1 $EJECT
    EVALUATE$CONDITION:
    PROCEDURE (ERROR$PTR)                BYTE PUBLIC;

'00 2 DECLARE ERROR$PTR
'01 2 DECLARE (ERROR$TYPE BASED ERROR$PTR) BYTE;

'02 2 DECLARE COND$OP
'03 2 DECLARE PAREN$FLAG
'04 2 DECLARE VALUE$1
'05 2 DECLARE VALUE$2
206 2 DECLARE STATUS

207 2 IF (SYM$TYPE = LPAREN$SYM)
    THEN DO;
209 3     PAREN$FLAG = TRUE;
210 3     CALL GET$SYM;
211 3     END;
212 2     ELSE PAREN$FLAG = FALSE;
        /* get first expression of condition */
213 2     CALL EXPRESSION (.VALUE$1, ABSOLUTE$SEG, WORD$LN);
214 2     ERROR$TYPE = EXPRES.ERROR;
215 2     IF (ERROR$TYPE <> NULL)
        THEN RETURN FALSE;
217 2     IF (INSTRUCTION.INST <> CONDITION$SYM)
        THEN DO; /* not conditional symbol */
219 3         IF (SYM$TYPE = RPAREN$SYM) AND (NOT(PAREN$FLAG))
        THEN DO;
221 4             ERROR$TYPE = 'P';
222 4             RETURN FALSE;
223 4             END;
        ELSE
224 3             /* test condition with one expression
                for true, ie. bit 0 high */
                IF (VALUE$1)
                THEN RETURN TRUE;
                ELSE RETURN FALSE;
        END;
226 3
227 3
228 2     /* save conditional operator */
229 2     COND$OP = SYM$TYPE;
        CALL GET$SYM;
        /* get second expression */
230 2     CALL EXPRESSION (.VALUE$2, ABSOLUTE$SEG, WORD$LN);
231 2     ERROR$TYPE = EXPRES.ERROR;
232 2     IF (ERROR$TYPE <> NULL)
        THEN RETURN FALSE;
        STATUS = FALSE;
234 2

```

```
35 2      /* compare the two expressions with the conditional operator */
37 3      IF (COND$OP = EQ$SYM)
39 3      THEN DO;
40 2          IF (VALUE$1 = VALUE$2)
42 3          THEN STATUS = TRUE;
244 3      ELSE
245 2          IF (COND$OP = NE$SYM)
247 3          THEN DO;
249 3              IF (VALUE$1 <> VALUE$2)
250 2              THEN STATUS = TRUE;
252 3          ELSE
254 3              IF (COND$OP = GE$SYM)
255 2              THEN DO;
257 3                  IF (VALUE$1 >= VALUE$2)
259 3                  THEN STATUS = TRUE;
260 2              ELSE
262 3                  IF (COND$OP = LE$SYM)
264 3                  THEN DO;
267 2                      IF (VALUE$1 <= VALUE$2)
269 2                      THEN STATUS = TRUE;
271 2                      ELSE
272 2                          IF (PAREN$FLAG = TRUE) AND (SYM$TYPE <> RPAREN$SYM)
273 2                          THEN ERROR$TYPE = 'P';
274 2                      IF (PAREN$FLAG = FALSE) AND (SYM$TYPE = RPAREN$SYM)
275 2                      THEN ERROR$TYPE = 'P';
276 2                      IF (SYM$TYPE = RPAREN$SYM)
277 2                      THEN CALL GET$SYM;
278 2                      RETURN STATUS;
279 2          END EVALUATE$CONDITION;
```

73 1 END CONDITIONAL\$ASSEMBLY;

DULE INFORMATION:

CODE AREA SIZE = 0194H 404D
VARIABLE AREA SIZE = 0009H 9D
MAXIMUM STACK SIZE = 0004H 4D
494 LINES READ
0 PROGRAM ERROR(S)

END OF PL/M-80 COMPILATION

SIS-II PL/M-80 V3.1 COMPILATION OF MODULE CONDITIONALEVALUATION
 3 OBJECT MODULE REQUESTED
 COMPILER INVOKED BY: PLM80 CONDIR. PLM DATE(10-MAR-82) NOOBJECT PRINT(:F1:CONDIT.LST) PAGELENGTH(50)

```
1      $TITLE('STATEMENT CODE GENERATION')
      CONDITIONAL$EVALUATION:
      DO,
```

```
2 1      DECLARE REV820206          BYTE AT (0);
```

```
/*
```

```
WRITTEN BY JOSEPH R. GARAPPOLO    12 OCTOBER 1981
```

DESCRIPTION:

This module contains the routines necessary for generating the code for the IF, WHILE, FOR and CASE statements. The generated assembly code is placed into 512 byte buffer called CONDITION\$BUFF, a pointer to this buffer is passed back to the calling routine.

```
INCLUDE FILES - INC      ELD
                    SYSTEM.ELD
                    GETSYM.EPD
                    FLAGS .EVD
```

```
*/
$NOLIST
```

		\$EJECT	
103	1	JUMP\$TAG:	
104	2	PROCEDURE(JUMP\$PTR)	EXTERNAL;
105	2	DECLARE JUMP\$PTR	POINTER;
		END JUMP\$TAG;	
106	1	DECLARE CONDITION\$BUFF (512)	BYTE;
107	1	DECLARE SUB\$BUFFER (30)	BYTE;
108	1	DECLARE I	BYTE;
109	1	DECLARE J	BYTE;
110	1	DECLARE NUMBER\$FLAG	BYTE;
111	1	DECLARE LITERAL\$FLAG	BYTE;
112	1	DECLARE CODE\$LNG	WORD;
113	1	DECLARE INDENTATION	BYTE;

```
$EJECT

14 1 COPY$CODE:
    PROCEDURE (SOURCE$PTR, PAD$LNG);

    /* This routine copies the code in SOURCE buffer into CONDITION$BUFF
       until the first blank character is found, then the buffer is padded
       with blanks until PAD$LNG is reached.
    */

15 2 DECLARE SOURCE$PTR          POINTER;
16 2 DECLARE PAD$LNG             BYTE;
17 2 DECLARE (SOURCE BASED SOURCE$PTR) (1) BYTE;
18 2 DECLARE I                   BYTE;

19 2 I = 0;
20 2 DO WHILE (SOURCE(I) <> ' ');
21 3   CONDITION$BUFF(CODE$LNG) = SOURCE(I);
22 3   CODE$LNG = CODE$LNG + 1;
23 3   I = I + 1;
24 3 END;
25 2 DO WHILE (I < PAD$LNG);
26 3   CONDITION$BUFF(CODE$LNG) = ' ';
27 3   CODE$LNG = CODE$LNG + 1;
28 3   I = I + 1;
29 3 END;

230 2 END COPY$CODE;
```

```

$EJECT
31 1  GET$EXPRES$STRING:          BYTE;
    PROCEDURE

/* This routine copies the expression into SUB$BUFFER and sets flags
   that reflect the type of data it finds.

   A period (.) before the first symbol indicates the symbol is to be used as
   literally or absolute data and not as an address.
*/

232 2  DECLARE ERROR$TYPE          BYTE;
233 2  ERROR$TYPE = NULL;
234 2  /* absolute data ? */
    IF (SYM$TYPE = PERIOD$SYM)
    THEN DO;
236 3      LITERAL$FLAG = TRUE;
237 3      CALL GET$SYM;
238 3  END;
239 2  /* no, use expression as address */
    ELSE LITERAL$FLAG = FALSE;
240 2  IF (SYM$TYPE = NUMBER$SYM)
    THEN NUMBER$FLAG = TRUE;
    ELSE NUMBER$FLAG = FALSE;
242 2  IF (SYM$TYPE = IDENT$SYM) OR (SYM$TYPE = NUMBER$SYM)
    THEN DO;
245 3      I = 0;
246 3      DO WHILE (IDENT.CHAR(I) <> ' ') AND (I < SYM$LENGTH);
247 4          SUB$BUFFER(I) = IDENT.CHAR(I);
248 4          I = I + 1;
249 4      END;
250 3      CALL GET$SYM;
251 3      DO WHILE (SYM$TYPE = PLUS$SYM) OR (SYM$TYPE = MINUS$SYM);
252 4          IF (SYM$TYPE = PLUS$SYM)
            THEN SUB$BUFFER(I) = '+';
            ELSE SUB$BUFFER(I) = '-';
            I = I + 1;
            CALL GET$SYM;
            J = 0;
254 4          /* copy identifier into SUB$BUFFER */
255 4          DO WHILE (IDENT.CHAR(J) <> ' ') AND (J < SYM$LENGTH);
256 4              SUB$BUFFER(I+J) = IDENT.CHAR(J);
257 4              I = I + 1;
              J = J + 1;
258 4          ;
259 5          ;
260 5          ;
261 5          ;

```

/M-80 COMPILER STATEMENT CODE GENERATION

```
62 5      END;
63 4      CALL GET$SYM;
64 4      END;
65 3      /* terminate SUB$BUFFER with blank */
66 3      SUB$BUFFER(I) = ' ';
67 2      END;
68 2      ELSE ERROR$TYPE = 'I';
        RETURN ERROR$TYPE;
69 2      END GET$EXPRES$STRING;
```



```

$EJECT
270 1 TAG$CODE:
    PROCEDURE (SOURCE$PTR ,LENGTH);

    /* This routine copies intermediate code and location tag, pointed to by
       source$ptr into condition$buffer. This intermediate code preceeds
       each line of generated code. This code flags the print routine that this
       line is generated code and it should only be listed if an error
       occurred when it was assembled or if the CODE control flag is on.
    */

    DECLARE SOURCE$PTR      POINTER;
    DECLARE LENGTH          BYTE;

    CALL COPY$CODE (. (SKIP$STATEMENT.STATEMENT$CODE.O.O.O.' '), 5);
    CALL COPY$CODE (SOURCE$PTR, LENGTH);
    CALL COPY$CODE(. (CR,LF,' '), 0);

    END TAG$CODE;

```

```

$EJECT

277 1  GEN$CODE:
    PROCEDURE (OPERAND$PTR, OPCODE1$PTR, OPCODE2$PTR);

    /* This routine generated assembly instruction whose operand is pointed to
       by OPERAND$PTR and opcodes are pointed to by OPCODE1$PTR and OPCODE2$PTR.
       Each line is preceded by the five byte intermediate code described in
       the TAG$PROC.
    */

    278 2  DECLARE OPERAND$PTR          POINTER;
    279 2  DECLARE OPCODE1$PTR         POINTER;
    280 2  DECLARE OPCODE2$PTR         POINTER;
    281 2  DECLARE (OPERAND BASED OPERAND$PTR) (1) BYTE;
    282 2  DECLARE (OPCODE1 BASED OPCODE1$PTR) (1) BYTE;
    283 2  DECLARE (OPCODE2 BASED OPCODE2$PTR) (1) BYTE;

    284 2  CALL COPY$CODE (. (SKIP$STATEMENT.STATEMENT$CODE,O.O.O,' '),INDENTATION+5);
    /* copy operand of instruction into condition$buffer */
    285 2  CALL COPY$CODE (OPERAND$PTR, 7);
    /* is there a first opcode ? */
    286 2  IF (OPCODE1$PTR <> 0)
        THEN CALL COPY$CODE(OPCODE1$PTR, 0);
    /* is there a second opcode ? */
    288 2  IF (OPCODE2$PTR <> 0)
        THEN DO;
            /* separate the opcodes by a comma */
            290 3  CONDITION$BUFF(CODE$LNG) = ',';
            291 3  CODE$LNG = CODE$LNG + 1;
            292 3  CALL COPY$CODE(OPCODE2$PTR, 0);
            293 3  END;
            /* terminate line with carriage return and line feed */
            294 2  CALL COPY$CODE(. (CR,LF,' '), 0);
            295 2  END GEN$CODE;

```

```

$EJECT

296 1  CONDITION:
    PROCEDURE (EXIT$PTR, CODE$PTR, CODE$LN$PTR, LINE$INDENT) BYTE PUBLIC;

/* PARAMETERS : exit$ptr - filled with location tag for statement exit.
               code$ptr - address of location where address of condition$buff
               is to be returned.
               code$ln$ptr - pointer to location where length of code
                           generated is stored.
               line$indent - where to indent to when generating code
                           ** makes it look nice **

This routine generated the code necessary for evaluation the conditional
part of the IF and WHILE statements
*/

297 2  DECLARE EXIT$PTR          POINTER;
298 2  DECLARE CODE$PTR          POINTER;
299 2  DECLARE CODE$LN$PTR       POINTER;
300 2  DECLARE LINE$INDENT       BYTE;

301 2  DECLARE (EXIT$JUMP BASED EXIT$PTR) (16) BYTE;
302 2  DECLARE (CODE$BUFF$PTR BASED CODE$PTR) POINTER;
303 2  DECLARE (LNG BASED CODE$LN$PTR) WORD;

304 2  DECLARE COND$OP           BYTE;
305 2  DECLARE PAREN$FLAG        BYTE;
306 2  DECLARE ERROR$TYPE        BYTE;
307 2  DECLARE I                 BYTE;

308 2  SUB$EXPRESSION:
    PROCEDURE (POSITION);

/* Generates code depending on the flag status set by get$expres$string.
   Checks if and generates different code for first and second expression
*/

309 3  DECLARE POSITION          BYTE;

310 3  ERROR$TYPE = GET$EXPRES$STRING;
/* first expression */
311 3  IF (POSITION = 1)
    THEN DO,
        IF (NUMBER$FLAG) OR (LITERAL$FLAG)
            /* use string as data */
            THEN CALL GEN$CODE( ('MVI '), ('A '), SUB$BUFFER);
313 4

```

```

115 4      /* use string as address of string */
116 4      ELSE CALL GEN$CODE(.('LDA '), SUB$BUFFER, 0);
117 3      END;
118 4      /* second expression */
119 4      ELSE DO;
120 4          IF (NUMBER$FLAG) OR (LITERAL$FLAG)
121 5              /* use string as data */
122 5              THEN CALL GEN$CODE(.('CPI '), SUB$BUFFER, 0);
123 5      ELSE DO; /* use string as address of string */
124 4          CALL GEN$CODE(.('LXI '), ('H '), SUB$BUFFER);
125 4          CALL GEN$CODE(.('CMP '), ('M '), 0);
126 4          END;
127 4      END;
128 3      END SUB$EXPRESSION;
129 2
130 2      CODE$LNG = 0;
131 2      LNG = 0;
132 2      /* set up condition$buff pointer for calling routine */
133 2      CODE$BUFF$PTR = .CONDITION$BUFF;
134 2      COND$OP = NULL;
135 2      ERROR$TYPE = NULL;
136 2      INDENTATION = LINE$INDENT;
137 2      /* generate location tag for statement exit */
138 2      CALL JUMP$TAG(EXIT$PTR);
139 2      /* condition may be surrounded by paren.'s */
140 2      IF (SYM$TYPE = LPAREN$SYM)
141 3      THEN DO;
142 3          PAREN$FLAG = TRUE;
143 3          CALL GET$SYM;
144 3          END;
145 2      ELSE PAREN$FLAG = FALSE;
146 2      /* get first expression */
147 2      CALL SUB$EXPRESSION(1);
148 2      IF (ERROR$TYPE = NULL)
149 3      THEN /* continue if no error */
150 3          IF (INSTRUCTION.INST = CONDITION$SYM)
151 3              /* conditional operator e. EQ, NE, GT, etc. */
152 3              THEN COND$OP = SYM$TYPE;
153 3      ELSE DO; /* not conditional symbol */
154 3          IF (SYM$TYPE = RPAREN$SYM) AND (NOT(PAREN$FLAG))
155 3              THEN ERROR$TYPE = 'P';
156 3          IF (ERROR$TYPE = NULL)
157 3              THEN DO; /* test condition with one expression
158 3                  for true, ie. bit 0 high */
159 3                  CALL GEN$CODE(.('RRC '), 0, 0);
160 3                  EXIT$JUMP(14) = ', ',

```


/M-80 COMPILER STATEMENT CODE GENERATION

```

090 3      END;
091 2      IF (ERROR$TYPE = NULL) AND (PAREN$FLAG = TRUE) AND
          (SYM$TYPE <> RPAREN$SYM)
          THEN ERROR$TYPE = 'P';
093 2      IF (ERROR$TYPE = NULL) AND (PAREN$FLAG = FALSE) AND
          (SYM$TYPE = RPAREN$SYM)
          THEN ERROR$TYPE = 'P';
095 2      IF (ERROR$TYPE = NULL)
          THEN LNG = CODE$LNG;
097 2      RETURN ERROR$TYPE;
398 2      END CONDITION;

```

```

399 1      $EJECT
      FOR$CODE:
      PROCEDURE (INC$PTR,EXIT$PTR, CODE$PTR, CODE$LONG$PTR, LINE$INDENT) BYTE PUBLIC;
400 2      DECLARE INC$PTR
401 2      DECLARE EXIT$PTR
402 2      DECLARE CODE$PTR
403 2      DECLARE CODE$LONG$PTR
404 2      DECLARE LINE$INDENT
      BYTE;

/* PARAMETERS : inc$ptr - buffer for location tag of generated code for
      incrementing.
      exit$ptr - buffer for location tag of statement exit.

      This routine generated the code for the FOR statement.
      ex.  FOR I = 1 TO TEMP BY 2
      */

405 2      DECLARE (INC$JUMP BASED INC$PTR) (16) BYTE;
406 2      DECLARE (EXIT$JUMP BASED EXIT$PTR) (16) BYTE;
407 2      DECLARE (CODE$BUFF$PTR BASED CODE$PTR) POINTER;
408 2      DECLARE (LNG BASED CODE$LONG$PTR) WORD;

409 2      DECLARE ERROR$TYPE
410 2      DECLARE I
411 2      DECLARE START$JUMP (16)
412 2      DECLARE CODE$JUMP (16)
413 2      DECLARE BASE$VARIABLE (20)
      BYTE;

414 2      INDENTATION = LINE$INDENT;
415 2      LNG = 0;
416 2      CODE$BUFF$PTR = CONDITION$BUFF;
417 2      CODE$LONG = 0;
418 2      ERROR$TYPE = NULL;
419 2      I = 0;
      /* check for illegal symbol */
420 2      IF (SYM$TYPE <> IDENT$SYM) AND (SYM$TYPE <> NUMBER$SYM)
      THEN RETURN 'E';
      /* fill base variable (the I in the above example) */
422 2      DO WHILE (IDENT.CHAR(I) <> ',') AND (I < SYM$LENGTH);
423 3      BASE$VARIABLE(I) = IDENT.CHAR(I);
424 3      I = I + 1;
425 3      END;
426 2      BASE$VARIABLE(I) = ' ';

      /* return to calling routine if not '=' symbol */
427 2      CALL GET$SYM;

```

```

428 2 IF (SYM$TYPE <> EQ$SYM)
      THEN RETURN 'I';

/* get expression and retrun if error */
430 2 CALL GET$SYM;
431 2 ERROR$TYPE = GET$EXPRES$STRING;
432 2 IF (ERROR$TYPE <> NULL)
      THEN RETURN ERROR$TYPE;

/* generate code to initialize base variable */
434 2 CALL GEN$CODE (.('LXI '), (.('H '), .BASE$VARIABLE));
435 2 IF (NUMBER$FLAG) OR (LITERAL$FLAG)
      THEN CALL GEN$CODE (.('MVI '), ('M '), SUB$BUFFER);
      ELSE DO,
          CALL GEN$CODE (.('LDA '), SUB$BUFFER, 0);
          CALL GEN$CODE (.('MOV '), ('M '), ('A '));
      END;

/* create and copy start location counter into condition$buffer */
441 2 CALL JUMP$TAG(.START$JUMP);
442 2 CALL TAG$CODE (.START$JUMP(8), 7);

/* return if not IO symbol */
443 2 IF (SYM$TYPE <> IO$SYM)
      THEN RETURN 'I';

/* get limit expression and return if error */
445 2 CALL GET$SYM;
446 2 ERROR$TYPE = GET$EXPRES$STRING;
447 2 IF (ERROR$TYPE <> NULL)
      THEN RETURN ERROR$TYPE;

/* Generate code for comparing base variable with limit
   Generate jump to exit statement and to programmers code */
449 2 IF (NUMBER$FLAG) OR (LITERAL$FLAG)
      THEN CALL GEN$CODE (.('MVI '), ('A '), SUB$BUFFER);
      ELSE CALL GEN$CODE (.('LDA '), SUB$BUFFER, 0);
451 2 CALL GEN$CODE (.('LXI '), ('H '), .BASE$VARIABLE);
452 2 CALL GEN$CODE (.('CMP '), ('M '), 0);
453 2 CALL JUMP$TAG(EXIT$PTR);
454 2 CALL JUMP$JUMP(14) = ' ';
455 2 EXIT$JUMP(14) = ' ';

/* generate jump to exit */
456 2 CALL GEN$CODE (.('JC '), .EXIT$JUMP(8), 0);
457 2 CALL JUMP$TAG(.CODE$JUMP);
458 2 CODE$JUMP(14) = ' ';
/* jump to programmers code */
459 2 CALL GEN$CODE (.('JMP '), CODE$JUMP(8), 0);
460 2 CODE$JUMP(14) = ' ';

```



```

/M-80 COMPILER      STATEMENT CODE GENERATION

51  2      CALL JUMP$TAG(INC$PTR);
52  2      CALL TAG$CODE (.INC$JUMP(8), 7);
53  2      CALL GEN$CODE (.('LDA '), BASE$VARIABLE, 0);

      /* if an base variable is increment is requested generate the
      code for in */
54  2      IF (SYM$TYPE = BY$SYM)
      THEN DO;
56  3          CALL GETSYM;
57  3          ERROR$TYPE = GET$EXPRES$STRING;
58  3          IF (ERROR$TYPE <> NULL)
      THEN RETURN ERROR$TYPE;

      IF (NUMBER$FLAG) OR (LITERAL$FLAG)
      THEN CALL GEN$CODE (.('ADI '), SUB$BUFFER, 0);
      ELSE DO;
472  3          CALL GEN$CODE (.('LXI '), ('H '), SUB$BUFFER);
473  4          CALL GEN$CODE (.('ADD '), ('M '), 0);
474  4          END;
475  4      END;
476  3      /* otherwise increment by 1 */
      ELSE CALL GEN$CODE (.('ADI '), ('1H '), 0);

477  2      START$JUMP(14) = ' ';
478  2      EXIT$JUMP(14) = ' ';
479  2      /* generate jump to start */
      CALL GEN$CODE (.('JNC '), START$JUMP(8), 0);
480  2      /* generate jump to exit incase of byte overflow */
      CALL GEN$CODE (.('JMP '), EXIT$JUMP(8), 0);
481  2      CALL TAG$CODE (.CODE$JUMP(8), 7);
482  2      EXIT$JUMP(14) = ' ';
483  2      LNG = CODE$LNG;
484  2      RETURN NULL;
485  2
486  2      END FOR$CODE;

```

```

$EJECT

87 1 CASE$CODE:
PROCEDURE (TBL$PTR,EXIT$PTR,CODE$PTR,CODE$LONG$PTR,LINE$INDENT) BYTE PUBLIC;
88 2 DECLARE TBL$PTR
89 2 DECLARE EXIT$PTR
90 2 DECLARE CODE$PTR
91 2 DECLARE CODE$LONG$PTR
92 2 DECLARE LINE$INDENT

/* PARAMETERS : tbl$ptr - buffer with location tag for jump table.
exit$ptr - buffer with location tag for exit statement.

This routine generated the code for the case statement.

ex. CASE JOE TO 10

*/

493 2 DECLARE (TBL$JUMP BASED TBL$PTR) (16) BYTE;
494 2 DECLARE (EXIT$JUMP BASED EXIT$PTR) (16) BYTE;
495 2 DECLARE (CODE$BUFF$PTR BASED CODE$PTR) POINTER;
496 2 DECLARE (LNG BASED CODE$LONG$PTR) WORD;

497 2 DECLARE ERROR$TYPE BYTE;
498 2 DECLARE I BYTE;
499 2 DECLARE BASE$VARIABLE (20) BYTE;

500 2 INDENTATION = LINE$INDENT;
501 2 LNG = 0;
502 2 CODE$BUFF$PTR = CONDITION$BUFF;
503 2 CODE$LONG = 0;

504 2 /* get first expression that contains case number */
505 2 ERROR$TYPE = GET$EXPRES$STRING;
IF (ERROR$TYPE <> NULL)
THEN RETURN ERROR$TYPE;

507 2 /* create jump table and exit location tags */
508 2 CALL JUMP$TAG(TBL$PTR);
CALL JUMP$TAG(EXIT$PTR);

509 2 /* generate code to move case into DE register pair */
IF (NUMBER$FLAG) OR (LITERAL$FLAG)
THEN CALL GEN$CODE ( ('LXI '), ('H '), SUB$BUFFER);
ELSE CALL GEN$CODE ( ('LHLD '), SUB$BUFFER, 0);
511 2
512 2 (ALL GEN$CODE ( ('MOV '), ('E '), ('L ')));

```

```

513 2      CALL GEN$CODE (.('MVI '), ('D '), ('O '));

514 2      /* see if the optional TO case limit is requested */
          IF (SYM$TYPE = TO$SYM)
          THEN DO;

516 3          /* generate code to check case against TO limit. */
          CALL GET$SYM;
517 3          ERROR$TYPE = GET$EXPRES$STRING;
518 3          IF (ERROR$TYPE <> NULL)
              THEN RETURN 'E';

520 3      IF (NUMBER$FLAG) OR (LITERAL$FLAG)
          THEN CALL GEN$CODE(.('MVI '), ('A '), SUB$BUFFER);
          ELSE CALL GEN$CODE(.('LDA '), SUB$BUFFER, 0);
          CALL GEN$CODE(.('CMP '), ('L '), 0);
          EXIT$JUMP(14) = ' ';
          CALL GEN$CODE(.('JC '), .EXIT$JUMP(8), 0);
          EXIT$JUMP(14) = ' ';
          END;

          /* generate code to index into jump table and transfer control
          proper case. */
          TBL$JUMP(14) = ' ';
          CALL GEN$CODE(.('LXI '), ('H '), TBL$JUMP(8));
          TBL$JUMP(14) = ' ';
          CALL GEN$CODE(.('DAD '), ('D '), 0);
          CALL GEN$CODE(.('DAD '), ('D '), 0);
          CALL GEN$CODE(.('MOV '), ('E '), ('M '));
          CALL GEN$CODE(.('INX '), ('H '), 0);
          CALL GEN$CODE(.('MOV '), ('D '), ('M '));
          CALL GEN$CODE(.('XCHG '), 0, 0);
          CALL GEN$CODE(.('PCHL '), 0, 0);
          TBL$JUMP(14) = ' ';

          LNG = CODE$LNG;
          RETURN NULL;

541 2      END CASE$CODE;

542 1      END CONDITIONAL$EVALUATION;

```

MODULE INFORMATION.

CODE AREA SIZE	= 0A51H	2641D
VARIABLE AREA SIZE	= 029DH	669D
MAXIMUM STACK SIZE	= 0008H	8D

922 LINES READ
0 PROGRAM ERROR(S)

3 OF PL/M-80 COMPILATION

M-80 COMPILER MACRO PROCESSOR

S-II PL/M-80 V3.1 COMPILATION OF MODULE MACROPROCESSOR
 IECT MODULE PLACED IN : F4:MACRO.OBJ
 IPILER INVOKED BY: PLM80 MACRO. PLM DATE(29-MAR-82) OBJECT(: F4:MACRO.OBJ) PRINT(: F1:MACRO.LST) DEBUG PAGELENGTH(50)

```

1      $TITLE('MACRO PROCESSOR ')
      MACRO$PROCESSOR:
      DO,

2      1      DECLARE REV820327          BYTE AT(O);
          /* WRITTEN BY JOSEPH R. GARAPPOLO      20 MARCH 1982

          INCLUDE FILES - INC .ELD
                        SYSTEM.ELD
                        GETSYM.EPD
                        SYMBOL.EPD
                        COPYSR.EPD
                        ASCII .EPD
                        SMATCH.EPD
    
```

DESCRIPTION:

This module contains the routine that manage macro processing.

```

*/
$NOLIST
    
```

```

$EJECT
214 1 COPY$BUFFS:
215 2 PROCEDURE (DEST$PTR, SRC$PTR, COUNT) EXTERNAL;
216 2 DECLARE DEST$PTR POINTER;
217 2 DECLARE SRC$PTR POINTER;
218 2 DECLARE COUNT WORD;
219 2 END COPY$BUFFS;

219 1 JUMP$TAG:
220 2 PROCEDURE (TAG$PTR) EXTERNAL;
221 2 DECLARE TAG$PTR POINTER;
222 2 END JUMP$TAG;

222 1 DECLARE NO$LEVELS LITERALLY '10';
223 1 DECLARE M$LEVEL BYTE;
224 1 DECLARE COUNT WORD;
225 1 DECLARE I BYTE;

226 1 DECLARE ACTUAL$SYM$LN$G BYTE EXTERNAL;
227 1 DECLARE SYM$START BYTE EXTERNAL;
228 1 DECLARE ACTIVE$MACRO$FLAG BYTE PUBLIC;

229 1 DECLARE MASTER$MACRO$PTR POINTER;
230 1 DECLARE MACRO$LIMIT WORD;

231 1 DECLARE MACRO$BUFFER POINTER;
232 1 DECLARE (MACRO BASED MACRO$BUFFER) STRUCTURE(
233 1 BODY$PTR POINTER);

233 1 DECLARE BODY$PTR POINTER;
234 1 DECLARE (BODY$BUFF BASED BODY$PTR)(1) BYTE;

235 1 DECLARE FORMAL$PARAM$PTR POINTER;
236 1 DECLARE (FORMAL$PARAM BASED FORMAL$PARAM$PTR) STRUCTURE(
237 1 NO$FORMAL$PARAMS BYTE,
238 1 NO$LOCAL$PARAMS BYTE,
239 1 BUFF (1) BYTE);

239 1 DECLARE ACTUAL$PARAM$PTR POINTER;
240 1 DECLARE (ACTUAL$PARAM BASED ACTUAL$PARAM$PTR) STRUCTURE(
241 1 NO$PARAMS BYTE,
242 1 DUMMY BYTE,
243 1 BUFF (1) BYTE);

243 1 DECLARE MACRO$CNTL$BLOCK (11) STRUCTURE(
244 1 BUFF$PTR POINTER,

```

BUFF\$CNT
 ACTUAL\$PARAM\$PTR
 ACTUAL\$LNG

 WORD,
 POINTER,
 WORD);

70 1 DECLARE PARAM\$LIST
 NO\$PARAMS
 DUMMY
 BUFF (256)

 STRUCTURE(
 BYTE,
 BYTE,
 BYTE);

/M-80 COMPILER MACRO PROCESSOR

```

$EJECT

71 1  SKIP$LINE:
72 2  PROCEDURE (ERROR$TYPE),
      DECLARE ERROR$TYPE          BYTE;

/* This routine generates the intermediate code header for macro
   statement and fills in the error field that is used by the print
   routine.
   This header is copied into line$buff$2 which is five bytes long
   and attached to the front of line$buffer.
   */

273 2  IF (ACTIVE$MACRO$FLAG)
275 2  THEN LINE$BUFF$2(1) = STATEMENT$LINE OR MACRO$LINE;
276 2  ELSE LINE$BUFF$2(1) = STATEMENT$LINE;
277 2  LINE$BUFF$2(0) = SKIP$STATEMENT;
278 2  LINE$BUFF$2(2) = ERROR$TYPE;
279 2  LINE$BUFF$2(3) = NULL;
280 2  LINE$BUFF$2(4) = NULL;
      /* copy to intermediate starting at line$buff$2 */
      CALL COPY$SOURCE (.LINE$BUFF$2, OFFFHH);

281 2  END SKIP$LINE;

```



```

232 1  MACRO$INIT:
      PROCEDURE
      /* This procedure initializes the macro$buffer to take 1/3 the symbol table
      */
233 2  MASTER$MACRO$PTR = SYM$TBL$PTR + (2 * (TABLE$SIZE / 3));
      /* reset number of symbols allowed in symbol table to 2/3 the original
      amount. This is for the first pass only */
234 2  NO$SYMBOLS = ((NO$OF$SYM * 2) / 3);
      /* address of macro overflow */
235 2  MACRO$LIMIT = SYM$TBL$PTR + TABLE$SIZE;
236 2  M$LEVEL = 0;
237 2  ACTIVE$MACRO$FLAG = OFF;
238 2  END MACRO$INIT;
```

\$EJECT

```

39  1      PARAM$COPY:
      PROCEDURE (PARAM$CNTL$PTR);

70  2      DECLARE PARAM$CNTL$PTR      POINTER;
71  2      DECLARE (PARAM BASED PARAM$CNTL$PTR) STRUCTURE(
      NO$FORMAL$PARAMS      BYTE,
      NO$LOCAL$PARAMS      BYTE,
      BUFF (1)              BYTE);

72  2      I = 0;
      /* fill actual parameter buffer from symbol buffer until first blank
      or until symbol length */
293  2      DO WHILE (IDENT.CHAR(I) <> ' ') AND (I < SYM$LENGTH);
294  3          PARAM.BUFF(COUNT) = IDENT.CHAR(I);
295  3          COUNT = COUNT + 1;
296  3          I = I + 1;
297  3      END;
      /* indicate end of identifier name */
298  2      PARAM.BUFF(COUNT) = EOIDENT;
299  2      COUNT = COUNT + 1;
300  2      END PARAM$COPY;

```

\$EJECT

```

301 1  MACRO$DEFINE:
PROCEDURE (M$IDENT$PTR) PUBLIC;
/* This procedure puts the body of the into the macro buffer */

302 2  DECLARE M$IDENT$PTR POINTER;
303 2  DECLARE (MACRO$IDENT BASED M$IDENT$PTR) STRUCTURE(
TYPE
STATUS
VALUE
DEPTH
CHAR (SYM$LENGTH)
DECLARE STATUS
DECLARE MACRO$STATUS
DECLARE MACRO$CNT
DECLARE LOCAL$FLAG
DECLARE DEFINE$CNT
DECLARE ERROR$TYPE
DECLARE TEMP$COUNT
DECLARE TEMP$MACRO$BUFFER

304 2
305 2
306 2
307 2
308 2
309 2
310 2
311 2

312 2  GET$PARAMS:
PROCEDURE (PARAM$CNTL$PTR) BYTE;
/* This procedure reads the current line which contains either local
identifiers or actual parameters and fill the appropriate buffer.
*/

313 3  DECLARE PARAM$CNTL$PTR POINTER;
314 3  DECLARE (PARAM BASED PARAM$CNTL$PTR) STRUCTURE(
NO$FORMAL$PARAMS
NO$LOCAL$PARAMS
BUFF (1)

/* get the parameters */
DO FOREVER;
CALL GET$SYM;
IF (SYM$TYPE = IDENT$SYM)
THEN DO;
/* the parameter is an identifier */
CALL PARAM$COPY (PARAM$CNTL$PTR);
END;
ELSE IF (SYM$TYPE = EOL$SYM) OR (SYM$TYPE = COMMENT$SYM)
THEN RETURN NULL;
ELSE RETURN 'I'; /* illegal symbol as parameter */

```

```

34  4      PARAM. NO$FORMAL$PARAMS = PARAM. NO$FORMAL$PARAMS + 1;
35  4      CALL GET$SYM;
36  4      IF (SYM$TYPE <> COMMA$SYM)
37  5      THEN DO;
38  5          IF (SYM$TYPE = EOL$SYM) OR (SYM$TYPE = COMMENT$SYM)
39  5          THEN RETURN NULL;
40  5          ELSE RETURN 'I'; /* illegal symbol */
41  5      END;
42  4      END /* WHILE */;
43  3      END GET$PARAMS;
44  2      GET$FORMAL$PARAMS:
45  2      PROCEDURE
46  2      /* This procedure get the actual parameters that may optionally be
47  2      supplied in the macro definition line
48  2      */
49  3      DECLARE ERROR$TYPE      BYTE;
50  3      /* set up base variable pointer */
51  3      FORMAL$PARAM$PTR = MACRO$BUFFER + 2;
52  3      /* get the parameters */
53  3      COUNT = 0;
54  3      FORMAL$PARAM. NO$FORMAL$PARAMS = 0;
55  3      FORMAL$PARAM. NO$LOCAL$PARAMS = 0;
56  3      ERROR$TYPE = GET$PARAMS (FORMAL$PARAM$PTR);
57  3      /* set up macro and local buffer pointers */
58  3      MACRO. BODY$PTR = . FORMAL$PARAM. BUFF(COUNT);
59  3      BODY$PTR = MACRO. BODY$PTR;
60  3      BODY$BUFF (0) = EOMACRO;
61  3      LOCAL$FLAG = ON;
62  3      RETURN ERROR$TYPE;
63  3      END GET$FORMAL$PARAMS;
64  2      GET$LOCAL$SYMBOLS:
65  2      PROCEDURE
66  2      /* This procedure get the local identifiers. all LOCAL
67  2      identifier lines must appear in consecutive lines and before
68  2      the first line of the macro body.
69  2      */
70  3      DECLARE ERROR$TYPE      BYTE;
71  3      DECLARE TEMP$NO$PARAMS  BYTE;
72  3      IF NOT(LOCAL$FLAG)
73  3      /* This local line did not appear in proper place */
74  3      THEN RETURN 'I';

```

```

32 3 TEMP$N$PARAMS = FORMAL$PARAM.N$FORMAL$PARAMS;
33 3 ERROR$TYPE = GET$PARAMS (FORMAL$PARAM$PTR);
34 3 FORMAL$PARAM.N$LOCAL$PARAMS = FORMAL$PARAM.N$FORMAL$PARAMS -
    TEMP$N$PARAMS;
    /* set up macro body pointer and set first location to end-of-macro
       in case of null macro body */
35 3 MACRO.BODY$PTR = . FORMAL$PARAM.BUFF(COUNT);
36 3 BODY$PTR = MACRO.BODY$PTR;
37 3 BODY$BUFF (0) = EOMACRO;

38 3 RETURN ERROR$TYPE;
39 3 END GET$LOCAL$SYMBOLS;

360 2 GET$MACRO$BODY:
    PROCEDURE
    /* This procedure fills the macro buffer with the contents of the macro
       body. comments and trailing spaces of a line are skipped
       */
    DECLARE CHAR BYTE;
    DECLARE SPACE$FIELD BYTE;

    /* local symbol can no longer be declared */
    LOCAL$FLAG = OFF;

    /* if identifier is a nested macro definition increment nested macro
       count so that the current definition can be terminated on the
       proper endmacro statement. */
    IF (SYM$TYPE = IDENT$SYM)
    THEN DO;
        CALL GET$SYM;
        IF (SYM$TYPE = MACRO$SYM)
        THEN MACRO$CNT = MACRO$CNT + 1;
        END;
    /* if symbol is an endmacro symbol determine if it belongs to the
       outer macro definition */
    ELSE IF (SYM$TYPE = ENDMACRO$SYM)
    THEN DO;
        IF (MACRO$CNT = 0)
        THEN RETURN FALSE; /* belong to outer macro */
        ELSE MACRO$CNT = MACRO$CNT - 1;
        END;

    /* to save room in macro buffer skip trailing blanks and comments
       started by ', ' */
    SPACE$FIELD = OFFH;

```

```

17 3      CALL G$CHAR$INIT(OFFH);
18 3      CHAR = GET$CHAR;
19 3      DO WHILE (CHAR <> CR) AND (CHAR <> EOF$FILE) AND (CHAR <> ', ');
20 4      IF (CHAR = ', ')
21 5      THEN DO;
22 5          IF (SPACE$FIELD = OFFH)
23 6          THEN SPACE$FIELD = G$CHAR$CNT;
24 5      END;
25 5      ELSE SPACE$FIELD = OFFH;
26 4      CHAR = GET$CHAR;
27 4      END /* WHILE */;

38 3      IF (SPACE$FIELD = OFFH)
39 3      THEN SPACE$FIELD = G$CHAR$CNT;
390 3      CALL COPY$BUFFS (. BODY$BUFF(DEFINE$CNT), . LINE$BUFF, . SPACE$FIELD);
391 3      CALL COPY$BUFFS (. BODY$BUFF(DEFINE$CNT + SPACE$FIELD), . (CRLF), 2);
392 3      DEFINE$CNT = DEFINE$CNT + SPACE$FIELD + 2;

393 3      RETURN TRUE;
394 3      END GET$MACRO$BODY;

395 2      IF (SYM$LOOKUP(M$IDENT$PTR) = SYM$EXSIST) AND
      (MACRO$IDENT.TYPE <> MACRO$SYM)
      THEN DO; /* this identifier was previously defined other than
      a macro. Return to calling routine*/
      CALL SKIP$LINE('M');
      RETURN;
      END;

397 3
398 3
399 3

400 2      /* reset nested macro counter and macro body code index */
401 2      MACRO$CNT = 0;
402 2      DEFINE$CNT = 0;
      MACRO$BUFFER = MASTER$MACRO$PTR;
      /* Get the actual parameters and copy the line to the intermediate
      file */
403 2      CALL SKIP$LINE(GET$FORMAL$PARAMS);
404 2      STATUS = TRUE;
405 2      MACRO$STATUS = TRUE;

406 2      /* fill the macro buffer with the macro body and define local symbols */
      DO WHILE (STATUS) AND (MACRO$STATUS) AND (SYM$TYPE <> EOF$SYM) AND
      (SYM$TYPE <> END$SYM);
      /* save count and macro$buffer in case destroyed by get$macro$line */
407 3      TEMP$COUNT = COUNT;
408 3      TEMP$MACRO$BUFFER = MACRO$BUFFER;
409 3      STATUS = GET$LINE;

```

```

10 3      /* reset macro$buffer and count */
11 3      MACRO$BUFFER = TEMP$MACRO$BUFFER,
12 3      FORMAL$PARAM$PTR = MACRO$BUFFER + 2,
13 3      BODY$PTR = MACRO.BODY$PTR,
14 3      COUNT = TEMP$COUNT,
15 3      CALL GET$SYM,
16 3      ERROR$TYPE = NULL,
17 3      IF (BYM$TYPE = LOCAL$SYM)
18 3          /* define local symbols */
19 3          THEN ERROR$TYPE = GET$LOCAL$SYMBOLS,
20 3          /* put line into macro buffer */
21 3          ELSE MACRO$STATUS = GET$MACRO$BODY,
22 3          CALL SKIP$LINE (ERROR$TYPE),
23 3          END /* WHILE */
24 3          BODY$BUFF(DEFINE$CNT) = EOMACRO,
25 3          DEFINE$CNT = DEFINE$CNT + 1,
26 3          /* enter or update symbol table with macro information */
27 3          IF (SYM$LOOKUP(M$IDENT$PTR) = SYM$EXIST)
28 3              THEN DO,
29 3                  MACRO$IDENT.VALUE = MACRO$BUFFER,
30 3                  STATUS = BYM$UPDATE(M$IDENT$PTR),
31 3                  END,
32 3              ELSE DO,
33 3                  MACRO$IDENT.TYPE = MACRO$SYM,
34 3                  MACRO$IDENT.STATUS = NULL,
35 3                  MACRO$IDENT.VALUE = MACRO$BUFFER,
36 3                  STATUS = BYM$ENTRY(M$IDENT$PTR),
37 3                  END,
38 3              MASTER$MACRO$PTR = .BODY$BUFF(DEFINE$CNT),
39 3              END MACRO$DEFINE,

```

\$EJECT

```

36 1  MACRO$EXPAND:
PROCEDURE (M$BUFFER$PTR) PUBLIC;
/* this procedure sets up the macro$cnt1$block for macro expansion */

37 2  DECLARE M$BUFFER$PTR
38 2  DECLARE ERROR$TYPE          POINTER;
                                     BYTE;

39 2  GET$ACTUAL$PARAMS:
PROCEDURE
/* This procedure reads the current line which has the option of
   declaring formal parameters used in the macro body at macro
   macro expansion */
                                     BYTE;

440 3  DECLARE STATUS              BYTE;

441 3  COUNT = 0;
442 3  PARAM$LIST.NO$PARAMS = 0;

443 3  /* get the parameters */
DO FOREVER;

444 4  IF (SYM$TYPE = IDENT$SYM)
THEN DO;
/* the parameter is an identifier */
CALL PARAM$COPY (.PARAM$LIST);
END;
ELSE IF (SYM$TYPE = QUOTE$SYM)
THEN DO;
/*the parameter is a string surrounded by quotes */
IF (Q$ASCII(.PARAM$LIST.BUFF(COUNT),20,BYTE$LNQ))
THEN DO;
COUNT = COUNT + ASCII$STATUS.ACT$LNQ;
PARAM$LIST.BUFF(COUNT) = EOIDENT;
COUNT = COUNT + 1;
END;
ELSE RETURN 'S';
END;
ELSE IF (SYM$TYPE = COMMA$SYM)
THEN DO;
PARAM$LIST.BUFF(COUNT) = EOIDENT;
COUNT = COUNT + 1;
END;
ELSE IF (SYM$TYPE = EOL$SYM) OR
(SYM$TYPE = COMMENT$SYM)

```



```

465 4      THEN RETURN NULL;
466 4      ELSE RETURN 'I';
467 4      PARAM$LIST.NO$PARAMS = PARAM$LIST.NO$PARAMS + 1;
468 4      CALL GET$SYM;
469 4      IF (SYM$TYPE <> COMMA$SYM)
470 5      THEN DO;
471 5          IF (SYM$TYPE = EOL$SYM) OR (SYM$TYPE = COMMENT$SYM)
472 5          THEN RETURN NULL;
473 5          ELSE RETURN 'I'; /* illegal symbol*/
474 4      END;
475 4      CALL GET$SYM;
476 4      END /* WHILE */;
477 3      END GET$ACTUAL$PARAMS;
478 3      CREATE$LOCAL$SYMBOLS:          BYTE;
479 3      PROCEDURE                      BYTE;
480 3
481 3      /* This procedure creates local symbols using the routine jump$tag.
482 3      These symbols are placed in the format parameter list.
483 3      */
484 3      DECLARE LOCAL$TAG (16)          BYTE;
485 3      DECLARE FORMAL$PARAMS          BYTE;
486 3      FORMAL$PARAM$PTR = MACRO$CNTL$BLOCK(M$LEVEL).BUFF$PTR + 2;
487 3      FORMAL$PARAMS = FORMAL$PARAM.NO$FORMAL$PARAMS -
488 3      FORMAL$PARAM.NO$LOCAL$PARAMS;
489 3      IF (PARAM$LIST.NO$PARAMS > FORMAL$PARAMS)
490 3      THEN RETURN 'N';
491 3      IF (FORMAL$PARAMS <> PARAM$LIST.NO$PARAMS)
492 3      THEN DO;
493 3          /* substitute with empty slots */
494 3          I = PARAM$LIST.NO$PARAMS;
495 3          DO WHILE (I < FORMAL$PARAMS);
496 3              PARAM$LIST.BUFF(COUNT) = EOIDENT;
497 3              COUNT = COUNT + 1;
498 3              I = I + 1;
499 3          END;
500 3      END;
501 3      FORMAL$PARAMS = FORMAL$PARAM.NO$LOCAL$PARAMS;
502 3      IF (FORMAL$PARAMS <> 0)
503 3      THEN DO;
504 3          I = 0;
505 3          DO WHILE (I < FORMAL$PARAMS);
506 3              CALL JUMP$TAG(.LOCAL$TAG);
507 3              LOCAL$TAG(14) = EOIDENT;
508 3              I = I + 1;
509 3          END;
510 3      END;

```

```

/M-80 COMPILER      MACRO PROCESSOR

300 5      CALL COPY$BUFFS (.PARAM$LIST.BUFF(COUNT),
301 5      .LOCAL$TAG(8), 7),
302 5      COUNT = COUNT + 7,
303 5      I = I + 1,
304 4      END,
305 3      RETURN NULL,
306 3      END CREATE$LOCAL$SYMBOLS;

307 2      IF (M$LEVEL = NO$LEVELS)
308 2      THEN CALL SKIP$LINE ('N');
309 2      ELSE DO,
310 3          M$LEVEL = M$LEVEL + 1,
311 3          MACRO$CNTL$BLOCK(M$LEVEL).BUFF$PTR = M$BUFFER$PTR,
312 3          MACRO$CNTL$BLOCK(M$LEVEL).BUFF$CNT = 0,
313 3          ERROR$TYPE = GET$ACTUAL$PARAMS,
314 3          IF (ERROR$TYPE = NULL)
315 3          THEN ERROR$TYPE = CREATE$LOCAL$SYMBOLS,
316 3          CALL SKIP$LINE (ERROR$TYPE),
317 3          IF (ERROR$TYPE <> NULL)
318 3          THEN DO,
319 4              M$LEVEL = M$LEVEL - 1,
320 4              RETURN,
321 4          END,
322 3          /* store the formal and local parameters on the stack that
323 3          starts at the end of the symbol table and works up */
324 3          MACRO$LIMIT = MACRO$LIMIT - (COUNT + 2),
325 3          MACRO$CNTL$BLOCK(M$LEVEL).ACTUAL$PARAM$PTR = MACRO$LIMIT,
326 3          MACRO$CNTL$BLOCK(M$LEVEL).ACTUAL$LEN = COUNT+2,
327 3          CALL COPY$BUFFS(MACRO$LIMIT, .PARAM$LIST, COUNT+2),
328 2          /* set active macro flag on so that getline will call
329 2          get$macro$line to get the lines created by macro expansion
330 2          */
331 2          ACTIVE$MACRO$FLAG = ON,
332 2          END,
333 2      END MACRO$EXPAND;

```

\$EJECT

```

29 1  GET$MACRO$LINE:
      PROCEDURE

30 2  DECLARE CNT
31 2  DECLARE LINE$CNT
32 2  DECLARE CHAR
33 2  DECLARE STATUS
34 2  DECLARE I$MACRO
35 2  DECLARE I$LINE
36 2  DECLARE E$LINE
37 2  DECLARE M$LINE (LINE$SIZE)

      BYTE PUBLIC;

      WORD;
      BYTE;
      BYTE;
      BYTE;
      BYTE;
      BYTE;
      BYTE;

538 2  PARAM$MATCH:
      PROCEDURE;
539 3  DECLARE MATCH$PARAM      BYTE;
540 3  DECLARE LNG              BYTE;

541 3  PARAM$INDEX:
      PROCEDURE (BUFF$PTR);
      /* this procedure sets COUNT to index into buff where the
         parameter number specified by match$param starts and sets
         lng to the length of the parameter
      */
542 4  DECLARE BUFF$PTR          POINTER;
543 4  DECLARE (BUFF BASED BUFF$PTR) (1) BYTE;
544 4  DECLARE PARAM            BYTE;

      CNT = 0;
      PARAM = 0;
      DO WHILE (PARAM <= MATCH$PARAM);
      COUNT = CNT;
      LNG = 0;
      DO WHILE (BUFF(CNT) <> E$IDENT);
      CNT = CNT + 1;
      LNG = LNG + 1;
      END;
      CNT = CNT + 1;
      PARAM = PARAM + 1;
      END;
      END PARAM$INDEX;

      FIND$MATCH:
      PROCEDURE
      /* this procedure searches the actual parameter list of each
         level of nested macros starting a the current level for a

```

/M-80 COMPILER MACRO PROCESSOR

```

59 4      parameter match.
60 4      */
61 4      DECLARE LEVELCNT      BYTE;
62 5      LEVELCNT = M$LEVEL;
63 5      DO WHILE (LEVELCNT <> 0);
64 5      FORMAL$PARAM$PTR = MACRO$CNTL$BLOCK(LEVELCNT).BUFF$PTR + 2;
65 5      ACTUAL$PARAM$PTR = MACRO$CNTL$BLOCK(LEVELCNT).ACTUAL$PARAM$PTR;
66 5      MATCH$PARAM = 0;
67 6      DO WHILE (MATCH$PARAM < FORMAL$PARAM.N$FORMAL$PARAMS);
68 6      CALL PARAM$INDEX(. FORMAL$PARAM.BUFF(0));
69 6      IF (STRING$MATCH(. IDENT.CHAR, . FORMAL$PARAM.BUFF(COUNT),
70 6      SYM$LENGTH) = MATCH)
71 6      THEN RETURN TRUE;
72 6      MATCH$PARAM = MATCH$PARAM + 1;
73 6      END;
74 6      LEVELCNT = LEVELCNT - 1;
75 6      END;
76 6      RETURN FALSE;
77 6      END FIND$MATCH;

78 6      /* start parameter match */
79 6      I$MACRO = 0;
80 6      I$LINE = 0;
81 6      CALL GET$SYM;
82 6      DO WHILE (SYM$TYPE <> EOL$SYM);
83 6      IF (SYM$TYPE = IDENT$SYM)
84 6      THEN DO;
85 6      IF (FIND$MATCH)
86 6      THEN DO;
87 6      CALL PARAM$INDEX(. ACTUAL$PARAM.BUFF);
88 6      E$LINE = SYM$START;
89 6      IF (E$LINE > I$LINE)
90 6      THEN DO;
91 6      CALL COPY$BUFFS(. M$LINE(I$MACRO),
92 6      .LINE$BUFF(I$LINE), E$LINE-I$LINE);
93 6      I$MACRO = I$MACRO + (E$LINE-I$LINE);
94 6      END;
95 6      IF (LNG <> 0)
96 6      THEN DO;
97 6      CALL COPY$BUFFS(. M$LINE(I$MACRO),
98 6      .ACTUAL$PARAM.BUFF(COUNT), LNG);
99 6      I$MACRO = I$MACRO + LNG;
100 6      END;
101 6      I$LINE = G$CHAR$CNT + 1;
102 6      END;
103 6      END;

```

```

/*M-80 COMPILER      MACRO PROCESSOR

78  4      CALL GET$SYM;
79  4      END;
80  3      CALL COPY$BUFFS(.M$LINE(I$MACRO),.LINE$BUFF(I$LINE),C$CHAR$CNT+1);
81  3      CALL COPY$BUFFS(.LINE$BUFF, .M$LINE, LINE$SIZE);

82  3      END PARAM$MATCH;

83  2      GET$CURR$LINE:
PROCEDURE
/* This procedure gets the next line from the current macros body
   buffer. If the macro level reaches zero before a line is obtained
   a status of false is returned to indicate that get$line should get
   this line from the source file.
*/

604 3      DO FOREVER;
/* stay in this loop until a the line buffer is filled or
   an end-of-macro is reached at the outer macro level
*/
      MACRO$BUFFER = MACRO$CNTL$BLOCK(M$LEVEL).BUFF$PTR;
      BODY$PTR = MACRO.BODY$PTR;
      CNT = MACRO$CNTL$BLOCK(M$LEVEL).BUFF$CNT;
      ACTUAL$PARAM$PTR = MACRO$CNTL$BLOCK(M$LEVEL).ACTUAL$PARAM$PTR;
      LINE$CNT = 0;
      CHAR = NULL;
      DO WHILE (CHAR <> EOMACRO) AND (CHAR <> LF);
        CHAR = BODY$BUFF (CNT);
        LINE$BUFF(LINE$CNT) = CHAR;
        LINE$CNT = LINE$CNT + 1;
        CNT = CNT + 1;
      END;
      IF (CHAR = EOMACRO)
      THEN DO;
        MACRO$LIMIT = MACRO$LIMIT +
          MACRO$CNTL$BLOCK(M$LEVEL).ACTUAL$LN$G;
        M$LEVEL = M$LEVEL - 1;
        IF (M$LEVEL = 0)
        THEN DO;
          ACTIVES$MACRO$FLAG = OFF;
          RETURN FALSE;
        END;
      END;
      ELSE RETURN TRUE;
    END /* FOREVER */;
    END GET$CURR$LINE;

/* start get$macro$line */

```

/M-80 COMPILER MACRO PROCESSOR

```

30 2      IF (NOT(GET$CURR$LINE))
      THEN RETURN FALSE;

32 2      MACRO$CNTL$BLOCK(M$LEVEL).BUFF$CNT = CNT;
33 2      CALL G$CHAR$INIT(OFFH);
34 2      CALL PARAM$MATCH;
35 2      CALL G$CHAR$INIT(OFFH);
36 2      RETURN TRUE;
37 2      END GET$MACRO$LINE;

38 1      END MACRO$PROCESSOR;

```

MODULE INFORMATION:

```

CODE AREA SIZE      = 08B0H  2224D
VARIABLE AREA SIZE = 020BH   523D
MAXIMUM STACK SIZE = 0008H    8D
1094 LINES READ
0 PROGRAM ERROR(S)

```

END OF PL/M-80 COMPILATION

IS-II PL/M-80 V3.1 COMPILATION OF MODULE LOCATIONTAG
 OBJECT MODULE REQUESTED
 MPILER INVOKED BY: PLM80 LOCTAG.PLM DATE(10-MAR-82) NOOBJECT PRINT(:F1:LOCTAG.LST) PAGELENGTH(50)

```

1            $TITLE('CREATE LOCATION TAGS')
             LOCATION$TAG:
DO;

2    1       DECLARE REV820130                    BYTE AT(O);
             /*
             WRITTEN BY JOSEPH R. GARAPPOLO      16 JAN 1982
             INCLUDE FILES - INC    ELD

PARAMETERS: TAG$PTR - pointer to buffer that ASCII tag is to be put

DESCRIPTION:

This routine creates ASCII location tags used for structure statements
expansion and Macro local symbols.

The TAG buffer is filled with a jump instruction followed by the tag and
a collon ':' character.

             ex.    JMP    @000001:

             */
$NOLIST

```

\$EJECT

```
22 1      NUMOUT:
23 2      PROCEDURE (VALUE, BASE, BUFF$PTR, WIDTH, PAD) EXTERNAL;
24 2      DECLARE VALUE      WORD;
25 2      DECLARE BASE      BYTE;
26 2      DECLARE BUFF$PTR  POINTER;
27 2      DECLARE WIDTH    BYTE;
28 2      DECLARE PAD      BYTE;
29 2      END NUMOUT;

29 1      COPY$BUFFS:
30 2      PROCEDURE (DEST$PTR, SRC$PTR, LNG) EXTERNAL,
31 2      DECLARE DEST$PTR  POINTER;
32 2      DECLARE SRC$PTR  POINTER;
33 2      DECLARE LNG      WORD;
34 2      END COPY$BUFFS;

34 1      DECLARE TAG$NUMBER      WORD;
```


\$EJECT

35 1 JUMP\$TAG\$INIT;
 PROCEDURE

PUBLIC;

36 2 TAG\$NUMBER = 0;

37 2 END JUMP\$TAG\$INIT;

```

$EJECT
38  1      JUMP$TAG:
      PROCEDURE (TAG$PTR)                      PUBLIC;

39  2      DECLARE TAG$PTR
40  2      DECLARE (TAG$BUFF BASED TAG$PTR) (23) BYTE;
      POINTER;

41  2      CALL COPY$BUFFS (TAG$PTR, .('JMP       @'), 9);
42  2      CALL NUMOUT (TAG$NUMBER, 10, TAG$BUFF(9), 5, '0');
43  2      TAG$BUFF(14) = ':';
44  2      TAG$BUFF(15) = ',';
45  2      TAG$NUMBER = TAG$NUMBER + 1;

46  2      END JUMP$TAG;

47  1      END LOCATION$TAG;

```

MODULE INFORMATION:

```

CODE AREA SIZE        = 0051H      81D
VARIABLE AREA SIZE = 0004H      4D
MAXIMUM STACK SIZE = 0008H      8D
99 LINES READ
0 PROGRAM ERROR(S)

```

END OF PL/M-80 COMPILATION

./M-80 COMPILER MANAGE OUTPUT TO SMABO.TMP FILE

10-MAR-82 PAGE 1

IS-II PL/M-80 V3.1 COMPILATION OF MODULE COPYSOURCEFILE
OBJECT MODULE REQUESTED
MPILER INVOKED BY: PLM80 COPYSR.PLM DATE(10-MAR-82) NOOBJECT PRINT(:F1: COPYSR.LST) PAGEDLENGTH(50)

1 \$TITLE('MANAGE OUTPUT TO SMABO.TMP FILE')
 COPY\$SOURCE\$FILE:
 DO,

2 1 DECLARE REV820130 BYTE AT(O);

/*

WRITTEN BY - JOSEPH R. GARAPPOLO 21 JAN 1982

INCLUDE FILES - INC ELD
 SYSTEM.ELD
 FILE10.EPD

PARAMETERS: SRC\$BUFF\$PTR- pointer to source code buffer
 COUNT - number of bytes to be copied.

DESCRIPTION:

This routine manages the coping of source code to the intermediate
file. The source code is copied into a buffer. When the buffer
becomes full it is written out to the file SMABO.TMP on the system disk.

*/
\$NOLIST

/M-80 COMPILER MANAGE OUTPUT TO SMABO.TMP FILE

	\$EJECT				
11	1	DECLARE	AFT\$TMP		WORD EXTERNAL;
12	1	DECLARE	OUT\$BUFF (256)		BYTE;
13	1	DECLARE	CHAR\$CNT		WORD;
14	1	DECLARE	BUFF\$CNT		WORD;
15	1	DECLARE	STATUS		WORD;

\$EJECT

```
/* This routine initializes the buffer count */
```

```

1 COPY$SOURCE$INIT:
216 PROCEDURE
PUBLIC;

```

```
217 2  BUFF$CNT = 0;
```

218 2 END COPY\$SOURCE\$INIT;

```

$EJECT
/* This routine copies the buffer to disk */

19 1    EMPTY$BUFFER:
PROCEDURE;

20 2    CALL WRITE(AFT$TMP, .OUT$BUFF, BUFF$CNT, .STATUS);
21 2    BUFF$CNT = 0;

22 2    END EMPTY$BUFFER;
```

\$EJECT

/* This routine is called before exiting PASS\$1 to write the last of the
buffer to disk */

```

223 1  COPY$SOURCE$FINISH:          PUBLIC;
      PROCEDURE
224 2  IF (BUFF$CNT <> 0)
      THEN CALL EMPTY$BUFFER;
226 2  END COPY$SOURCE$FINISH;

```

```

$EJECT

/* This routine copies the source code into the buffer and calls
EMPTY$BUFFER to empty it.
If the COUNT = OFFFFH upon entry to this routine the contents of the source
file are copied into the buffer upto and including the first line feed
character. Otherwise count holds the number of bytes to be transferred */

227 1  COPY$SOURCE;
PROCEDURE (SRC$BUFF$PTR, COUNT)      PUBLIC;

228 2  DECLARE SRC$BUFF$PTR              POINTER;
229 2  DECLARE COUNT                      WORD;
230 2  DECLARE (SRC$BUFF BASED SRC$BUFF$PTR) (1) BYTE;

231 2  CHAR$CNT = 0;
232 2  IF (COUNT = OFFFFH)
233 3  THEN DO;
234 3  DO WHILE (SRC$BUFF(CHAR$CNT) <> CR);
235 4  IF (BUFF$CNT = 256)
236 5  THEN CALL EMPTY$BUFFER;
237 4  OUT$BUFF(BUFF$CNT) = SRC$BUFF(CHAR$CNT);
238 4  BUFF$CNT = BUFF$CNT + 1;
239 4  CHAR$CNT = CHAR$CNT + 1;
240 4  END;
241 3  END;
242 2  ELSE DO;
243 3  DO WHILE (CHAR$CNT < COUNT);
244 4  IF (BUFF$CNT = 256)
245 5  THEN CALL EMPTY$BUFFER;
246 4  OUT$BUFF(BUFF$CNT) = SRC$BUFF(CHAR$CNT);
247 4  BUFF$CNT = BUFF$CNT + 1;
248 4  CHAR$CNT = CHAR$CNT + 1;
249 4  END;
250 3  END;
251 2  IF (BUFF$CNT = 256)
252 3  THEN CALL EMPTY$BUFFER;
253 2  OUT$BUFF(BUFF$CNT) = CR;
254 2  BUFF$CNT = BUFF$CNT + 1;
255 2  IF (BUFF$CNT = 256)
256 3  THEN CALL EMPTY$BUFFER;
257 2  OUT$BUFF(BUFF$CNT) = LF;
258 2  BUFF$CNT = BUFF$CNT + 1;

259 2  END COPY$SOURCE,

```



```

$EJECT
260 1 COPY$CHAR:
PROCEDURE (SRC$BUFF$PTR, COUNT) PUBLIC;
261 2 DECLARE SRC$BUFF$PTR POINTER;
262 2 DECLARE COUNT BYTE;
263 2 DECLARE (SRC$BUFF BASED SRC$BUFF$PTR) (1) BYTE;
264 2 CHAR$CNT = 0;
265 2 DO WHILE (CHAR$CNT < COUNT) AND (SRC$BUFF(CHAR$CNT) <> ' ');
266 3 IF (BUFF$CNT = 256)
THEN CALL EMPTY$BUFFER;
268 3 OUT$BUFF(BUFF$CNT) = SRC$BUFF(CHAR$CNT);
269 3 BUFF$CNT = BUFF$CNT + 1;
270 3 CHAR$CNT = CHAR$CNT + 1;
271 3 END;
272 2 DO WHILE (CHAR$CNT < COUNT);
273 3 IF (BUFF$CNT = 256)
THEN CALL EMPTY$BUFFER;
275 3 OUT$BUFF(BUFF$CNT) = ' ';
276 3 BUFF$CNT = BUFF$CNT + 1;
277 3 CHAR$CNT = CHAR$CNT + 1;
278 3 END;
279 2 END COPY$CHAR;
280 1 END COPY$SOURCE$FILE;

```

MODULE INFORMATION:

```

CODE AREA SIZE = 01A9H 425D
VARIABLE AREA SIZE = 010DH 269D
MAXIMUM STACK SIZE = 000BH BD
459 LINES READ
0 PROGRAM ERROR(S)

```

END OF PL/M-80 COMPILATION

1-80 COMPILER STRUCTURED MACRO ASSEMBLER - PASS 2

3-II PL/M-80 V3.1 COMPILATION OF MODULE PASS2COMMAND
 OBJECT MODULE PLACED IN : F4: PASS2.OBJ
 COMPILER INVOKED BY: PLM80 PASS2.PLM DATE(27-MAR-82) OBJECT(:F4: PASS2.OBJ) PRINT(:F1: PASS2.LST) DEBUG PAGELENGTH(50)

1 \$TITLE('STRUCTURED MACRO ASSEMBLER - PASS 2')
 PASS\$2\$COMMAND:
 DO;

2 1 DECLARE REV820128 BYTE AT (0);

/* WRITTEN BY - JOSEPH R. GARAPPOLO MARCH 30, 1981

/* The main responsibility of this pass of SMABO is to define the
 first level identifier values. PASS2 ignores errors, it saves them
 PASS3 to handle.

INCLUDE FILES - INC .ELD
 SYSTEM.ELD
 LOCCNT.EPD
 SYMBOL.EPD
 ASCII .EPD
 EXPRES.EPD
 GETSYM.EPD
 CNLT .EPD

*/
 \$NOLIST

```

3      1      $EJECT
4      1      COPY$BUFFS:
5      2      PROCEDURE (DEST$PTR, SRC$PTR, COUNT) EXTERNAL;
6      2      DECLARE DEST$PTR      POINTER;
7      2      DECLARE SRC$PTR      POINTER;
8      2      DECLARE COUNT      WORD;
9      2      END COPY$BUFFS;

0      1      DECLARE IDENT$FLAG      BYTE;

```

```

261 1  MEMORY$PROC:
      PROCEDURE
      ;
      /* each memory segment CODE, DATA and ABSOLUTE have there own program
      counters */
262 2  DO CASE MEMORY. INST;
263 3      /* DSEQ - data segment */
264 4      DO;
265 5      IF SEGMENT$TYPE = DATA$SEQ
266 6      THEN;
267 7      ELSE DO;
268 8          /* update previous counter */
269 9          IF SEGMENT$TYPE = CODE$SEQ
270 10         THEN CODE$LOC$CNT = LOCATION$CNT;
271 11         ELSE ASEQ$LOC$CNT = LOCATION$CNT;
272 12         SEGMENT$TYPE = DATA$SEQ;
273 13         LOCATION$CNT = DATA$LOC$CNT;
274 14         END;
275 15     END;
276 16     /* CSEQ - code segment */
277 17     DO;
278 18     IF SEGMENT$TYPE = CODE$SEQ
279 19     THEN;
280 20     ELSE DO;
281 21         /* update previous counter */
282 22         IF SEGMENT$TYPE = DATA$SEQ
283 23         THEN DATA$LOC$CNT = LOCATION$CNT;
284 24         ELSE ASEQ$LOC$CNT = LOCATION$CNT;
285 25         SEGMENT$TYPE = CODE$SEQ;
286 26         LOCATION$CNT = CODE$LOC$CNT;
287 27         END;
288 28     END;
289 29     /* ASEQ - absolute segment */
290 30     DO;
291 31     IF SEGMENT$TYPE = ABSOLUTE$SEQ
292 32     THEN;
293 33     ELSE DO;
294 34         /* update previous segment */
295 35         IF SEGMENT$TYPE = DATA$SEQ
296 36         THEN DATA$LOC$CNT = LOCATION$CNT;
297 37         ELSE CODE$LOC$CNT = LOCATION$CNT;
298 38         END;
299 39     END;

```

1-80 COMPILER STRUCTURED MACRO ASSEMBLER - PASS 2

```
2 5          SEGMENT$TYPE = ABSOLUTE$SEG;  
3 5          LOCATION$CNT = ASEG$LOC$CNT;  
4 5          END;  
5 4          END;
```

```
/* ORG - set current location counter to new value contained in  
operand of ORG instruction */
```

```
5 3          DO;  
7 4              CALL GETSYM;  
8 4              CALL EXPRESSION(.LOCATION$CNT, NULL, WORD$LN);  
9 4          END;
```

```
300 3          END /* CASE */;
```

```
301 2          END MEMORY$PROC;
```

```

$EJECT
2 1 IDENT$PROC: REENTRANT;
   PROCEDURE

      /* define first level of identifier value */

3 2 DECLARE TEMP$IDENT STRUCTURE(
   TYPE
   STATUS
   VALUE
   MACRO$LEV
   CHAR (SYM$LENGTH)
   BYTE;
   WORD;
   BYTE;

304 2 DECLARE STATUS
305 2 DECLARE TEMP$LOC$CNT
306 2 DECLARE I

      /* this is a recursive routine, must save ident */
      CALL COPY$BUFFS(. TEMP$IDENT.CHAR, . IDENT.CHAR, SYM$LENGTH);

307 2

      /* see if symbol is already defined */
      STATUS = SYM$LOOKUP(. TEMP$IDENT);

308 2

      /* if it is external it must be an error so return */
      IF ((TEMP$IDENT.STATUS AND EXTRN$DEF) = EXTRN$DEF)
      THEN RETURN;

309 2

      /* other identifiers may have been entered into symbol table by
      PUBLIC instruction but not defined, so continue even if symbol
      exist in symbol table */

311 2 CALL GETSYM;

      /* location tag */
      IF (SYM$TYPE = COLON$SYM)
      THEN DO;
312 2 TEMP$LOC$CNT = LOCATION$CNT;
314 3 /* only get tag on line ? */
315 3 IF (IDENT$FLAG <> ON)
      /* no */
      THEN CALL PASS2$PROC;
      /* yes, skip rest of line */
      ELSE IDENT$FLAG = OFF;

317 3

      /* look symbol up again, may have call PASS2 again */
      STATUS = SYM$LOOKUP(. TEMP$IDENT);
318 3

```

1-80 COMPILER STRUCTURED MACRO ASSEMBLER - PASS 2

```

3  /* if identifier not previously defined, define it */
   IF NOT((TEMP$IDENT.STATUS AND PASS2$DEF) = PASS2$DEF) AND
   (STATUS = SYM$EXSIST))
   THEN DO;
4     TEMP$IDENT.TYPE = VAR$SYM;
5     TEMP$IDENT.STATUS = SEGMENT$TYPE OR PASS2$DEF;
6     TEMP$IDENT.VALUE = TEMP$LOC$CNT;
7     END;
8     /* do not define again, retrun PASS3 worries about
   errors */
   ELSE RETURN;
9
10    END;
11
12    ELSE
13    IF (SYM$TYPE = EQU$SYM)
14    THEN DO;
15      /* check to see if attempting to multiply define */
16      IF NOT((TEMP$IDENT.STATUS AND PASS2$DEF) = PASS2$DEF) AND
17      (STATUS = SYM$EXSIST))
18      THEN DO;
19        TEMP$IDENT.TYPE = EQU$SYM;
20        CALL GETSYM;
21        CALL EXPRESSION(.TEMP$IDENT.VALUE,ABSOLUTE$SEQ,WORD$LN);
22        TEMP$IDENT.STATUS = ABSOLUTE$SEQ OR PASS2$DEF;
23        END;
24      ELSE RETURN;
25    END;
26  ELSE
27
28  IF (SYM$TYPE = SET$SYM)
29  /* set instruction can be redefined as often as the please */
30  THEN IF (STATUS = SYM$UNDEF)
31  THEN DO;
32    TEMP$IDENT.TYPE = SET$SYM;
33    TEMP$IDENT.STATUS = NULL;
34    CALL GETSYM;
35    CALL EXPRESSION(.TEMP$IDENT.VALUE, ABSOLUTE$SEQ, WORD$LN);
36    END;
37  ELSE DO;
38    IF TEMP$IDENT.TYPE = SET$SYM
39    THEN DO;
40      CALL GETSYM;
41      CALL EXPRESSION(.TEMP$IDENT.VALUE,
42      ABSOLUTE$SEQ,WORD$LN);
43    END;
44    ELSE RETURN;
45  END;
46
47  ELSE DO;
48
49  END;
50
51  ELSE DO;

```

M-80 COMPILER STRUCTURED MACRO ASSEMBLER - PASS 2

```

15 3      /* who knows what is here lets see what is beyoud it */
16 3      CALL PASS2$PROC;
17 3      RETURN;
      END;

18 2      IF (STATUS = SYM$EXSIST)
19 2      THEN STATUS = SYM$UPDATE(. TEMP$IDENT);
20 2      ELSE STATUS = SYM$ENTRY(. TEMP$IDENT);
21 2      END IDENT$PROC;

```



```

$EJECT
62  1  DATA$PROC:
    PROCEDURE;

63  2  DECLARE DATA$VAL                WORD;
64  2  DECLARE ACT$LENGTH              BYTE;
65  2  DECLARE DATA$INST              BYTE;

66  2  DATA$INST = DATA$STORAGE.INST;
67  2  IF (DATA$INST = DB$SYM)
    THEN DO;
        /* data storage increment the location count by the expression
           in the operand field */
        CALL GET$SYM;
        CALL EXPRESSION(.DATA$VAL, ABSOLUTE$SEG, WORD$LNQ);
        LOCATION$CNT = LOCATION$CNT + DATA$VAL;
    END;

    /* DB$SYM or DW$SYM
       parse line buffer to determine how many byte or word of memory are
       used by instruction */
    ELSE DO;
        CALL GET$SYM;
        DO WHILE (SYM$TYPE <> COMMENT$SYM) AND (SYM$TYPE <> EOL$SYM);
            /* ascii string */
            IF (SYM$TYPE = QUOTE$SYM)
            THEN DO;
                ACT$LENGTH = G$ASCII$LNQ;
                IF (DATA$INST = DW$SYM) AND (ACT$LENGTH MOD 2 <> 0)
                THEN LOCATION$CNT = LOCATION$CNT + ACT$LENGTH+1;
                ELSE LOCATION$CNT = LOCATION$CNT + ACT$LENGTH;
            END;
            /* if not string must be expression that requires one or
               two words depending on instruction */
            ELSE IF (DATA$INST = DW$SYM)
            THEN LOCATION$CNT = LOCATION$CNT + 2;
            ELSE LOCATION$CNT = LOCATION$CNT + 1;

            CALL GET$SYM;
            /* get next comma or end of line */
            DO WHILE (SYM$TYPE <> COMMA$SYM) AND (SYM$TYPE <> COMMENT$SYM)
                AND (SYM$TYPE <> EOL$SYM);
            CALL GET$SYM;
        END;
    END;
END;
END;

```

'M-80 COMPILER STRUCTURED MACRO ASSEMBLER - PASS 2

'2 2 END DATA\$PROC;

```

$EJECT
93  1  EXTRN$PROC:
    PROCEDURE;

94  2  DECLARE STATUS                    BYTE;

/* enter identifiers in the external statement into the symbol table.
   The value field of for external identifiers is used for the external
   number that is used when formation the object code */
95  2  CALL GETSYM;
96  2  DO WHILE (SYM$TYPE <> EOL$SYM) AND (SYM$TYPE <> COMMENT$SYM);
97  3  IF (SYM$TYPE = IDENT$SYM)
    THEN IF (SYM$LOOKUP(SYM$PTR) = SYM$UNDEF)
    THEN DO;
        IDENT. TYPE = VAR$SYM;
        IDENT. STATUS = EXTRN$SEG OR EXTRN$DEF;
        IDENT. VALUE = EXTRN$CNT;
        EXTRN$CNT = EXTRN$CNT + 1;
        STATUS = SYM$ENTRY(SYM$PTR);
    END;
    CALL GETSYM;
    IF (SYM$TYPE = COMM$SYM)
    THEN CALL GETSYM;
    END;
98  3  END EXTRN$PROC;
99  2

```

```

$EJECT
11 1  PUBLIC$PROC:
    PROCEDURE;

12 2  DECLARE STATUS                    BYTE;

    /* enter identifiers in the public instruction into the symbol table */

13 2  CALL GET$SYM;
14 2  DO WHILE (SYM$TYPE <> EOL$SYM) AND (SYM$TYPE <> COMMENT$SYM);
15 3  IF (SYM$TYPE = IDENT$SYM)
    THEN IF (SYM$LOOKUP(. IDENT) = SYM$UNDEF)
    THEN DO;
        IDENT.TYPE = NULL;
        IDENT.STATUS = PUBLIC$DEF;
        STATUS = SYM$ENTRY(SYM$PTR);
    END;
    ELSE DO;
        IF (IDENT.STATUS AND EXTRN$DEF) <> EXTRN$DEF
        THEN DO;
            IDENT.STATUS = IDENT.STATUS OR
            PUBLIC$DEF;
            STATUS = SYM$UPDATE(. IDENT);
        END;
    END;

    CALL GET$SYM;
    IF (SYM$TYPE = COMMA$SYM)
    THEN CALL GET$SYM;
    END;

432 3
433 2  END PUBLIC$PROC;

```

```

$EJECT
34  1  PASS2$PROC:      REENTRANT;
    PROCEDURE

35  2  DECLARE STATUS  BYTE;

36  2  CALL GET$SYM;

37  2  IF (SYM$TYPE = SKIP$LINE$SYM) AND
    ((SKIP.LINE$TYPE AND NOT$MACRO$LINE) <> STATEMENT$LINE)
    THEN DO;
439  3      /* tag identifier is on line get that and only that
        if tag$line */
        IF ((SKIP.LINE$TYPE AND NOT$MACRO$LINE) = TAG$LINE)
        THEN IDENT$FLAG = ON;

441  3      CALL G$CHAR$INIT(4);
442  3      CALL PASS2$PROC;
443  3      END;

444  2  ELSE

446  2      /* only need length of instruction */
        IF SYM$TYPE = INSTRUCTION$SYM
        THEN LOCATION$CNT = LOCATION$CNT + INSTRUCTION.LENGTH;
        ELSE

448  2      IF SYM$TYPE = IDENT$SYM
        THEN CALL IDENT$PROC;
        ELSE

450  2      IF SYM$TYPE = DATA$STORAGE$SYM
        THEN CALL DATA$PROC;
        ELSE

452  2      IF SYM$TYPE = EXTRN$SYM
        THEN CALL EXTRN$PROC;
        ELSE

454  2      IF SYM$TYPE = PUBLIC$SYM
        THEN CALL PUBLIC$PROC;
        ELSE

456  2      IF (SYM$TYPE = CONTROL$SYM) AND (CONTROL.INST = TITLE$SYM)
        THEN STATUS = CNTL$PROC(2);

    END PASS2$PROC;

```

```

$EJECT
39  1  MAIN$PASS2:
    PROCEDURE
    BYTE PUBLIC;

50  2  DECLARE STATUS
    BYTE;

61  2  /* initialize program location counters, symbol table and line buffer */
62  2  CALL SEGMENT$INIT;
63  2  CALL SYM$INIT;
    CALL GET$LINE$INIT;

    /* used by intermediate code. When a skip line encountered it may be
       necessary to get an identifier and only an identifier from the line. */
464 2  IDENT$FLAG = OFF;

    /* get first line and initialize symbol type */
465 2  STATUS = GETLINE;
466 2  SYM$TYPE = SYM$TYPE$RESET;
    /* repeat loop until end of file or end statement */
467 2  DO WHILE (SYM$TYPE <> EOF$SYM) AND (SYM$TYPE <> END$SYM) AND
    (STATUS = TRUE);
468 3  CALL PASS2$PROC;
469 3  STATUS = GETLINE;
470 3  END;

471 2  RETURN STATUS;

472 2  END MAIN$PASS2;

473 1  END PASS$2$COMMAND;

```

MODULE INFORMATION:

```

CODE AREA SIZE      = 0527H   1319D
VARIABLE AREA SIZE = 000BH    8D
MAXIMUM STACK SIZE = 001CH   28D
874 LINES READ
0 PROGRAM ERROR(S)

```

END OF PL/M-80 COMPILATION

IS-II PL/M-SO V3.1 COMPILATION OF MODULE PASS3COMMAND
 JECT MODULE PLACED IN : F4:PASS3.OBJ
 MPILER INVOKED BY: PLMSO PASS3.PLM DATE(28-MAR-82) OBJECT(:F4:PASS3.OBJ) PRINT(:F1:PASS3.LST) DEBUG PAGELENGTH(50)

*TITLE('STRUCTURED MACRO ASSEMBLER - PASS 3')

PASS3\$COMMAND:
 DO;

2 1 DECLARE REV52032S BYTE AT(0);

/*

WRITTEN BY - JOSEPH R. GARAPPOLO 01 APRIL 1981

This pass of SMASO is responsible for,
 1. Defining the second and last level of identifier values,
 2. Creating the object code file, and
 3. Create the expanded source code listing.

INCLUDE FILES - INC .ELD
 SYSTEM.ELD
 LOCNT.EPD
 CNTL .EPD
 ASCII .EPD
 SYMBOL.EPD
 EXPRES.EPD
 GETSYM.EPD
 OBJECT.EPD
 PRINT .EPD

*/
 \$NOLIST

```

280 1      $EJECT
      FILE$INIT$PASS3:
281 2      PROCEDURE
      END FILE$INIT$PASS3;
      EXTERNAL;

282 1      COPY$BUFFS:
      PROCEDURE (DEST$PTR, SRC$PTR, LNG)
283 2      DECLARE DEST$PTR
284 2      DECLARE SRC$PTR
285 2      DECLARE LNG
286 2      END COPY$BUFFS;
      EXTERNAL;
      POINTER;
      POINTER;
      WORD;

287 1      DECLARE I
288 1      DECLARE BYTE$TEMP
289 1      DECLARE STATUS
290 1      DECLARE IDENT$FLAG
      BYTE;
      BYTE;
      BYTE;
      BYTE;
```



```

291 1      MEM$PROC:
      PROCEDURE
      BYTE;

      /* SMABO has three location counters on for each of the memories, CODE
      DATA and ABSOLUTE. This routine changes these counters when the
      type of memory changes */

292 2      DECLARE ERROR$TYPE          BYTE;

293 2      ERROR$TYPE = NULL;

294 2      DO CASE MEMORY. INST;

295 3          /* DSEQ - data segment */
296 4          DO;
              IF (SEGMENT$TYPE = DATA$SEQ)
                  /* return if no change */
                  THEN RETURN ERROR$TYPE;
              ELSE DO;
                  /* save previous counter */
                  IF SEGMENT$TYPE = CODE$SEQ
                      THEN CODE$LOC$CNT = LOCATION$CNT;
                      ELSE ASEQ$LOC$CNT = LOCATION$CNT;
                      SEGMENT$TYPE = DATA$SEQ;
                      SEGMENT. TYPE = DATA$SEQ;
                      LOCATION$CNT = DATA$LOC$CNT;
                      SEGMENT.OFFSET = DATA$LOC$CNT;
                  END;
              END;

298 4
299 5
301 5
302 5
303 5
304 5
305 5
306 5
307 4      END;

308 3      /* CSEQ - code segment */
309 4      DO;
          IF SEGMENT$TYPE = CODE$SEQ
              /* return if no change */
              THEN RETURN ERROR$TYPE;
              ELSE DO;
                  /* save previous counter */
                  IF SEGMENT$TYPE = DATA$SEQ
                      THEN DATA$LOC$CNT = LOCATION$CNT;
                      ELSE ASEQ$LOC$CNT = LOCATION$CNT;
                      SEGMENT$TYPE = CODE$SEQ;
                      SEGMENT. TYPE = CODE$SEQ;
                      LOCATION$CNT = CODE$LOC$CNT;
                      SEGMENT.OFFSET = CODE$LOC$CNT;
                  END;
              END;

311 4
312 5
314 5
315 5
316 5
317 5
318 5
319 5

```

M-80 COMPILER STRUCTURED MACRO ASSEMBLER - PASS 3

```

30 4      END;

31 3      /* ASEG - ABSOLUTE SEGEMENT */
32 4      DO;
33 5          IF SEGMENT$TYPE = ABSOLUTE$SEG
34 6              /* return if no change */
35 7              THEN RETURN ERROR$TYPE;
36 8      ELSE DO;
37 9          /* save previous counter */
38 10         IF SEGMENT$TYPE = DATA$SEG
39 11             THEN DATA$LOC$CNT = LOCATION$CNT;
40 12         ELSE CODE$LOC$CNT = LOCATION$CNT;
41 13         SEGMENT$TYPE = ABSOLUTE$SEG;
42 14         SEGMENT.TYPE = ABSOLUTE$SEG;
43 15         LOCATION$CNT = ASEG$LOC$CNT;
44 16         SEGMENT.OFFSET = ASEG$LOC$CNT;
45 17     END;
46 18 END;

47 19 /* ORG */
48 20 /* change memory counter to new location specified in the operand field
49 21 of the ORG instruction */
50 22 DO;
51 23     CALL GETSYM;
52 24     CALL EXPRESSION(.LOCATION$CNT, NULL, WORD$LN$);
53 25     RETURN EXPRES.ERROR;
54 26 END;

55 27 END /* CASE */;
56 28 /* tell object code formatter and tell them */
57 29 CALL NEW$SEGMENT;
58 30 RETURN ERROR$TYPE;
59 31 END MEM$PROC;

```

```

$EJECT
43  1  INST$PROC:
    PROCEDURE
        BYTE;

/* instructions can be divided into 12 categories.  From these categories
   all instruction can be constructed
*/

44  2  DECLARE ERROR$TYPE
45  2  DECLARE SINGLE$REG
46  2  DECLARE REG$PAIR
        BYTE;
        LITERALLY '1';
        LITERALLY '2';

347  2  DECLARE INST$TYPE
        BYTE;

348  2  ENCODE$REG:
    PROCEDURE (REG$TYPE, REG$MASK, SPECIAL$CASE);

/* PARAMETERS: reg$type - defines if single or double register pair
   rereg$mask - masks the section of the register code you
   want to use for this instruction.
   special$case - a register that can not be used in this
   instruction.

   this routine puts adds the register code into the opcode of the
   instruction
*/
349  3  DECLARE REG$TYPE
350  3  DECLARE REG$MASK
351  3  DECLARE SPECIAL$CASE
        BYTE;
        BYTE;
        BYTE;

352  3  IF (SYM$TYPE <> REG$SYM)
        /* symbol in symbol buffer is not a register */
        THEN ERROR$TYPE = 'R',
        ELSE
354  3  IF (REG$TYPE = SINGLE$REG)
        /* single register */
        THEN DO;
356  4  IF ((REG.$CODE AND R$ERROR) = R$ERROR) OR
        (REG.$REGISTER = SPECIAL$CASE)
        /* is this an illegal register */
        THEN ERROR$TYPE = 'R',
        /* generate object code */
        OBJECT.$CODE = OBJECT.$CODE OR (REG.$CODE AND REG$MASK);
        END;
    ELSE DO;
358  4
359  4
360  3

```

./M-80 COMPILER STRUCTURED MACRO ASSEMBLER - PASS 3

```

161 4      /* register pair */
      IF ((REG.RP$CODE AND RP$ERROR) = RP$ERROR) OR
        (REG.REGISTER = SPECIAL$CASE)
163 4          THEN ERROR$TYPE = 'R';
164 4      OBJECT.CODE = OBJECT.CODE OR (REG.RP$CODE AND REG$MASK);
      END;

165 3      END ENCODE$REG;

      /* initialize object code data structure */
366 2      OBJECT.LOCATION = LOCATION$CNT;
367 2      OBJECT.LENGTH = INSTRUCTION.LENGTH;
368 2      OBJECT.CODE$STATUS = BYTE$LNQ OR BYTE$TYPE;
369 2      OBJECT.DATA$STATUS = NULL;
370 2      OBJECT.CODE = INSTRUCTION.CODE;
      /* list object code in listing */
371 2      LISTING.OBJECT = ON;

      /* increment location$cnt by length of instruction */
372 2      LOCATION$CNT = LOCATION$CNT + INSTRUCTION.LENGTH;
373 2      ERROR$TYPE = NULL;
374 2      INST$TYPE = INSTRUCTION.INST$TYPE;
375 2      CALL GETSYM;

376 2      DO CASE INST$TYPE;

/*
*/      instruction type 0      instruction- RST n

      DO;
377 3          CALL EXPRESSION(.OBJECT.DATA$VAL, ABSOLUTE$SEG, BYTE$LNQ);
378 4          ERROR$TYPE = EXPRES.ERROR;
379 4          BYTE$TEMP = OBJECT.DATA$VAL;
380 4          OBJECT.CODE = OBJECT.CODE OR (SHL(BYTE$TEMP,3));
381 4          END;
382 4

/*      instruction type 1      format - opcode
*/

383 3      DO;
384 4          END;

/*      instruction type 2      format - opcode <single register in destination
                                position>
*/

385 3      DO; CALL ENCODE$REG(SINGLE$REG, DEST$REG, NULL);
386 4          END;
387 4

```

```

/*      instruction type 3      format - opcode <single register in source
*/
188      3      DO;
189      4      CALL ENCODE$REG(SINGLE$REG, SRC$REG, NULL);
190      4      END;

/*      instruction type 4      format - opcode <register pair>
*/
391      3      DO;
392      4      CALL ENCODE$REG(REG$PAIR, RP$1$REG, PSW$REG);
393      4      END;

/*      instruction type 5      format - opcode <register pair>
*/
394      3      DO;
395      4      CALL ENCODE$REG(REG$PAIR, RP$2$REG, PSW$REG);
396      4      IF (ERROR$TYPE = NULL) AND ((REG.REGISTER <> B$REG) OR
397      4      (REG.REGISTER <> D$REG))
398      4      THEN ERROR$TYPE = 'R';
399      4      END;

/*      instruction type 6      format - opcode <single register source pos.>,
*/
400      3      DO;
401      4      CALL ENCODE$REG(SINGLE$REG, DEST$REG, NULL);
402      4      CALL GETSYM;
403      4      IF (SYM$TYPE = COMMA$SYM)
404      4      THEN CALL GETSYM;
405      4      CALL ENCODE$REG(SINGLE$REG, SRC$REG, NULL);
406      4      END;

/*      instruction type 7      format - opcode <register pair>
*/
407      3      DO;
408      4      CALL ENCODE$REG(REG$PAIR, RP$1$REG, NULL);
409      4      END;

/*      instruction type 8      format - opcode <single register destination pos.>,
*/
410      3      DO;
411      4      CALL ENCODE$REG(SINGLE$REG, DEST$REG, NULL);
412      4      CALL GETSYM;

```

```

112  4      IF (SYM$TYPE = COMMA$SYM)
114  4          THEN CALL GETSYM;
115  4      CALL EXPRESSION(.OBJECT.DAT$VAL, ABSOLUTE$SEQ, BYTE$LNQ);
116  4      OBJECT.DAT$STATUS = EXPRES.SEG$TYPE OR BYTE$TYPE;
118  4      IF (ERROR$TYPE = NULL)
          THEN ERROR$TYPE = EXPRES.ERROR;
          END;

/*
*/
instruction type 9  format - opcode <byte absolute data>

DO;
119  3      CALL EXPRESSION(.OBJECT.DAT$VAL, ABSOLUTE$SEQ, BYTE$LNQ);
120  4      OBJECT.DAT$STATUS = EXPRES.SEG$TYPE OR BYTE$TYPE;
121  4      IF (ERROR$TYPE = NULL)
122  4          THEN ERROR$TYPE = EXPRES.ERROR;
124  4      END;

/*
*/
instruction type 10 format - opcode <word value>

DO;
125  3      CALL EXPRESSION(.OBJECT.DAT$VAL, NULL, WORD$LNQ);
126  4      OBJECT.DAT$STATUS = EXPRES.SEG$TYPE OR WORD$TYPE;
127  4      OBJECT.EXTRN$NO = EXPRES.EXTRN$NO;
128  4      IF (ERROR$TYPE = NULL)
129  4          THEN ERROR$TYPE = EXPRES.ERROR;
131  4      END;

/*
*/
instruction type 11 format - opcode <register pair except PSW>,
/*
*/
                                <word value>

DO;
132  3      CALL ENCODEREG(REG$PAIR, RP$1$REG, PSW$REG);
133  4      CALL GETSYM;
134  4      IF (SYM$TYPE = COMMA$SYM)
135  4          THEN CALL GETSYM;
137  4      CALL EXPRESSION(.OBJECT.DAT$VAL, NULL, WORD$LNQ);
138  4      OBJECT.DAT$STATUS = EXPRES.SEG$TYPE OR WORD$TYPE;
139  4      OBJECT.EXTRN$NO = EXPRES.EXTRN$NO;
140  4      IF (ERROR$TYPE = NULL)
142  4          THEN ERROR$TYPE = EXPRES.ERROR;
          END;

143  3      END; /* END CASE STATEMENT */

144  2      CALL CREATE$OBJECT;
145  2      RETURN ERROR$TYPE;

```

46 2 END INST\$PROC;

```

$EJECT
47  1  IDENT$PROC:
      PROCEDURE                                BYTE REENTRANT;

48  2  DECLARE ERROR$TYPE                      BYTE;

49  2  DECLARE STATUS                          BYTE;

50  2  DECLARE TEMP$IDENT
      TYPE
      STATUS
      VALUE
      DUMMY
      CHAR (SYM$LENGTH)
      STRUCTURE(
      BYTE,
      BYTE,
      WORD,
      BYTE,
      BYTE);

451 2  ERROR$TYPE = NULL;

452 2  /* this is a recursive routine, must save identifier name */
      CALL COPY$BUFFS(. TEMP$IDENT.CHAR, . IDENT.CHAR, SYM$LENGTH);

453 2  /* look up symbol */
454 2  STATUS = SYM$LOOKUP(. TEMP$IDENT);
      IF ((TEMP$IDENT.STATUS AND PUBLIC$DEF) = PUBLIC$DEF) AND
      ((TEMP$IDENT.STATUS AND PASS3$DEF) <> PASS3$DEF)
      /* call object code routine to handle public symbols */
      THEN CALL DCL$PUBLIC$SYM(. TEMP$IDENT);

456 2  CALL GET$SYM;

457 2  IF (SYM$TYPE = COLON$SYM)
      THEN DO;

459 3  IF (IDENT$FLAG <> ON)
      /* see if anything else on this line */
      THEN ERROR$TYPE = PASS3$PROC;
      /* only process the location tag on this line */
      ELSE IDENT$FLAG = OFF;

461 3  /* look up symbol again */
462 3  STATUS = SYM$LOOKUP(. TEMP$IDENT);
463 3  IF ((TEMP$IDENT.STATUS AND EXTRN$DEF) = EXTRN$DEF) OR
      IF ((TEMP$IDENT.STATUS AND PASS3$DEF) = PASS3$DEF)
      /* symbol already defined during pass3 */
      THEN ERROR$TYPE = 'M';
      ELSE DO;

465 3  /* set defined during pass 3 */
466 4  TEMP$IDENT.STATUS = TEMP$IDENT.STATUS OR PASS3$DEF;
467 4  STATUS = SYM$UPDATE(. TEMP$IDENT);

```


/M-80 COMPILER STRUCTURED MACRO ASSEMBLER - PASS 3

```

468 4      END;
469 3      ELSE

470 2      IF (SYMTYPE = EQUSSYM)
471 THEN DO;
472 3          IF ((TEMP$IDENT.STATUS AND EXTRN$DEF) = EXTRN$DEF) OR
              ((TEMP$IDENT.STATUS AND PASS3$DEF) = PASS3$DEF)
              /* already defined during pass 3 */
          THEN ERROR$TYPE = 'M';
          ELSE DO;
              /* set defined during pass 3 and get value */
              TEMP$IDENT.STATUS = TEMP$IDENT.STATUS OR PASS3$DEF;
              STATUS = SYM$UPDATE(.TEMP$IDENT);
              CALL GETSYM;
              CALL EXPRESSION (.OBJECT.LOCATION, ABSOLUTE$SEG, WORD$LNQ);
              /* set error to expression error */
              ERROR$TYPE = EXPRES.ERROR;
              OBJECT.LENGTH = NULL;
              OBJECT.CODE$STATUS = NULL;
              OBJECT.DATA$STATUS = NULL;
              /* list value in location field in listing
                 no object generated for EGU statements */
              LISTING.OBJECT = ON;
          END;
          ELSE
              END;

483 4      IF (SYMTYPE = SET$SYM)
484 4      THEN DO;
485 3          ELSE

486 2          IF (TEMP$IDENT.TYPE <> SET$SYM)
487 3          THEN DO;
488 3              /* was previously defined other than SET identifier */
              THEN ERROR$TYPE = 'M';
              ELSE DO;
                  /* get new value for SET identifier,
                     SET identifier can be changer during assembly */
                  CALL GETSYM;
                  CALL EXPRESSION(.TEMP$IDENT.VALUE, ABSOLUTE$SEG, WORD$LNQ);
                  /* update symbol table */
                  TEMP$IDENT.STATUS = ABSOLUTE$SEG;
                  STATUS = SYM$UPDATE(.TEMP$IDENT);
                  ERROR$TYPE = EXPRES.ERROR;
                  OBJECT.LOCATION = TEMP$IDENT.VALUE;
                  OBJECT.CODE$STATUS = NULL;
                  OBJECT.DATA$STATUS = NULL;
                  OBJECT.LENGTH = NULL;
                  /* print value in location field of listing,

```

```

/M-80 COMPILER      STRUCTURED MACRO ASSEMBLER - PASS 3

      no object code is generated for EGU statements */
      LISTING.OBJECT = ON;
      END;

      ELSE DO;
      END;
      /* who knows what is here see what comes after it */
      ERROR$TYPE = PASS3$PROC;
      IF (ERROR$TYPE = NULL)
      /* if no error set error to undefined */
      THEN ERROR$TYPE = 'U';
      END;

      RETURN ERROR$TYPE;

      END IDENT$PROC;

```

```

$EJECT
310 1  DATA$PROC:
      PROCEDURE
      BYTE,
311 2  DECLARE ERROR$TYPE
      BYTE,
312 2  DECLARE DATA$STORAGE$TYPE
      BYTE,
      /* This routine evaluated DS, DB and DW statements */
313 2  ERROR$TYPE = NULL,
314 2  LISTING.OBJECT = ON,
315 2  DATA$STORAGE$TYPE = DATA$STORAGE.INST,
316 2  CALL GETSYM,
317 2  DO CASE DATA$STORAGE$TYPE,
      /* DS SYMBOL */
318 3  DO,
      /* read number of bytes to leave room for from operand field */
319 4  CALL EXPRESSION (.OBJECT.LENGTH, ABSOLUTE$SEG, WORD$LNQ),
      /* call routine to leave space specified */
320 4  CALL OBJECT$SPACE,
      /* set location counter for object code */
321 4  OBJECT.LOCATION = LOCATION$CNT,
322 4  LOCATION$CNT = LOCATION$CNT + OBJECT.LENGTH,
      /* set error to expression error */
323 4  ERROR$TYPE = EXPRES.ERROR,
324 4  OBJECT.LENGTH = NULL,
325 4  OBJECT.CODE$STATUS = NULL,
326 4  END,
      /* DB SYMBOL
      parse line buffer to determine the value of each byte and the number
      of bytes defined by the DB instruction. */
327 3  DO,
328 4  ASCII$STATUS.ERROR = NULL,
329 4  DO WHILE (SYM$TYPE <> COMMENT$SYM) AND (SYM$TYPE <> EOL$SYM),
330 5  OBJECT.LOCATION = LOCATION$CNT,
331 5  IF (SYM$TYPE = QUOTE$SYM) AND (G$ASCII$LNQ > BYTE$LNQ)
      THEN DO,
      /* string surrounded by quotes set object data
      status to byte and absolute data */
332 6  OBJECT.DATA$STATUS = BYTE$TYPE OR ABSOLUTE$SEG,
      /* if data in a DB statements used the data and
      code fields of the data structure OBJECT to
      store the data. This makes four byte available */

```

```

434 6 DO WHILE (NOT(0=ASCII(. OBJECT.CODE, 4, BYTE$LNQ))) AND
435 7 (ASCII$STATUS.ERROR = NULL);
436 7 OBJECT.LENGTH = 4;
437 7 OBJECT.CODE$STATUS = WORD$LNQ OR BYTE$TYPE;
438 7 /* print line */
439 7 CALL PRINT$LINE(ASCII$STATUS.ERROR);
440 7 /* create object code for data */
441 7 CALL CREATE$OBJECT;
442 7 /* after the first line of data is printed
443 7 is printed */
444 7 LISTING.LINE = OFF;
445 7 LOCATION$CNT = LOCATION$CNT + 4;
446 7 OBJECT.LOCATION = LOCATION$CNT;
447 7 ASCII$STATUS.ACT$LNQ = 0;
448 7 END /* WHILE */;
449 7 IF (ASCII$STATUS.ACT$LNQ <> 0)
450 7 THEN DO;
451 7 OBJECT.LENGTH = ASCII$STATUS.ACT$LNQ;
452 7 /* set status for object code creation */
453 7 IF (OBJECT.LENGTH > 1)
454 7 THEN OBJECT.CODE$STATUS = BYTE$TYPE OR WORD$LNQ;
455 7 ELSE OBJECT.CODE$STATUS = BYTE$TYPE OR BYTE$LNQ;
456 7 CALL PRINT$LINE(ASCII$STATUS.ERROR);
457 7 CALL CREATE$OBJECT;
458 7 LISTING.LINE = OFF;
459 7 LOCATION$CNT = LOCATION$CNT +
460 7 ASCII$STATUS.ACT$LNQ;
461 7 END;
462 7 ERROR$TYPE = ASCII$STATUS.ERROR;
463 7 CALL GET$SYM;
464 7 END;
465 7 ELSE DO;
466 7 /* get byte value */
467 7 CALL EXPRESSION(. OBJECT.DAT$VAL, ABSOLUTE$SEQ, BYTE$LNQ);
468 7 OBJECT.LENGTH = BYTE$LNQ;
469 7 OBJECT.DAT$STATUS = ABSOLUTE$SEQ OR BYTE$TYPE;
470 7 OBJECT.CODE$STATUS = NULL;
471 7 CALL PRINT$LINE(EXPRES.ERROR);
472 7 CALL CREATE$OBJECT;
473 7 LISTING.LINE = OFF;
474 7 LOCATION$CNT = LOCATION$CNT + 1;
475 7 ERROR$TYPE = EXPRES.ERROR;
476 7 END;
477 7 IF (SYM$TYPE = COMMA$SYM)
478 7 THEN CALL GET$SYM;
479 7 END /* WHILE */;
480 7
571 5

```

```

72  4      END;

/* DW SYMBOL */
/* parse the line buffer and get the word value of each element
and determine the number of words to be stored. Elements are
separated by commas */
73  3      DO;
74  4          ASCII$STATUS.ERROR = NULL;
75  4          DO WHILE (SYM$TYPE <> COMMENT$SYM) AND (SYM$TYPE <> EOL$SYM);
76  5              OBJECT.LOCATION = LOCATION$CNT;
/* data is ascii string separated by quotes
word value are stored in memory low order byte followed
by the high order byte */
              IF (SYM$TYPE = QUOTE$SYM) AND (G$ASCII$LN > WORD$LN)
              THEN DO;
/* as with DB statement only print source line once
after that only print object code fields of
listing */
              OBJECT.DATA$STATUS = ABSOLUTE$SEQ OR WORD$TYPE;
              DO WHILE (NOT(G$ASCII(.OBJECT.CODE, 4, WORD$LN))) AND
              (ASCII$STATUS.ERROR = NULL);
                  OBJECT.LENGTH = 4;
                  OBJECT.CODE$STATUS = WORD$LN OR WORD$TYPE;
                  CALL PRINT$LINE(ASCII$STATUS.ERROR);
                  CALL CREATE$OBJECT;
                  LISTING.LINE = OFF;
                  LOCATION$CNT = LOCATION$CNT + 4;
                  OBJECT.LOCATION = LOCATION$CNT;
                  ASCII$STATUS.ACT$LN = 0;
              END /* WHILE */;
/* clean up when last part of string is not 4 bytes
in length */
              IF (ASCII$STATUS.ACT$LN <> 0)
              THEN DO;
                  OBJECT.LENGTH = ASCII$STATUS.ACT$LN;
                  OBJECT.CODE$STATUS = WORD$TYPE OR WORD$LN;
                  CALL PRINT$LINE(ASCII$STATUS.ERROR);
                  CALL CREATE$OBJECT;
                  LISTING.LINE = OFF;
                  LOCATION$CNT = LOCATION$CNT +
                      ASCII$STATUS.ACT$LN;
              END;
                  ERROR$TYPE = ASCII$STATUS.ERROR;
                  CALL GET$SYM;
              END;
            ELSE DO;
/* elements are other than ascii string longer than

```

```

03      6          two characters, ie. numeric or identifiers */
04      6      CALL EXPRESSION(,OBJECT.DATASVAL, NULL, WORD$LNQ);
05      6      OBJECT.LENGTH = WORD$LNQ;
06      6      OBJECT.DATASSTATUS = WORD$TYPE OR EXPRES.SEQ$TYPE;
07      6      OBJECT.EXTRN$ND = EXPRES.EXTRN$ND;
08      6      OBJECT.CODE$STATUS = NULL;
09      6      CALL PRINT$LINE(EXPRES.ERROR);
10      6      CALL CREATE$OBJECT;
11      6      LISTING.LINE = OFF;
12      6      LOCATION$CNT = LOCATION$CNT + 2;
13      6      ERROR$TYPE = EXPRES.ERROR;
14      5      END;
        IF (SYM$TYPE = COMMA$SYM)
        THEN CALL GETSYM;
        END /* WHILE */;
616      5      END;
617      4      END;
618      3      END /* CASE */;
619      2      RETURN ERROR$TYPE;
620      2      END DATA$PROC;

```

```

$EJECT
121 1  EXTRN$PROC:
      PROCEDURE
      BYTE;

122 2  DECLARE STATUS
123 2  DECLARE ERROR$TYPE
      BYTE;
      BYTE;

      /* get identifiers in external statement and see if they were previously
        defined other than external. */
124 2  ERROR$TYPE = NULL;
125 2  CALL GET$SYM;
126 2  DO WHILE (SYM$TYPE <> EOL$SYM) AND (SYM$TYPE <> COMMENT$SYM);
127 3  IF (SYM$TYPE = IDENT$SYM)
      THEN DO;
129 4  STATUS = SYM$LOOKUP(SYM$PTR);
130 4  IF ((IDENT.STATUS AND SEQ$DEF) <> EXTERNAL$SEQ)
      /* previously defined other than external */
      THEN ERROR$TYPE = 'M';
      /* tell object code formatter about the external */
      ELSE CALL DCL$EXTRN$SYM(SYM$PTR);
      END;
      /* symbol other than identifier */
134 3  ELSE ERROR$TYPE = 'I';
135 3  CALL GET$SYM;
136 3  IF (SYM$TYPE = COMMA$SYM)
      THEN CALL GET$SYM;
138 3  END;
139 2  RETURN ERROR$TYPE;
140 2  END EXTRN$PROC;

```

```

$EJECT
1 1 PUBLIC$PROC:
PROCEDURE BYTE;

2 2 DECLARE STATUS BYTE;
3 2 DECLARE ERROR$TYPE BYTE;

4 2 /* define identifiers in public statement as public */
5 2 ERROR$TYPE = NULL;
6 2 CALL GET$SYM;
7 2 DO WHILE (SYM$TYPE <> EOL$SYM) AND (SYM$TYPE <> COMMENT$SYM);
8 3 IF (SYM$TYPE = IDENT$SYM)
9 4 THEN DO;
10 4 STATUS = SYM$LOOKUP(SYM$PTR);
11 4 IF ((IDENT.STATUS AND PASS2$DEF) <> PASS2$DEF)
12 5 THEN ERROR$TYPE = 'U';
13 4 END;
14 4 ELSE ERROR$TYPE = 'I';
15 3 CALL GET$SYM;
16 3 IF (SYM$TYPE = COMMA$SYM)
17 3 THEN CALL GET$SYM;
18 3 END;
19 2 RETURN ERROR$TYPE;
20 2 END PUBLIC$PROC;

```



```

$EJECT
60 1  PASS3$PROC:
    PROCEDURE
        BYTE REENTRANT;

        /* this is the control loop that calls the appropriate routine or
           takes the required for a given symbol */

        LISTING.OBJECT = OFF;
        LISTING.LINE = ON;
        OBJECT.LENGTH = NULL;
        OBJECT.CODE$STATUS = NULL;
        OBJECT.DATA$STATUS = NULL;

        CALL GETSYM;

        IF (SYM$TYPE = SKIP$LINE$SYM)
        THEN DO;

        669 3  IF ((SKIP.LINE$TYPE AND NOT$MACRO$LINE) <> STATEMENT$LINE)
        671 4  THEN DO;
            IF ((SKIP.LINE$TYPE AND NOT$MACRO$LINE) = TAG$LINE)
            /* location tag is on a line with a structured
               statement. Get the tag and only the tag. this
               is what ident$flag = on means */
            THEN IDENT$FLAG = ON;
            CALL G$CHAR$INIT(4);
            STATUS = PASS3$PROC;
            END;
            ELSE STATUS = NULL;
            END;
        ELSE
        673 4  IF SYM$TYPE = IDENT$SYM
        674 4  THEN STATUS = IDENT$PROC;
        675 4  ELSE
        676 3  IF SYM$TYPE = MEMORY$SYM
        677 3  THEN STATUS = MEM$PROC;
            ELSE
        678 2  IF SYM$TYPE = IDENT$SYM
            THEN STATUS = IDENT$PROC;
            ELSE
        680 2  IF SYM$TYPE = MEMORY$SYM
            THEN STATUS = MEM$PROC;
            ELSE
        682 2  IF SYM$TYPE = CONTROL$SYM
            THEN STATUS = CNTL$PROC(3);
            ELSE
        684 2  IF SYM$TYPE = DATA$STORAGE$SYM
            THEN STATUS = DATA$PROC;
            ELSE
        686 2  IF SYM$TYPE = INSTRUCTION$SYM
            THEN STATUS = INST$PROC;
            ELSE
        688 2  IF SYM$TYPE = EXTRN$SYM

```

/M-80 COMPILER STRUCTURED MACRO ASSEMBLER - PASS 3

```
      THEN STATUS = EXTRN$PROC;  
    ELSE  
      IF SYM$TYPE = PUBLIC$SYM  
      THEN STATUS = PUBLIC$PROC;  
    ELSE  
      IF (SYM$TYPE = END$SYM) OR (SYM$TYPE = COMMENT$SYM) OR  
      (SYM$TYPE = EOL$SYM)  
      THEN STATUS = NULL;  
    ELSE STATUS = 'U';  
  
    RETURN STATUS;  
  
  END PASS3$PROC;
```

```

$EJECT
397 1  MAIN$PASS3:
    PROCEDURE                                PUBLIC;

598 2  DECLARE STATUS                        BYTE;
599 2  DECLARE ERROR$TYPE                  BYTE;

700 2  /* initialize */
701 2  CALL FILE$INIT$PASS3;
702 2  CALL SEGMENT$INIT;
703 2  CALL PRINT$INIT;
704 2  CALL GET$LINE$INIT;
705 2  IDENT$FLAG = OFF;
706 2  SYM$TYPE = NULL;
707 2  /* get first line */
708 2  STATUS = GET$LINE;
709 2  DO WHILE ((SYM$TYPE <> EOF$SYM) AND (SYM$TYPE <> END$SYM)) AND
    (STATUS = TRUE);
710 2  ERROR$TYPE = PASS3$PROC;
711 2  IF (LISTING.LINE = ON)
712 2  THEN CALL PRINT$LINE(ERROR$TYPE);
713 2  STATUS = GET$LINE;
714 2  END;
715 2  /* clean up */
716 2  CALL SEGMENT$FINISH;
717 2  IF (SYM$TYPE <> END$SYM)
718 2  THEN CALL PRINT$LINE('F');

716 2  END MAIN$PASS3;
717 1  END PASS$3$COMMAND;

```

MODULE INFORMATION:

```

CODE AREA SIZE      = 0A24H    2596D
VARIABLE AREA SIZE = 0012H    18D
MAXIMUM STACK SIZE = 0019H    25D
1297 LINES READ
0 PROGRAM ERROR(S)

```

END OF PL/M-80 COMPILATION

./M-80 COMPILER SMA PRINT FORMATTER

10-MAR-82 PAGE 1

```

)IS-II PL/M-80 V3.1 COMPILATION OF MODULE PRINTFILE
) OBJECT MODULE REQUESTED
)MPILER INVOKED BY: PLM80 PRINT. PLM DATE(10-MAR-82) NOOBJECT PRINT(:F1:PRINT.LST) PAGELENGTH(50)

```

```

1      $TITLE('SMA PRINT FORMATTER ')
      PRINT$FILE;
      DO;

```

```

2      1      DECLARE REV820205          BYTE AT(0);
      /* WRITTEN BY JOSEPH R. GARAPPOLO    07 JULY 1981

```

This routine is responsible for formatting and producing the listing of the source code and copy this listing to the list device. This list device may be a disk file for another device such as a lineprinter

```

      INCLUDE FILES - INC      ELD
                        SYSTEM.ELD
                        FILEIO.EPD
                        OBJECT.EPD
                        CNTL      EPD
                        LOCNT.EPD
                        GETSYM.EPD

```

```

      */
      $NOLIST

```


./M-80 COMPILER SMA PRINT FORMATTER

108	1	DECLARE PRINT\$BUFFER (132)	BYTE;
109	1	DECLARE STATUS	WORD;
110	1	DECLARE LINE\$NUMBER	WORD;
111	1	DECLARE PAGE\$NUMBER	WORD;
112	1	DECLARE PAGE\$INDEX	BYTE;
113	1	DECLARE I	BYTE;
114	1	DECLARE TEMP\$VAL	WORD;

```

15 1 $EJECT
    PAGE$ADVANCE;
    PROCEDURE;

/* This routine writes a formfeed command to the listing device and
   writes the header on the top of the page
*/

116 2 DECLARE FORM$FEED          LITERALLY 'OCH';

/* send form feed */
CALL WRITE (AFT$LIST, (FORM$FEED), 1, STATUS);
/* initialize page index */
PAGE$INDEX = 6;
CALL CLEAR$BUFF (.PRINT$BUFFER, 132);
/* send header which may include title */
CALL COPY$BUFFS(.PRINT$BUFFER(2), ('SMABO STRUCTURED MACRO ASSEMBLER'), 33);
CALL COPY$BUFFS(.PRINT$BUFFER(36), TITLE.BUFF, TITLE.LNG);
CALL COPY$BUFFS(.PRINT$BUFFER(70), (' PAGE '), 6);
CALL NUMOUT (PAGE$NUMBER, 10, .PRINT$BUFFER(76), 4, ' ');
CALL COPY$BUFFS(.PRINT$BUFFER(80), (CR,LF), 2);
CALL WRITE (AFT$LIST, .PRINT$BUFFER, 82, STATUS);
CALL WRITE (AFT$LIST, (CR,LF,LF,LF, LOC CODE DATA LINE',
    SOURCE CODE ',CR,LF,LF).54, STATUS);
PAGE$NUMBER = PAGE$NUMBER + 1;

325 2
326 2 END PAGE$ADVANCE;

```

```

$EJECT
327 1 PRINT$FINISH:
PROCEDURE PUBLIC;
/* This routine is called to finish listing
*/
/* return if no listing */
328 2 IF (PRINT$FLAG = OFF)
THEN RETURN;
/* report error if any structured statements were not closed */
330 2 IF (IF$COUNT <> 0) OR (WHILE$COUNT <> 0) OR
(FOR$COUNT <> 0) OR (CASE$COUNT <> 0)
THEN DO;
332 3 IF ((PAGE$INDEX + 7) > PAGE$LENGTH)
THEN CALL PAGE$ADVANCE;
334 3 CALL WRITE(AFT$LIST, (CR, LF,
STATEMENT ERRORS ', CR, LF, LF), 26, STATUS);
335 3 PAGE$INDEX = PAGE$INDEX + 3;
336 3 END;
337 2 IF (IF$COUNT <> 0)
THEN DO;
339 3 CALL COPY$BUFFS(.PRINT$BUFFER,
(' XXX ENDIF(S) MISSING', CR, LF), 26);
340 3 CALL NUMOUT(IF$COUNT, 16, .PRINT$BUFFER(3), 3, ' ');
341 3 CALL WRITE(AFT$LIST, .PRINT$BUFFER, 26, STATUS);
342 3 PAGE$INDEX = PAGE$INDEX + 1;
343 3 END;
344 2 IF (WHILE$COUNT <> 0)
THEN DO;
346 3 CALL COPY$BUFFS(.PRINT$BUFFER,
(' XXX ENDWHILE(S) MISSING', CR, LF), 29);
347 3 CALL NUMOUT(WHILE$COUNT, 16, .PRINT$BUFFER(3), 3, ' ');
348 3 CALL WRITE(AFT$LIST, .PRINT$BUFFER, 29, STATUS);
349 3 PAGE$INDEX = PAGE$INDEX + 1;
350 3 END;
351 2 IF (FOR$COUNT <> 0)
THEN DO;
353 3 CALL COPY$BUFFS(.PRINT$BUFFER,
(' XXX ENDFOR(S) MISSING', CR, LF), 27);
354 3 CALL NUMOUT(FOR$COUNT, 16, .PRINT$BUFFER(3), 3, ' ');
355 3 CALL WRITE(AFT$LIST, .PRINT$BUFFER, 27, STATUS);
356 3 PAGE$INDEX = PAGE$INDEX + 1;
357 3 END;

```



```

358 2 IF (CASE$COUNT <> 0)
359 3 THEN DO;
360 3 CALL COPY$BUFFS(.PRINT$BUFFER,
361 3 (' XXX ENDCASE(S) MISSING', CR, LF), 28);
362 3 CALL NUMOUT(CASE$COUNT, 16, .PRINT$BUFFER(3), 3, ' ');
363 3 CALL WRITE(AFT$LIST, .PRINT$BUFFER, 28, STATUS);
364 3 PAGE$INDEX = PAGE$INDEX + 1;
365 3 END;

/* report the values of the three location counters */
366 2 IF (CODE$LOC$CNT <> 0) OR (DATA$LOC$CNT <> 0) OR
367 3 (ASEG$LOC$CNT <> 0)
368 3 THEN DO;
369 3 IF ((PAGE$INDEX + 6) > PAGE$LENGTH)
370 3 THEN CALL PAGE$ADVANCE;
371 3 CALL WRITE(AFT$LIST,
372 3 (CR, LF, ' MODULE INFORMATION', CR, LF, LF),
373 3 26, STATUS);
374 3 PAGE$INDEX = PAGE$INDEX + 3;
375 3 END;
376 2 IF (CODE$LOC$CNT <> 0)
377 3 THEN DO;
378 3 CALL COPY$BUFFS(.PRINT$BUFFER,
379 3 (' CODE AREA SIZE = XXXX XXXXD', CR, LF), 41);
380 3 CALL NUMOUT(CODE$LOC$CNT, 16, .PRINT$BUFFER(25), 4, ' ');
381 3 CALL NUMOUT(CODE$LOC$CNT, 10, .PRINT$BUFFER(33), 5, ' ');
382 3 CALL WRITE(AFT$LIST, .PRINT$BUFFER, 41, STATUS);
383 3 PAGE$INDEX = PAGE$INDEX + 1;
384 3 END;
385 2 IF (DATA$LOC$CNT <> 0)
386 3 THEN DO;
387 3 CALL COPY$BUFFS(.PRINT$BUFFER,
388 3 (' DATA AREA SIZE = XXXX XXXXD', CR, LF), 41);
389 3 CALL NUMOUT(DATA$LOC$CNT, 16, .PRINT$BUFFER(25), 4, ' ');
390 3 CALL NUMOUT(DATA$LOC$CNT, 10, .PRINT$BUFFER(33), 5, ' ');
391 3 CALL WRITE(AFT$LIST, .PRINT$BUFFER, 41, STATUS);
392 3 PAGE$INDEX = PAGE$INDEX + 1;
393 3 END;
394 2 IF (ASEG$LOC$CNT <> 0)
395 3 THEN DO;
396 3 CALL COPY$BUFFS(.PRINT$BUFFER,
397 3 (' ABSOLUTE AREA SIZE = XXXX XXXXD', CR, LF), 41);
398 3 CALL NUMOUT(ASEG$LOC$CNT, 16, .PRINT$BUFFER(25), 4, ' ');
399 3 CALL NUMOUT(ASEG$LOC$CNT, 10, .PRINT$BUFFER(33), 5, ' ');
400 3 CALL WRITE(AFT$LIST, .PRINT$BUFFER, 41, STATUS);
401 3 PAGE$INDEX = PAGE$INDEX + 1;

```

```

395 3      END;

/* print number of errors */
396 2      CALL COPY$BUFFS( PRINT$BUFFER, ( '      PROGRAM ERROR(S)', CR, LF, LF), 27);
397 2      IF (NUMBER$OF$ERRORS = 0)
399 2          THEN PRINT$BUFFER(6) = '0';
400 2          ELSE CALL NUMOUT(NUMBER$OF$ERRORS, 10, .PRINT$BUFFER(3), 4, ' ');
401 2          CALL WRITE(AFT$LST, .PRINT$BUFFER, 27, STATUS);
      CALL WRITE(AFT$LST, ( 'END OF SMABO', CR, LF), 14, STATUS);

402 2      END PRINT$FINISH;
  
```

./M-80 COMPILER SMA PRINT FORMATTER

```

$EJECT
103 1  PRINT$INIT:
      PROCEDURE
      PUBLIC;

      /* initialize listing */

104 2  IF (PRINT$FLAG = ON) AND (LIST$FLAG = ON)
      THEN DO,
106 3      LINE$NUMBER = 1;
107 3      PAGE$NUMBER = 1;
108 3      NUMBER$OF$ERRORS = 0;
109 3      CALL PAGE$ADVANCE;
      /* these messages only appear on the first page */
110 3      CALL WRITE(AFT$ST, (CR, LF, 'ISIS-II SMA-80 ASSEMBLER', CR, LF), 28, .STATUS);
111 3      CALL COPY$BUFFS(.RUN$STRING, ('SMA-80 INVOKED BY: SMABO'), 27);
112 3      I = 0;
113 3      DO WHILE (I < LINE$SIZE) AND (RUN$STRING(I) <> LF);
114 4          I = I + 1;
115 4      END;
116 3      CALL WRITE(AFT$ST, RUN$STRING, I+1, .STATUS);
117 3      CALL WRITE(AFT$ST, (CR, LF, LF), 4, .STATUS);
118 3      PAGE$INDEX = PAGE$INDEX + 7;
119 3      END;

420 2  END PRINT$INIT;

```

```

$EJECT
121 1 PRINT$LINE: PUBLIC;
PROCEDURE (ERROR$TYPE)
/* this is the main listing routine */

122 2 DECLARE ERROR$TYPE BYTE;
123 2 DECLARE CHAR BYTE;
124 2 DECLARE LINE$END BYTE;
125 2 DECLARE DATA$STATUS BYTE;
126 2 DECLARE LINE$INDEX BYTE;
127 2 DECLARE SKIP$LINE$TYPE BYTE;

428 2 IF (LINE$BUFF(0) = SKIP$STATEMENT)
THEN DO;
430 3 IF (LINE$BUFF(1) <> STATEMENT$LINE) AND (LINE$BUFF(1) <> TAG$LINE)
/* this line contains code generated by macro expansion
or code generated by structured statements. Lines of
this type are printed only if CODE or EXPMACRO controls
were used */
THEN SKIP$LINE$TYPE = LINE$BUFF(1);
ELSE DO;
432 3 /* this is a structured statement line which
is always listed unless no list is specified */
IF (LINE$BUFF(2) <> NULL) OR
(LINE$BUFF(1) <> TAG$LINE)
THEN ERROR$TYPE = LINE$BUFF(2);
SKIP$LINE$TYPE = NULL;
END;
/* index past 5 byte intermediate code */
LINE$INDEX = 5;
END;
437 3
438 3 /* no intermediate code set index to 0 */
439 2 SKIP$LINE$TYPE = NULL,
LINE$INDEX = 0;
ELSE DO;
440 3
441 3
442 3 END;

/* increment error count if there is an error */
443 2 IF (ERROR$TYPE <> NULL)
THEN NUMBER$OF$ERRORS = NUMBER$OF$ERRORS + 1;

445 2 IF ((PRINT$FLAG = ON) AND ((LIST$FLAG = ON) OR (LIST$FLAG$2 = ON)) AND
(SKIP$LINE$TYPE <> STATEMENT$CODE)) OR
((PRINT$FLAG = ON) AND (ERROR$TYPE <> NULL)) OR
((PRINT$FLAG = ON) AND ((LIST$FLAG = ON) OR (LIST$FLAG$2 = ON)) AND

```

```

447 3      (STATEMENT$CODE$FLAG = ON))
      THEN DO;
      IF (LIST$FLAG = OFF)
      THEN LIST$FLAG$2 = OFF;

449 3      /* eject control was just encountered */
      IF (PAGE$FLAG = ON)
      THEN DO;
451 4          CALL PAGE$ADVANCE;
452 4          PAGE$FLAG = OFF;
453 4      END;

454 3      CALL CLEAR$BUFF (. PRINT$BUFFER, 32);
455 3      IF ERROR$TYPE = NULL
      THEN ERROR$TYPE = ' ';
457 3      PRINT$BUFFER(1) = ERROR$TYPE;

458 3      IF (LISTING.OBJECT = ON)
      THEN DO;

460 4          /* fill in object code fields of listing */
461 4          CALL COPY$BUFFS (.PRINT$BUFFER(32), (CR,LF), 2);
462 4          LINE$END = 34;
463 4          CALL NUMOUT(OBJECT.LOCATION, 16, PRINT$BUFFER(5), 4, 'O');
464 4          CODE$LENGTH = (OBJECT.CODE$STATUS AND DATA$FLD$LNG);
465 4          DATA$LENGTH = OBJECT.LENGTH - CODE$LENGTH;
      IF (DATA$LENGTH <> 0)
      THEN DO;
467 5              DATA$STATUS = OBJECT.DATA$STATUS AND
468 5                  SEG$DEF;
      IF (DATA$STATUS = ABSOLUTE$SEG)
      THEN;
      ELSE
470 5          IF (DATA$STATUS = CODE$SEG)
      THEN PRINT$BUFFER(22) = 'C';
      ELSE
472 5          IF (DATA$STATUS = DATA$SEG)
      THEN PRINT$BUFFER(22) = 'D';
      ELSE
474 5          IF (DATA$STATUS = EXTERNAL$SEG)
      THEN PRINT$BUFFER(22) = 'E';
      END;

477 4      IF (CODE$LENGTH <> 0)
      THEN DO I = 1 TO CODE$LENGTH,
479 5          TEMP$VAL = CODE(I-1);
480 5          CALL NUMOUT(TEMP$VAL, 16,
      . PRINT$BUFFER(10+(2*(I-1))), 2, 'O');
481 5      END;

```

./M-80 COMPILER SMA PRINT FORMATTER

```

482 4 IF (DATA$LENGTH <> 0)
483 5 THEN DO I = 1 TO DATA$LENGTH;
484 5 TEMP$VAL = DATA$VAL(I-1);
485 5 CALL NUMOUT (TEMP$VAL,16,
      .PRINT$BUFFER (15+(2*(I-1))),2, 'O');
486 5 END;
487 4 END;
488 3 IF (LISTING.LINE = ON)
      THEN DO;
      /* list source code line */
      CALL COPY$BUFFS(.PRINT$BUFFER(32),
        .LINE$BUFF(LINE$INDEX), LINE$SIZE);
      /* if line is code generated by structured
      statement flag line with '-' in col. 30 */
      IF (SKIP$LINE$TYPE = STATEMENT$CODE)
      THEN PRINT$BUFFER(30) = '-';
      ELSE
      /* if line is expanded macro flag line
      with '+' in col. 30 */
      IF (SKIP$LINE$TYPE = MACRO$LINE)
      THEN PRINT$BUFFER(30) = '+';
      CHAR = NULL;
      LINE$END = 32;
      DO WHILE (CHAR <> LF) AND (CHAR <> OFFH);
        CHAR = PRINT$BUFFER(LINE$END);
        LINE$END = LINE$END + 1;
      END;
491 4 END;
493 4
496 4 CHAR = NULL;
497 4 LINE$END = 32;
498 5 DO WHILE (CHAR <> LF) AND (CHAR <> OFFH);
499 5 CHAR = PRINT$BUFFER(LINE$END);
500 5 LINE$END = LINE$END + 1;
501 4 END;
502 3 /* increment line count and write line */
      IF (LISTING.OBJECT = ON) OR (LISTING.LINE = ON)
      THEN DO;
        CALL NUMOUT(LINE$NUMBER, 10, .PRINT$BUFFER(25), 4, ' ');
        LINE$NUMBER = LINE$NUMBER + 1;
        CALL WRITE(AFT$LIST, .PRINT$BUFFER, LINE$END, STATUS);
        PAGE$INDEX = PAGE$INDEX + 1;
        /* is page full */
        IF (PAGE$INDEX > PAGE$LENGTH)
        THEN CALL PAGE$ADVANCE;
508 4 END;
510 4 END;
511 3 ELSE IF (PRINT$FLAG = ON) AND (LISTING.LINE = ON)
512 2 THEN LINE$NUMBER = LINE$NUMBER + 1;

```

END PRINT\$LINE;

515 1 END PRINT\$FILE;

MODULE INFORMATION:

CODE AREA SIZE	= 0BF6H	2294D
VARIABLE AREA SIZE	= 009AH	154D
MAXIMUM STACK SIZE	= 000AH	10D
905 LINES READ		
0 PROGRAM ERROR(S)		

END OF PL/M-80 COMPILATION

./M-80 COMPILER SMA OBJECT CODE FORMATTER

IIS-II PL/M-80 V3.1 COMPILATION OF MODULE OBJECTCODE
) OBJECT MODULE REQUESTED
 MPILER INVOKED BY: PLM80 OBJECT.PLM DATE(10-MAR-82) NOOBJECT PRINT(: F1: OBJECT. LST) PAGELENGTH(50)

1	\$TITLE('SMA OBJECT CODE FORMATTER')	
	OBJECT\$CODE:	
	DO;	
2	1	DECLARE REV810912
		BYTE AT(O);
		07 JULY 1981
		/* WRITTEN BY JOSEPH R. GARAPPOLO
		INCLUDE FILES - INC .ELD
		SYSTEM.ELD
		FILEID.EPD
		*/
		\$NOLIST

/M-80 COMPILER SMA OBJECT CODE FORMATTER

		\$EJECT
11	1	NUMOUT:
12	2	PROCEDURE (VALUE, BASE, BUFF\$PTR, WIDTH, PAD) EXTERNAL;
13	2	DECLARE VALUE
14	2	DECLARE BASE
15	2	DECLARE BUFF\$PTR
16	2	DECLARE WIDTH
17	2	DECLARE PAD
		END NUMOUT;
218	1	DECLARE AFT\$OBJ
219	1	DECLARE OBJECT
		LOCATION
		LENGTH
		CODE\$STATUS
		DATA\$STATUS
		CODE
		DATA\$VAL
		EXTRN\$NO
		STRUCTURE (
		WORD,
		WORD,
		BYTE,
		BYTE,
		WORD,
		BYTE) PUBLIC;
220	1	STRUCTURE (
		BYTE,
		WORD) PUBLIC;
221	1	WORD;
		DECLARE STATUS

```

$EJECT
22 1  NEW$SEGMENT:
    PROCEDURE
        PUBLIC;
23 2  CALL WRITE(AFT$OBJ, ('NEW SEGMENT STARTED',CR,LF), 21, STATUS);
24 2  END NEW$SEGMENT;
25 1  CREATE$OBJECT:
    PROCEDURE
        PUBLIC;
226 2  DECLARE WORD$TEMP
227 2  DECLARE OBJECT$BUFF (132)
        WORD,
        BYTE DATA(
'LOC= XXXX LNG= XXXX C$STAT= XX D$STAT= XX CODE= XXXX DATA= XXXX EXTRN= XX',
CR,LF);
228 2  CALL NUMOUT(OBJECT.LOCATION, 16, .OBJECT$BUFF(5), 4, '0');
229 2  CALL NUMOUT(OBJECT.LENGTH, 16, .OBJECT$BUFF(15), 4, '0');
230 2  WORD$TEMP = OBJECT.CODE$STATUS;
231 2  CALL NUMOUT(WORD$TEMP, 16, .OBJECT$BUFF(28), 2, '0');
232 2  WORD$TEMP = OBJECT.DATA$STATUS;
233 2  CALL NUMOUT(WORD$TEMP, 16, .OBJECT$BUFF(39), 2, '0');
234 2  CALL NUMOUT(OBJECT.CODE, 16, .OBJECT$BUFF(49), 4, '0');
235 2  CALL NUMOUT(OBJECT.DATA$VAL, 16, .OBJECT$BUFF(60), 4, '0');
236 2  WORD$TEMP = OBJECT.EXTRN$ND;
237 2  CALL NUMOUT(WORD$TEMP, 16, .OBJECT$BUFF(72), 2, '0');
238 2  CALL WRITE(AFT$OBJ, OBJECT$BUFF, 76, STATUS);
239 2  END CREATE$OBJECT;
240 1  DCL$EXTRN$SYM:
    PROCEDURE
        PUBLIC;
241 2  CALL WRITE(AFT$OBJ, ('EXTERNAL DECLARED',CR,LF), 19, STATUS);
242 2  END DCL$EXTRN$SYM;
243 1  DCL$PUBLIC$SYM:
    PROCEDURE
        PUBLIC;
244 2  CALL WRITE(AFT$OBJ, ('PUBLIC DECLARED',CR,LF), 17, STATUS);
245 2  END DCL$PUBLIC$SYM;
246 1  OBJECT$SPACE
    PROCEDURE
        PUBLIC;

```

```
247 2 STATUS = 0;
248 2 END OBJECT$SPACE;
249 1 OBJECT$FINISH:
      PROCEDURE
      PUBLIC;
250 2 CALL WRITE(AFT$OBJ, ('FINISHED !!!!!', CR, LF), 14, .STATUS);
251 2 END OBJECT$FINISH;
252 1 END OBJECT$CODE;
```

MODULE INFORMATION:

```
CODE AREA SIZE      = 01C4H      452D
VARIABLE AREA SIZE = 0012H      18D
MAXIMUM STACK SIZE = 0008H       8D
422 LINES READ
0 PROGRAM ERROR(S)
```

END OF PL/M-80 COMPILATION

'M-80 COMPILER CHECK FOR MATCH OF TWO STRING

S-II PL/M-80 V3.1 COMPILATION OF MODULE STNGMATCH
 ECT MODULE PLACED IN : F4: SMATCH. OBJ PRINT(: F1: SMATCH. LST) DEBUG PAGELENGTH(50)
 IPILER INVOKED BY: PLM80 SMATCH. PLM DATE(27-MAR-82) OBJECT(: F4: SMATCH. OBJ)

```

1      $TITLE('CHECK FOR MATCH OF TWO STRING')
      STNG$MATCH:
      DO,

2      1  DECLARE REV820320          BYTE AT(0);

/* WRITTEN BY JOSEPH R. GARAPPOLO    29 APRIL 1981

INCLUDE FILES - INC .ELD

PARAMETERS:  BUFF$1$PTR - pointer to buffer 1.
              BUFF$2$PTR - pointer to buffer 2.
              COUNT - number of byte to be compared

DESCRIPTION:

This routine compared the two string for a match. If the match fails
if returns the reason.

*/
$NOLIST

```

/M-80 COMPILER CHECK FOR MATCH OF TWO STRING

```

$EJECT
25 1  STRING$MATCH;
    PROCEDURE (BUFF$1$PTR, BUFF$2$PTR, COUNT) BYTE PUBLIC;

26 2  DECLARE BUFF$1$PTR      POINTER;
27 2  DECLARE BUFF$2$PTR      POINTER;
28 2  DECLARE COUNT           BYTE;

29 2  DECLARE (BUFF$1 BASED BUFF$1$PTR) (1) BYTE;
30 2  DECLARE (BUFF$2 BASED BUFF$2$PTR) (1) BYTE;
31 2  DECLARE I               BYTE;

32 2  DECLARE MATCH           LITERALLY '0';
33 2  DECLARE L$THAN          LITERALLY '1';
34 2  DECLARE G$THAN          LITERALLY '2';

35 2  I = 0;
36 2  DO WHILE (I < COUNT);
37 3  IF (BUFF$1(I) < BUFF$2(I))
    THEN DO;
        /* check for end-of-macro in second buffer and a
        blank in the first buffer. If so we have a match */
        IF (BUFF$2(I) = EOIDENT)
        THEN DO;
            IF (BUFF$1(I) = ' ')
            THEN RETURN MATCH;
            END;
            /* string does not match return L$THAN status */
            RETURN L$THAN;
        END;
        ELSE
        IF (BUFF$1(I) > BUFF$2(I))
        THEN DO;
            /* string does not match retrun G$THAN status */
            RETURN G$THAN;
            END;
            I = I + 1;
        END /* WHILE */;
        /* we have a match */
        RETURN MATCH;
    END STRING$MATCH;
54 1  END STNG$MATCH;

```

/M-80 COMPILER CHECK FOR MATCH OF TWO STRING

MODULE INFORMATION:

CODE AREA SIZE	= 00BAH	138D
VARIABLE AREA SIZE	= 0006H	6D
MAXIMUM STACK SIZE	= 0002H	2D
103 LINES READ		
0 PROGRAM ERROR(S)		

ID OF PL/M-80 COMPILATION

SIS-II PL/M-80 V3.1 COMPILATION OF MODULE GETASCII
 3 OBJECT MODULE REQUESTED
 COMPILER INVOKED BY: PLM80 GASCII.PLM DATE(10-MAR-82) NOOBJECT PRINT(:F1:GASCII.LST) PAGELENGTH(50)

```

1      $TITLE('GET VALUE OF ASCII STRING, WORD OR BYTE ')
      GET$ASCII:
      DO;

```

```

2      1      DECLARE REVB10730          BYTE AT(0);

```

```

/* WRITTEN BY JOSEPH R. GARAPPOLO      27 APRIL 1981

```

```

INCLUDE FILES ~ INC      ELD
                      GETSYM.EPD

```

PARAMETERS:

1. POINTER TO BUFFER WHERE VALUES ARE TO BE RETURNED
2. LENGTH OF VALUE BUFFER
3. DATA SIZE (word or byte)

DESCRIPTION:

This routine converts the ASCII string in LINE BUFFER, surrounded by quotes, or up to the length of the value buffer, into the DATA SIZE specified. The data is put into the return value buffer. To include a quote in the string it must be adjacent to another quote. Word values are made up of two characters, and are stored high order byte first followed by the low order byte. This is the format required by the 8085 microprocessor.

Upon exiting the global data structure ASCII\$STATUS contains the actual number of bytes returned and the error status.

A status of TRUE is returned if the string is terminated by a quote before the value buffer is filled. If a quote is not found before the buffer is filled, or an end of line is encountered a status of FALSE is returned.

```

/*
$NOLIST

```

```

49 1 PUT$BAK:
50 2 PROCEDURE
51 1 END PUT$BAK;

51 1 DECLARE ASCII$STATUS
      ERROR
      ACT$LNG

      EXTERNAL;

      STRUCTURE(
      BYTE,
      BYTE) PUBLIC;
```


\$EJECT

```

52 1  G$ASCII:
    PROCEDURE (BUFF$PTR, STRING$LONG, DATA$SIZE) BYTE PUBLIC;

53 2  DECLARE BUFF$PTR          POINTER;
54 2  DECLARE STRING$LONG      BYTE;
55 2  DECLARE DATA$SIZE      BYTE;

56 2  DECLARE (VALUE BASED BUFF$PTR) (1) BYTE;

57 2  DECLARE I                BYTE;
58 2  DECLARE J                BYTE;
59 2  DECLARE CHAR              BYTE;
60 2  DECLARE HIGH$ORDER       BYTE;

61 2  POS$I:
    PROCEDURE;

/* This procedure sets the index so that the high order and low order
   bytes appear in the correct locations */

62 3  IF (DATA$SIZE = BYTE$LONG)
64 3  THEN I = I + 1;
    ELSE IF HIGH$ORDER
    THEN DO;
        HIGH$ORDER = FALSE;
        I = I + 3;
        END;
    ELSE DO;
        HIGH$ORDER = TRUE;
        I = I - 1;
        END;

73 3  END POS$I;

74 2  DO I = 0 TO STRING$LONG - 1;
75 3  VALUE(I) = NULL,
76 3  END;

/* initialize status and actual length */

77 2  ASCII$STATUS.ERROR = NULL,
78 2  ASCII$STATUS.ACT$LONG = 0;

/* set initial index depending on data size */
79 2  IF (DATA$SIZE = BYTE$LONG)

```

```

81 2      THEN I = -1;
82 3      ELSE DO;
83 3          I = -2;
84 3          HIGH$ORDER = TRUE;
85 3      END;
86 2      CHAR = GET$CHAR;
87 2      DO WHILE (CHAR <> CR);
88 3      CALL POS$I; /* set I index to next location in value */
89 3      IF (I >= STRING$LNG)
90 4      THEN DO;
91 5          /* the buffer is full see if the last character if a quote
92 5          and make sure the quote is not ment to be included in the
93 5          string before deciding to return a good or bad status. */
94 5          IF (CHAR = '''')
95 6          THEN DO;
96 6              CHAR = GET$CHAR;
97 6              IF (CHAR = '''')
98 6              THEN DO;
99 6                  /* quote was ment ot be in string
100 6                  return a bad status */
101 6                  CALL PUT$BAK;
102 6                  CALL PUT$BAK;
103 6                  RETURN FALSE;
104 6              END;
105 6              /* the quote was a terminating quote return a
106 6              good status */
107 6              CALL PUT$BAK;
108 6              RETURN TRUE;
109 6          END;
110 6          ELSE DO;
111 6              /* last character was not quote return bad status */
112 6              CALL PUT$BAK;
113 6              RETURN FALSE;
114 6          END;
115 6          END;
116 6          IF (CHAR = '''')
117 6          THEN DO;
118 6              /* see if quote is ment ot be included in string or if
119 6              the quote is a string terminator */
120 6              CHAR = GET$CHAR;
121 6              IF (CHAR <> '''')
122 6              THEN DO;
123 6                  /* terminator quote, get ready for return */
124 6                  CALL PUT$BAK;
125 6                  IF (DATA$SIZE <> BYTE$LNG) AND
126 6                  ((ASCII$STATUS.ACT$LNG MOD 2) <> 0)

```

./M-80 COMPILER GET VALUE OF ASCII STRING, WORD OR BYTE

```

15 6      THEN DO; ASCII$STATUS.ACT$LNG =
16 6      ASCII$STATUS.ACT$LNG + 1;
17 6      VALUE(J-1) = VALUE(J);
18 6      VALUE(J) = NULL;
19 5      END;
20 5      RETURN TRUE;
21 4      END;
22 3      ASCII$STATUS.ACT$LNG = ASCII$STATUS.ACT$LNG + 1;
23 3      J = I;
24 3      /* store value */
25 3      VALUE(I) = CHAR;
26 3      CHAR = GET$CHAR;
27 2      END /* WHILE */;
28 2      /* end of line reached set error type and return bad status */
29 2      ASCII$STATUS.ERROR = 'Q';
30 2      RETURN FALSE;
31 2      END G$ASCII;
32 1      END GET$ASCII;

```

MODULE INFORMATION:

```

CODE AREA SIZE      = 015BH  347D
VARIABLE AREA SIZE = 000AH   10D
MAXIMUM STACK SIZE = 0006H    6D
299 LINES READ
0 PROGRAM ERROR(S)

```

END OF PL/M-80 COMPILATION

./M-80 COMPILER FIND LENGTH OF AND ASCII STRING STARTED AND ENDED BY QUOTES

IS-II PL/M-80 V3.1 COMPILATION OF MODULE ASCLNG
OBJECT MODULE REQUESTED
MPILER INVOKED BY: PLM80 ASCLNG.PL M DATE(10-MAR-82) NDOBJECT PRINT(:F1:ASCLNG.LST) PAGELENGTH(50)

\$TITLE('FIND LENGTH OF AND ASCII STRING STARTED AND ENDED BY QUOTES')

ASCLNG:

DO;

2 1 /* DECLARE REV810520 BYTE AT(O);

WRITTEN BY JOSEPH R. GARAPPOLO 26 APRIL 1981

INCLUDE FILES - INC .ELD
GETSYM. EPD

DESCRIPTION:

This routine returns the length of an ASCII string surrounded by Quotes.
*/

\$NOLIST

./M-80 COMPILER FIND LENGTH OF AND ASCII STRING STARTED AND ENDED BY QUOTES

```

$EJECT
49 1 PUT$BAK:
50 2 PROCEDURE
51 1 END PUT$BAK;
52 2 EXTERNAL;
53 1 G$ASCII$LNG.
54 2 PROCEDURE
55 2 DECLARE CNT
56 2 DECLARE CHAR
57 2 DECLARE LNG
58 2 BYTE;
59 2 BYTE;
60 2 BYTE;
61 2 LNG = 0;
62 2 /* save the current index into LINE BUFFER */
63 2 CNT = G$CHAR$CNT;
64 2 CHAR = GET$CHAR;
65 2 DO WHILE (CHAR <> CR);
66 2 LNG = LNG + 1;
67 2 IF (CHAR = ',')
68 2 THEN DO;
69 2 /* see if quote is terminator or is ment to be included
70 2 in string */
71 2 CHAR = GET$CHAR;
72 2 IF (LINE$BUFF(CNT) <> ',')
73 2 THEN DO;
74 2 /* quote is terminator restore LINE BUFFER
75 2 index */
76 2 CALL G$CHAR$INIT(CNT);
77 2 RETURN LNG-1;
78 2 END;
79 2 ELSE LNG = LNG - 1;
80 2 END;
81 2 ELSE CHAR = GET$CHAR;
82 2 END /* WHILE */;
83 2 /* end of line reached, restore LINE BUFFER index */
84 2 CALL G$CHAR$INIT(CNT);
85 2 RETURN LNG;
86 2 END G$ASCII$LNG;
87 1 END ASCLNG;

```

MODULE INFORMATION

PL/M-80 COMPILER FIND LENGTH OF AND ASCII STRING STARTED AND ENDED BY QUOTES

CODE AREA SIZE = 0063H 99D
VARIABLE AREA SIZE = 0003H 3D
MAXIMUM STACK SIZE = 0002H 2D
196 LINES READ
0 PROGRAM ERROR(S)

ID OF PL/M-80 COMPILATION

SIS-II PL/M-80 V3.1 COMPILATION OF MODULE GETSTRING
) OBJECT MODULE REQUESTED
)MPILER INVOKED BY: PLM80 GSTRNG.PLM DATE(10-MAR-82) NOOBJECT PRINT(:F1:GSTRNG.LST) PAGELENGTH(50)

```

1      GET$STRING:
      DO;

2      1      DECLARE REV810429          BYTE AT(0);

/* WRITTEN BY JOSEPH R GARAPPOLO      29 APRIL 1981

INCLUDE FILES INC   ELD
                  GETSYM.EPD
  
```

PARAMETERS : 1. POINTER TO DESTINATION BUFFER
 2. STOP CHARACTER
 3. MAXIMUM NUMBER OF CHARACTERS

DESCRIPTION.

This routine copies characters from the LINE BUFFER into the
 DESTINATION BUFFER until the STOP CHARACTER is encountered
 or the maximum number of characters are read

```

*/
$NOLIST
  
```

```

$EJECT
49 1  G$STRING:
    PROCEDURE (DEST$PTR, MAX$LNG, STOP$CHAR) BYTE PUBLIC;

50 2  DECLARE DEST$PTR          POINTER;
51 2  DECLARE MAX$LNG           BYTE;
52 2  DECLARE STOP$CHAR        BYTE;

53 2  DECLARE (DEST BASED DEST$PTR) (1) BYTE;
54 2  DECLARE CHAR             BYTE;
55 2  DECLARE LNG$CNT          BYTE;

56 2  CHAR = GET$CHAR;
57 2  LNG$CNT = 0;
58 2  DO WHILE (LNG$CNT < MAX$LNG) AND (CHAR <> STOP$CHAR);
59 3  DEST(LNG$CNT) = CHAR;
60 3  CHAR = GET$CHAR;
61 3  LNG$CNT = LNG$CNT + 1;
62 3  END; /* WHILE */

63 2  IF (CHAR <> STOP$CHAR)
64 3  THEN RETURN 'S';
65 2  ELSE RETURN NULL;

66 2  END G$STRING;

67 1  END GET$STRING;

```

MODULE INFORMATION:

```

CODE AREA SIZE      = 005CH      92D
VARIABLE AREA SIZE = 0006H       6D
MAXIMUM STACK SIZE = 0002H       2D
188 LINES READ
0 PROGRAM ERROR(S)

```

END OF PL/M-80 COMPILATION

./M-80 COMPILER

SIS-II PL/M-80 V3.1 COMPILATION OF MODULE CLRBUF
 OBJECT MODULE REQUESTED
 COMPILER INVOKED BY: PLM80 CLRBUF PLM DATE(10-MAR-82) NOOBJECT PRINT(:F1:CLRBUF LST) PAGELENGTH(50)

```

1      CLRBUF:
      DO;

2  1      DECLARE REV810912          BYTE AT(0);

/* WRITTEN BY JOSEPH R. GARAPPOLO    29 APRIL 1981

INCLUDE FILES - INC      ELD

PARAMETERS:
      DEST$PTR - pointer to buffer
      LNG - number of bytes to be cleared.

DESCRIPTION:
      This routine fills LNG fields of teh destination buffer with blanks

*/
$NOLIST

```

```

$EJECT
22 1  CLEAR$BUFF:
    PROCEDURE (DEST$PTR, LNG)      PUBLIC;
23 2  DECLARE DEST$PTR             POINTER;
24 2  DECLARE LNG                   BYTE;
25 2  DECLARE (DEST BASED DEST$PTR) (1) BYTE;
26 2  DECLARE I                     BYTE;
27 2  DO I = 0 TO LNG-1;
28 3  DEST(I) = ' ';
29 3  END;
30 2  END CLEAR$BUFF;
31 1  END CLRBUF;

```

MODULE INFORMATION:

```

CODE AREA SIZE      = 002CH      44D
VARIABLE AREA SIZE = 0004H      4D
MAXIMUM STACK SIZE = 0000H      OD
68 LINES READ
0 PROGRAM ERROR(S)

```

END OF PL/M-80 COMPILATION

```

SIS-II PL/M-80 V3.1 COMPILATION OF MODULE COPYBUFFER
NO OBJECT MODULE REQUESTED
COMPILER INVOKED BY: PLM80 COPYBF PLM DATE(10-MAR-82) NOOBJECT PRINT(:F1: COPYBF LST) PAGELENGTH(50)

```

```

1      COPY$BUFFER:
      DO;

2      1      DECLARE REV810611          BYTE AT(0);

/* WRITTEN BY JOSEPH R. GARAPPOLO      29 APRIL 1981

INCLUDE FILES - INC    ELD

PARAMETERS:  DEST$PTR - pointer to destination buffer.
              SRC$PTR - pointer to source buffer
              COUNT - number of byte to be copied.

DESCRIPTION:

This routine copies COUNT bytes from the source buffer to the
destination buffer

*/
$NOLIST

```

./M-80 COMPILER

```

$EJECT
22 1 COPY$BUFFS:
    PROCEDURE (DEST$PTR, SRC$PTR, COUNT) PUBLIC;

23 2 DECLARE DEST$PTR          POINTER;
24 2 DECLARE SRC$PTR          POINTER;
25 2 DECLARE COUNT            WORD;

26 2 DECLARE (DEST BASED DEST$PTR) (1) BYTE;
27 2 DECLARE (SRC BASED SRC$PTR) (1) BYTE;
28 2 DECLARE I                BYTE;

29 2 IF (COUNT <> 0)
    THEN DO;
31 3 DO I = 0 TO COUNT-1;
32 4 DEST(I) = SRC(I);
33 4 END;
34 3 END;

35 2 END COPY$BUFFS;

36 1 END COPY$BUFFER;

```

MODULE INFORMATION:

```

CODE AREA SIZE      = 0050H      B0D
VARIABLE AREA SIZE = 0007H      7D
MAXIMUM STACK SIZE = 0002H      2D
75 LINES READ
0 PROGRAM ERROR(S)

```

END OF PL/M-80 COMPILATION