CRASMB

The Lloyd I/0
Macro Cross Assembler
for
FLEX and OS-9

Lloyd I/O
P.O Box 387
Chehalis, WA 98532

REGISTER YOUR COPY

When you purchase CRASMB, you purchase a single user/system license to use it. The product is supplied AS IS, however you may receive updated versions for the operating system you use if you have registered your copy with LLOYD I/O. You will be mailed a notice of changes and new products which may add to the available CPM'S, and other systems on which they operate.

The restriction are that you must send a diskette of the size you need copied with the copies of the programs to be replaced on it in the original name and extension. Return postage MUST be included.

To register fill out the form below and mail to:

LLOYD I/O
19535 NE GLISAN STREET
PORTLAND, OR 97230

cut here ----------

NAME..... TITLE.....

COMPANY.....

ADDRESS.....

CITY ..... STATE ..... ZIP .....

CRASMB VERSION ( )FLEX ( )OS9

CPM's .....

DISK SIZE .....

DEALER .....

DATE .....

OS9 VERSION 2 FLEX VERSION 4

JANUARY 1983

COPYRIGHT NOTICE

PRODUCT WARRANTEE INFORMATION

The CRASMB user manual, source code and object code software is supplied AS IS and without warrantee. Reasonable care has been taken to insure that the software does function as described in this manual. If you find a situation where the assembler or a CPU personality module does not function as the manual describes, then contact LLOYD I/O. An attempt will be made to correct any errors brought to our attention, however we make no guarantee to do so.

CUSTOM MODIFICATIONS AND VERSIONS

Upon request, LLOYD I/O will provide custom designed CPU personality modules (CPM's) at the rate of $35.00 per hour under contract agreement. (Notification of any price change will not be made.) Exact details may be requested.

Write, LLOYD I/O, 19535 NE GLISAN STREET, PORTLAND, OR 97230.

T A B L E 0 F C 0 N T E N T S

INTRODUCTION

CRASMB is a fast and versatile (8 bit) macro cross assembler. It has the necessary elements to support structured construct like WHILE and FOR etc. These are the ability to define macros with substitutable parameters, conditional assembly directives, and the ability to change the value of a label or symbol . In addition, source code may be assembled in modular form. That is as a series of LIBrary called files. A short file containing the list of file specifications in standard assembler source format may call as many library files as desired. Symbols default to a maximum length of 6 for Flex and 8 for OS9, but may be edited to a maximum length ranging from 3 to 30 characters. This manual reflects both FLEX and OS9 versions. Differences will be noted, and these will be only where I/O (input/Output) is involved with the system software. Items enclosed within the greater-less than signs ('<', '>') are required, and items enclosed within brackets ('[', ']') are optional.

The assembler achieves CPU mnemonic to object code translation by the use of CPU personality modules (CPM's). In Flex the assembler handles the care and feeding of a CPM as it is loaded into memory at the end of the assembler. The assembler's storage area begins at the new end of the module. The OS9 assembler lets OS9 load or link the module to the assembler. The storage area is located to the assembler at run-time and defaults to about 11K. There are currently modules available for the 6809, 68OO-2, 6801-3, 68O5, 8080-5, Z8O, 1802, and 6502.

The assembler operates in a position independent manner, and requires the CPM's to do the same. This requirement allows later versions of the assembler to be issued which still work with existing CPM'S. The assembler uses the U (user stack pointer) register to point to all scratch pad, file control blocks, flags, pointers, CPM subroutine addresses,... etc. When called from FLEX the end of memory ($CC2B) is used to allocate this area from the end of memory downwards for the needed space. The OS9 version will allocate a minimum of about 11K of memory, which may be anywhere in ram, of which 5K is the symbol table. All offsets to the scratch area are positive. Very little changes are made to CPM source code files to make then run under either operating system.

INVOKING THE ASSEMBLER

The invoking syntax for the assembler is:

FLEX:

```
CRASMB,<source code file spec..>[,<object code file spec..>]
      [,<CPM file spec.>],SWITCHES,PARAMETERS
```

OS9:

```
CRASMB <source path list> [<Q=SWITCHES> <O=Object code path list>
      <C=CPM path list> <P=calling line parameters> ]
```

For Flex the source code file specification defaults to the extension of ".TXT". The object code file specification defaults to the extension of ".BIN" and the name the same as the source code file. The OS9 version uses the normal path lists and execution directory for the CPM's. Only one source code file name is allowed, and one object code file name is allowed, but the assembler directives "LIB" or "USE" may be used within a source code file to specify another file to assemble. See the "LIB" directive description for further details. The CPM file spec. is the name of a CPM file which will be loaded and initialized just before the source code file is accessed on pass one. The default extension is ".BIN". (This feature allows assembling existing programs without having to add the 'CRO' directive.) See the 'CRO' directive for more information. The Calling line parameters follow the same syntax as macro parameters. The OS9 calling parameters must be the last item on the invoking line.

The OS9 object code file will default to the execution directory, using the same file name as the source file. It will have its public read and execute, and other read, write, and execute permission bits turned on.

SWITCHES

There are several option switches that may be switched if they are specified in the SWITCH list in the assembler invoking line.

SUMMARY:

```
Y ... (FLEX ONLY) Automatic object code file deletion. (OFF)
L ... No List option turned on (no assembler output). (LIS)
S ... No Symbol Table list turned on (suppressed). (SYM)
G ... Single line list on FCC, FCS, FDB turned on. (GEN)
B ... (FLEX ONLY) No Object Code saved an disk. (BIN)
P[number] ... PAG option turned on, list starts a page no. (NOP)
N ... Use line numbers. (OFF)
O ... Set to position independent code (OFF)
T ... Start printing comment lines in address field (OFF)
```

The OPTion directive has the following restrictions. 'LIS' will not turn on if it is has been turned off by the invoking switch. 'BIN' will not turn on if it has been turned off by any switch (NOTE: Object code files not controllable from the OS9 option switches.) 'NOP' will not turn off if it has been turned on by any switch.

INVOCATION EXAMPLES

```
Flex.........
    1.  CRASMB TAPE
    2.  CRASMB TAPE+YSP1
    3.  CRASMB TAPE,TAPE.CMD,I6809+YL
    4.  CRASMB TAPE+PGYS
    5.  CRASMB PROG+YLS+$0100,$C100
OS9...........(Syntax for the same result as the Flex examples)
    1.  CRASMB TAPE O
    2.  CRASMB TAPE O Q=SPI
    3-  CRASMB TAPE O=TAPE_CMD Q=L C=I6809
    4'. CRASMB TAPE Q=PGS
    5.  CRASMB PROG Q=LS P=$0100,$C100
```

The major difference in the invocation line is the control of the output file for the object code. In OS9, it CANNOT be automatically deleted. OS9 has no file name extensions, and none are generated by the OS9 version. NOTE: No comma or space is allowed before the '+' sign in the Flex assembler switch settings.

Example 1 could assemble the source code file TAPE.TXT. An object code file by the name of TAPE.BIN would be generated. For Flex, if it already exists the user will be prompted with "DELETE BINARY OBJECT FILE (Y-N) ?". A 'Y'es answer will cause the assembler to continue processing after deleting the object code file. If any other character is typed then the assembler quits and returns to the disk operating system. The assembler listing, and the symbol table would be printed an the output device. All FCC's, FCS's, FCB's, and FDB's would print their full object code listings.

Example two would do the same as example one, however, in Flex, the object code file would automatically be deleted if need be and the assembler symbol table not be printed. However, the listing would begin an page 1. WARNING: The 'P' option followed by a starting page number will surpress all output including all errors. Therefore, you should assemble your source code at least once without any page surpression after any changes are made to that part of the source that would not have been printed or displayed.

Example three will do the same as the above, but the object code file will be "TAPE.CMD" and if it exists, will be automatically deleted. No assembler listing will be generated, but the symbol table will be listed. It will use the CPM file 'I6809.BIN' as specified in the command line. Note, that if the binary file is not given in the command line, then the command line would be:

```
CRASMB TAPE,,I6809+YL
(CRASMB TAPE Q=L C=I6809)
```

The two commas separating the source file name and the CPM file name must be used in this instance. (For Flex only.)

In example four, the object file will again default to the source code file name, but with an extension of ".BIN", and if it exists it will be automatically deleted. The PAG and NO GEN options are turned on. The symbol table will not be listed. The assembler list function is defaulted in the ON state. Listing begins on page zero. The 'P' option may have a page number immediately following it, which specifies the first page to start listing. Page numbers range from 0 to 65535.

Example five shows a file which will be assembled and the binary file "PROG.BIN" will automatically be deleted, along with no listing or symbol table listing. However, this example shows the use of the 'calling' line parameters which will be substituted much like macro parameters. For Flex the second plus sign must be used to tell CRASMB that the parameters follow. Parameters are always the last item on the invocation line. These will be inserted for the '&'- sign which is immediately followed by the characters 'A' to 'I' for 1 to 9 which are used in the macro parameter substitution syntax. The characters 'A' to 'I' are used to distinguish between the assembler calling line parameters and macro calling line parameters.

ASSEMBLER DESCRIPTION

This assembler was written to accept standard assembler free format source code files. It follows then that the user must be familiar with assembly language, and more particularly, with the Motorola format.

The source code file must contain ASCII characters between $20 through $7E. The ASCII character $OD is also allowed as an end of line marker. (Standard editor format.)

Each line may be a Maximum of 127 characters followed by a carriage return ($OD). Four fields are recognized by the assembler as valid code to process. These consist of (from left to right) the LABEL, the OPERATOR (mnemonic), the OPERAND, and the COMMENT. Fields are separated by one or more space characters ($20). Form:

[label] <operator> [operand] [comment]

There are two types of comment lines allowed. A line which consists of just a carriage return, and any line which begins with a '*'. Traditionally comment lines begin to be printed at the same column in which the label field begins . However, this limits the length of the comment before it reaches the right side of the paper. This assembler breaks tradition and allows the printing of comment lines to start in column eight, the same position as the address or in the standard position of the start of the label field. The option TAB (TAB, NOT, -T) is used to select which form. The default is the standard form. In addition, when a line reaches the maximum line length (defaulted to 80, but may be set by 'LEN'), a new line is issued. For comment lines, the comment begins the same column again, but a '* ' (asterisk, space) is inserted. All other lines which reach the maximum line length, issue another line, and then begin in the column where the field normally begins.

The restrictions and option for each field are as follows:

LABEL FIELD

1. The label must begin in column one (1)
2. The label must be unique (except for the SET directive)
3. The label must begin with the letters (A-Z, a-z)
   If the UPS option is on then lower case letters
   are converted to upper case (default).
4. The label must consist of the characters
   'A'-'Z', 'a'-'z', or ' 1' - '9' , or '$' or '.' or '-'
5. The label's first six characters are significant,
   the remainder is ignored, unless changed by SYM.

6. The label must be followed by a space ($20)

OPERATOR FIELD

1. The operator is one to six characters of the letters 'A' to 'Z', 'a' to 'z', and '0' to '9', and are followed by a space ($20).
2. Lower case letters are converted to upper case.
3. Mnemonics that use a register specifier may delete the separating space.

OPERAND FIELD

1. All operands are generally considered an expression
2. The operands are evaluated for the expression value and the addressing mode used.
3. Some instructions do not require an operand so in that case, this field is considered the comment field.

COMMENT FIELD

1. This field is optional.
2. The characters in this field may be any ASCII character from $20 through $7F.
   Note that a total line length of 128 characters including the carriage return is allowed.

EXPRESSIONS

Expressions consist of combinations of numbers or labels separated by the operators. The arithmetic is done in 16 bit integers. Eight bit results are taken from the least significant 8 bits. Expressions must not contain spaces, and the expression is terminated when a space, carriage return, or an illegal character is found.

The arithmetic operators '+', '-', '*', '/'. These do integer addition, subtraction, multiplication, and division respectively.

The logical operators are '&', '|', '!', '>>'and '<<'. These do 16 integer AND, OR, NOT, SHIFT RIGHT, and SHIFT LEFT respectively.

The relational operators are '=', '<', '>', '<>', '<=' and '>='. These operators may be used in conjunction with the conditional assembly directives. They yield a true or false result by comparing the expression on the left with the expression on the right. Their meaning is Equal, Less than, Greater than, Not equal, Less than or equal, and Greater than or

equal respectively. A true conditions results in all ones, and a false condition in zero.

Expressions are evaluated according to the list of operator precedence. More than one operator of the same type without parenthesis are evaluated left to right.

1. Parenthesized expressions
2. Unary plus and minus (+,-)
3. Shift operators (>>,<<)
4. Multiply and divide (*,/)
5. Addition and subtraction (+.-)
6. Relational operators (<,>,<=,>=,<>,=)
7. Logical NOT operator (!)
8. Logical AND and OR operators (&, |).

NUMBERS

There are five types of numbers allowed. These are decimal (default), binary, octal, hexadecimal, and ASCII. This is a summary of the allowed formats for numbers:

BASE	PREFIX	POSTFIX	ALLOWED CHARACTERS
Decimal	none	none	0-9
Binary	%	not allowed	0-1
Octal	@	not allowed	0-7
Hexadecimal	$	nor allowed	0-9, A-F, (a-f)
ASCII	'	not allowed	$20-$7F

type	operation	operator	form
math	add	+	value+value
	subtract	-	value-value
	multiply	*	value*value
	divide	/	value/value
logical	and	&	value&value
	or	|	value|value
	not	!	!value
	shift right	>>	value>>count
	shift left	<<	value<<count
relational	equal	=	value=value
	less than	<	value<value
	greater than	>	value>value
	less than or equal	<=	value<=value
	greater than or equal	>=	value>=value
	not equal	<>	value<>value

SYMBOLS

Symbols are names of expression values. They may be an expression value, the PC value, or the storage counter, all of which may appear in another expression. The first six (Flex) or eight (OS9) characters are significant, and the first characters must be in the ASCII range of (A-Z,a-z). The directive SYM can be used to change the length of significant characters used for symbols. The current PC value is defined by the character '*'. The current storage counter is defined by the character '.' (period). Upper case characters ARE equivalent to lower case characters when the UPS option is on (default).

SYMBOL EVALUATION

This assembler is a two pass assembler, so it follows that a forward referencing symbol must be defined in the first pass. This means that a symbol that references a symbol who's value is an expression containing more forward references, will not evaluate as expected. The 6809 CPM allows for one byte addresses (taken from the lower byte of the symbolic value), and if a forward address is specified in pass one and two, an error would have existed within pass one. If this were the case, an error is indicated, and the longer addressing mode is used instead. Generally, symbols are defined on the first appearance of it whether it is within an expression or in a label field. Several flags are used to indicate whether or not the symbol has really been defined.

AUTO FIELDING

The four fields of the source code are tabulated to an area on the page so that all four fields line up vertically.

AUTOMATIC LABEL GENERATION

Labels may automatically be generated and accessed within expressions. A label generated in this fashion is "Lxxxxx" where 'x' is a decimal digit (0-9). For example the first use of this function gives "L00001". Labels generated are always six characters long.

To create a label use the form...

: <mnemonic>

Where the colon is followed by at least one space. At assembly time the colon will be changed to 'L' and the rest of the line moved right five characters and then the ASCII form of the label counter (after being incremented by one) will follow.

To access a label, use the form...

:<decimal offset>

Where the colon may appear anywhere within an expression as a number or symbolic name. The decimal offset is optional and may be minus, or positive (default). Only decimal digits are allowed.

An example could be:

```
     BRA :1                      BRA LOO482
     FCC "HI",4                  FCC "HI",4
   : EQU *              LOO482   EQU *
```

This function has greater power when used within macros. An example is the BASIC statement PRINT "HI". A macro would be created as...

```
         DEFINED                              EXPANDED

        PRINT  MACRO                           'PRINT "HI"
               LDX #:l                         LDX #L00001
               JSR PSTRNG                      JSR PSTRNG
               BRA :2                          BRA L00002
               FCC "&1",4            L0000I    FCC "HI",4
               EQU *                 L00002    EQU *
               ENDM
```

OBJECT CODE FORMAT

This assembler can generate object code files using one of two selectable formats. The first is the defaulted Flex binary file format. The second is the OS9 format or more generally known as straight output.

The Flex binary file format looks like this:

Byte 0 Start of record ($02, the ASCII STX)
Byte 1 Most significant byte of load address
Byte 2 Least significant byte of load address
Byte 3 Number of data bytes in the record
Byte 4-n The binary object code data in the record

A Flex binary file may also contain an optional transfer address record. This record is used within Flex as the starting location of the program. This record looks like this:

Byte 0 Transfer address indicator ($16, ASCII ACK)
Byte 1 Most significant byte of the transfer address
Byte 2 Least significant byte of the transfer address

The OS9 or 'no record format' is turned on by the option 'NOR' or when an OS9 module is started. (Using the MOD statement.) The object code output under this mode doesn't use any sort of record set up. Therefore, it should only be used with the MOD statement. Object code is assumed to be one continuous memory block.

ASSEMBLER DIRECTIVES

CRASMB supports the following directives or pseudo operators.

SUMMARY

FCC	form constant characters
FCS	form constant string
FCB	form constant byte
FDB	form double byte
SPC	insert spaces in the output listing
LEN	set up length of output line for printing
OPT	switch assembler options
PAG	skip to next page
ORG	define a new origin (*)
RAM	define a new storage counter origin (.)
EQU, SET	(re-) assign a value to a symbol
END, MON	signal end of source code
NAM, TTL	specify a name or title
STTL	specify a subtitle
RMB	reserve memory bytes
ERR	print error message
RPT	repeat following line n times.
MACRO	define a macro
ENDM	end a macro definition
EXITM	exit macro being called
DUP	duplicate lines n times up to 'ENDD'
ENDD	end duplication bracket
IF	conditional assembly control
ELSE	complement true-false flag
ENDIF	end conditional assembly clause
ENDC	end conditional assembly clause
WHILE	incremental conditional assembly control
WELSE	complement sense of WHILE test
ENDW	end WHILE clause
LIB	open a library source code file.
USE	same as the library directive.
CRO	Load or link to CPU personality module.
SYM	Define length of significant characters for symbols.
MOD	start an OS9 module
EMOD	end an OS9 module
OS9	create an OS9 service call

FCC

The function of FCC is to create character strings for messages, or tables. The character string 'text' is broken down to ASCII, with one character per byte. The format is:

```
label FCC delimiter text same delimiter
```

where the delimiter is used to define the starting and ending areas of the string. A maximum of 256 characters may be defined. There is another form for this directive. It is for example:

```
FCC "THIS IS TEXT",$OD,$OA,4
```

This variation allows strings to be set up for printing without having to use the form FCC,FCB. Commas may be used within the delimiters. There may be multiple strings as in this example:

```
FCC /This is text/,$OD,SOA,"This is another line",4
```

or

```
FCC /This, however is a line with a comma which is legal/,4
```

The only expressions recognized are decimal or hexadecimal, or an expression starting with a symbol.

FCS

This directive is identical to the FCC directive, however the last byte generated is modified. The most significant bit is set. This directive is generally used when you are creating an OS9 module. The OS9 operating system uses this form of strings to save memory space.

FCB

FCB is used to evaluate an expression and use the results for an eight bit result. Multiple expressions are separated only by a comma and may generate up to 256 bytes of object code. The format is:

```
label FCB expression l,expression 2,... expression N
```

FDB

The directive FDB is the same as the FCB directive, however 16 bit results are used rather than 8 bit.

SPC

The directive SPC will cause a specified number of spaces to be inserted in the output listing. The format is:

SPC expression

If the PAGE mode is on then the outputting of spaces will not go past the end of the page.

LEN

The length of a line on your printer or screen may be such that changing the normally defaulted 80 characters may be an asset. Some printers allow 132 characters, and using this directive may be an asset. The format is:

LEN expression

Where expression cannot be greater than 255. It should never be set to less than 56. If any field exceeds the line length, a new line will be issued, and the printing will begin at the start of the current field position.

OPT

The directive OPT is used to activate or deactivate the assembler options from within the source code. No label is allowed, and no output is generated. The format is:

OPT option 1, option 2, ... option N

The allowable options are:

ON	OFF	OFF	
SYM	NOS	-S	sorted symbol table listing
GEN	NOG	-G	print all lines of code generated by FCC,FDB,FCS
LIS	NOL	-L	print the assembled source listing (on=default)
PAG	NOP	-P	enable page formatting
BIN	NOB	-B	object code output control
MAC	NOM	-M	macro calling line print control
EXP	NOE	-E	macro expansion print control
CON	NOC	-C	conditional assembly skipped lines print control
OS9	NCO	-O	set up storage counter mode
REC	NOR	-R	output object code using FLEX records (on=default)
TAB	NOT	-T	start printing comment lines next to line number
FDB	NCF	-F	set up bytes generated from FDB directive
UPC	NOU	-U	convert lower case symbol characters to upper case
XIN	NOX	-X	control machine mnemonic instruction set assembly
INN	NOI	-I	use local library file line numbers

The last two characters of the ON options don't have to be present, as the first character is the only significant character. Either the characters 'NO' or '-' in front of the significant character will turn the option off.

EXAMPLES:

```
OPT PAG,EXP,REC
OPT -S,T
OPT L,P,C,E,M,S,-B
```

An explanation is needed here to describe the storage counter mode. Normally position independent code is set up so that it is not self modifying and only contains program instructions and constants. This requires that no RMB's be used within the confines of the first and last instructions. However if the storage counter mode is turned on, either from the calling line options, or the option 'OS9', (actually 'O', will work just fine), any RMB's found will update only the storage counter, and not the program counter. The directive RAM will only set the storage counter. Remember the storage counter can be accessed by the '.' character in place of a number or symbol, just like the '*' accesses the program counter. The MOD directive turns the options OS9 on and REC off (i.e. NOR).

The last option takes precedence aver contradictions. See the invoking option switches section for details and precedence of any preset switches. (Note that if the assembled source listing option is on, then the source program can have control as to what is actually printed.) When the PAG Option is on, automatic page formatting and titling (with page numbers) will occur.

The FDB option causes the two bytes generated from the FDB directive to be interchanged. This is a necessary function for the 6502, 8080, and Z-80 CPM'S. These three CPM's will turn this option on automatically. The 6502, 8080, and Z-80 use two byte addresses which are formed as LSB,MSB in start up vectors, etc. The normal mode for the FDB directive generates MSB,LSB code.

PAG

If the PAGE mode is on this directive will cause the current page to eject. This directive causes a form feed character ($0C) to be sent out to the normal output path or vector.

PAGE

This directive is just an alternate spelling of the PAG directive.

ORG

The ORG directive causes a new origin address (PC) for the subsequent code that follows. The format is:

ORG expression

The default origin is zero (0000). If for any reason the OS9 option is on, the storage counter is updated, not the program.

RAM

This directive sets the current storage counter. The storage counter defaults to zero. Only the directive RMB will update it. If the storage counter mode is ON, then the directive RMB will only update the storage counter, and not the program counter. The format is:

RAM expression

EQU, SET

The EQU (equate) directive is used to assign an expression to a label. No code is generated and the label must appear. The SET (reassign) directive is used to reset a symbolic value without causing a multiply defined symbol error.

label EQU expression

END or MON.

This directive is used to signal an end of the source code. If an expression appears, then the 16 bit result is used as a transfer address for the starting location of the object code being generated if the REC option is on. The format is:

END expression

NAM or TTL

NAM or TTL are used to specify a title which is printed at the top of each page if the PAGE mode is on. The maximum length of the title is 40 characters, and any characters exceeding that are ignored.

STTL

This directive specifies the subtitle which is printed on the line following the main title. It is limited 40 characters in length. The remainder at the line is used for the system date. This area starts at column 42 just under the assembler name and page number.

The page title area is broken down to four fields. The first line contains the NAM or TTL string which is limited to 40 characters and then the last section of the first line contains the assembler title and page number. The second line contains the STTL string in the first forty characters, and the system date in the second half of the line.

RMB

The directive RMB will cause the assembler to add to the current program counter (PC) the value of the expression. It is used to reserve space for data. The format is:

```
label RMB expression
```

If the OS9 option is on, only the storage counter is updated. The directive MOD will turn the OS9 option on automatically so you need not be concerned about using the RMB directive within a MOD-EMOD statement.

ERR

The user may wish to generate an error message of his own definition. Use of this directive will add 1 (one) to the error counter, and print three '*' followed by the text of the error message as defined by the text following the directive. The format is:

```
ERR text of error message
```

RPT

Single source code lines may be repeated from 1 to 127 times. This directive will cause next source code line to be repeated. The format is:

```
RPT expression
```

where the expression is evaluated and if greater than 127 will cause error number 16 (Illegal expression).

MACRO

To define a macro, use this directive. The source code text lines following are placed into a reserved area of memory and are referenced by the name given the macro. The last line of the macro definition must be an ENDM directive, or the text will continue to be defined into the macro space. If all user memory is used the macro cannot be defined, and an error will be displayed. Macros may be defined from within another macro that is being defined, and the text for each macro is not related. Macros may call other macros from within other macros. However, the macro routine uses a stack at the end of user memory and this stack may over-run the symbol, macro name-text tables. The label is the macro name.

The format is:

name MACRO (up to six significant characters)

To call the text for a macro, merely use its name as any mnemonic. If any parameters are to be passed to the macro, then follow the name by a space, then the expressions. Each expression must be followed by a comma (,) if more expressions are to be included.

While macros are being called, parameters will be substituted at any point in the lines in the macro. Up to nine parameters are supported, and are referenced by the character '&' followed by a digit of 1 to 9. If no parameters were defined on the macro call the parameter referencing characters are deleted from the line. Parameters are referenced on a local level to each macro. That is if a macro is called from within another macro, the current macro's parameters are pushed on to a stack. The macro calling line may use the parameter substitution characters to pass on a parameter to a subsequent macro.

The assembler calling line parameters are referenced by the '&' character followed by 'A' to 'I' for parameters 1 to 9 of the assembler calling line parameter buffer. These are in a sense 'global' in nature, and may be called at any time (doesn't have to be within a macro).

Note that parameter substitution is done an a text basis. The same characters for each parameter are substituted for each

reference to it, not the value of any symbol.

The format is:

<label> name expression 1,expression 2.... expression 9

Remember that since macros may be nested it is possible to call one macro from within the other that calls the first macro. This condition would eventually cause a 'macro calling stack' over run error. Also note, that macros may define other macros from within the first macro. The defining process of the second macro occurs at the same time the first macro is defined. This is a change from the TSC assembler, where macros are defined from within other macros when the first one is called (at expansion time)

ENDM

This directive is to signal the end of a macro definition. That is its only purpose. If a macro is not being defined, and this directive is found, it is ignored. No labels are allowed. If a macro was previously being defined, then the previous macro definition continues until another ENDM is found. Internally the first macro will have a split in it where the text is linked together.

The format is:

ENDM

EXITM

This directive will cause the current macro call to be exited. Control will be given to the previous macro or to normal assembly. It is normally used with conditional assembly.

DUP

This directive is used to cause the succeeding lines up to the directive 'ENDD' to be duplicated n times up to 255. It is allowed only within macro calls, and cannot be nested. The 'ENDD' is not printed until the duplicate count is equal to zero.
The format is:

DUP expression

ENDD

This is the closing bracket for the directive 'DUP'. Its only purpose is to mark the end of the duplicate lines in the macro. The DUP, ENDD is similar to a FOR-NEXT loop in BASIC.

IF, IFN, IFxx, IFC, IFNC, IFP1, IFP2, (,SKIP)

These directives are used to start conditional assembly clauses. The form 'IF' will set true if the expression is greater than zero. The form 'IFN' will set true if the expression is equal to zero. The 'N' refers to the compliment of the test results. The succeeding lines will either be skipped (false condition) or assembled (true condition) up to the ENDIF. In either case, if 'ELSE' is found the test condition is complemented. That is, if it was true, ELSE makes is false or if it was false, ELSE makes it true. So assembly may be structured with this conditional.

The syntax is:

IF <expression>

or

IFN <expression>

IFxx <expression>

The form "IFxx" has the following forms: IFEQ (if equal to zero), IFNE (if not equal to zero), IFLT (if less than zero (most significant bit set)), IFLE (if less than or equal to zero), IFGE (if greater than or equal to zero), IFGT (if greater than zero). The form "IF" is the same as "IFNE" and the form "IFN" is the same as "IFEQ". In all cases the value of the expression is compared to zero and the test is performed to see if the condition is true or false.

The forms 'IFC' and 'IFNC' do a string comparison rather than expression comparison. These forms follow a relation of EQUAL or NOT EQUAL to set or reset the conditional test flag. The 'string' may be just a string with no embedded commas or spaces or it may be enclosed in single or double quotes with embedded spaces or commas.

The syntax is:

IFC <string>,<string>

or

IFNC <string>,<string>

Example: IFC RESUL,'MINUS'

In addition to the conditional assembly code there is a form that allows some number of lines to be skipped as a result of a true condition of the directive in use. The directives 'EXITM' and 'ENDIF' are not allowed and will generate 'Unrecognizable Mnemonic' errors if found. The direction and limit of the skip value are limited to minus 255 or plus 255. The skip mode is allowed ONLY WITHIN A MACRO. A backwards skip value must be preceded by a minus sign. The plus sign is optional for forward skips. The syntax is:

IF <test equation>,<skip count>

As an example,

IF COUNT<20,-3

If the symbol COUNT is less than 20, then go back three lines.

The form IFP1 and IFP2 allow for assembly by the either pass one or pass two. Equate files don't have to be assembled again on pass 2, so the form:

```
IFP1
LIB FLEXEQUS
ENDIF
```

will assemble the library file FLEXEQUS on pass 1, and skip it on pass 2. The form 'IFP2' is a complement of the pass one function in that any code between a IFP2 and ENDIF will be skipped on pass one and assembled on pass two.

ELSE

This directive is only recognizable within a conditional clause. It will reverse or compliment the test condition result flag.

ENDIF or ENDC

Conditional clauses are ended with this directive. It has no other function or form. The syntax is:

```
ENDIF
or
ENDC
```

WHILE, WHILEN

This directive is a very convenient method of controlling assembly based upon a symbolic value. Within the network of this clause the assembler will first check the value of a specified symbol for a true or false condition before allowing further assembly. Normally the symbol must be non-zero before a true state is found, however the form WHILEN will allow zero to be the true state for the test. Once assembly has been turned off, it is only possible to turn it on again by WELSE or ENDW. The structure WHILE... ENDW may be nested to any depth as long as there is room on the assembler stack.

The syntax is:

WHILE <symbol name> or WHILEN <symbol name>

WELSE

The WHILE... ENDW test sense is switched with this directive. This is useful for turning assembly back on.

ENDW

The WHILE structure must always have a matching end phrase. ENDW performs the function just like ENDIF does for IF.

The BIG thing to remember about the two conditional phrases, IF and WHILE, is that the IF has first priority, and WHILE has second. Which means at the first level (outer most) IF clauses may control the WHILE clauses. An example of the WHILE structure is ...

```
  WHILE PICK
  ...code.-.
 PICK SET (A<9)|(B<9)
  ... code...
  ENDW
```

The above example shows how the SETting of a symbolic value may control further assembly.

LIB

The LIBrary directive is used to open a source code file for assembly. This directive is very useful for large programs. They may be broken down into many parts, yet only have to be assembled by referencing one major source code file which contains the list of library files to assemble. The Flex assembler will allow a maximum of 13 source files to be open at any time, meaning you can nest the LIB directive up to 12 levels. The main source file counts as the first file.

The OS9 version may nest this directive up to 11 times, i.e. in OS9, 12 files may be open at any time.

If the INN option is on, and the calling line option "N" is selected then the actual line number of the current library file will be printed as the line number in the listing. Error messages always report the line number and the name of the file in which the error came from. The format is:

LIB <file spec. or path name>

The default extension is ".TXT".

USE

This directive is identical to the LIB directive. See the above documentation on the LIB directive to use it.

Examples:

```
LIB 0.DINEW
USE FOUUER
LIB FLEXEQUS
USE /D0/DEFS/OS9DEFS
```

CRO

This directive is used to load or link to a CPU personality module. In Flex the module is loaded from the working drive, to the end of the assembler, and the storage area is adjusted for the difference in size. The module must be written in PIC (position independent code.) For OS9 the module is linked to first, then if it is not found, it is loaded from the execution directory and linked again. Under OS9, the assembler may be loaded anywhere in RAM, and the CPM may be located above or below it. The CPM is not given any storage area, because it uses the same storage area as the assembler. Calls to and from the CPM are made by pointers which are pointed to by offsets from the User stack pointer. The format is:

CRO <File spec. or path name (module name)>

Once a module has been loaded or linked, no more may be loaded or linked. Under OS9, the module will be UNLINKED when the assembler is done with it.

SYM

The SYM directive allows you to define how many characters are to be used for symbols. Normally, (without defining) only 6 (8 for OS9) characters are the significant characters stored in the symbol table. This directive allows you to define the length to any value in the range of 3 to 30 characters. Longer symbols use more memory, so the larger the program being assembled, the shorter the symbols need to be. However, long symbols allow

better program self documentation. The syntax is:

SYM <expression>

where 'expression' must range between 3 and 30. if any symbol is defined before the SYM statement, you will get an error message, because the length will have already assumed its default size.

MOD

This directive is provided to allow you to generate OS9 memory modules with their associated module headers, parity check value, and CRC check values. You probably won't use this feature unless you also can run OS9.

The MOD directive is used at the beginning of an OS9 module. Its function is to create a standard module header and to set the assembler's NOR and OS9 options. It also sets up the CRC check value for the generation of the CRC check value which is placed at the end of the module by the EMOD directive.

MOD uses an operand that is exactly four or exactly six expressions long, each separated by commas. Each expression corresponds to the elements of a module header. See the 'OS-9 System Programmer's Manual' for details.

The following is a description of the MOD operation:

1. The assembler's program address counter ('*') and data address counters ('.') are set to zero, and the internal CRC accumulator and vertical parity generators are initialized. The assembler is put into the NOR and OS9 option modes.

2. The OS9 sync codes $87 and $CD are generated.

3. The first four expressions in the operand list are evaluated and generated as object code. They are:

    a. Module size (two bytes)
    b. Module name offset (two bytes)
    C. Type/language byte (one byte)
    d. Attribute/revision byte (one byte)

4. The header parity byte computed from the previous bytes is generated.

5. If the two optional additional operands are present, they are evaluated and generated. They are:

    e. Execution offset (two bytes)
    f. Permanent storage size (two bytes)

Note that some of the expressions are two bytes long and some are only one byte long.

This directive forces the origin of the object code to zero, so all labels used after this directive are inherently relative to the beginning of the module. This is perfect for the name and execution offsets. The code in the body of the module follows the MOD directive. As the subsequent lines are assembled, the assembler's internal CRC Generator is updated to keep a running CRC value. The EMOD statement (which has no operand) is used to terminate the module. It outputs the correct three byte CRC generated over the entire module. The assembler is left with the NOR and OS9 options selected.

EXAMPLE:

```
MOD <program length>,<name offset>,<type/lang>,<attr/revis.>,
    [<execution offset>,<permanent storage size>]

        type set prgm+objct
        revs set reent+l
             mod pgmlen,name,type,revs,start,memsiz

        temp  rmb 1
        buffer rmb 80

        memsiz equ . data storage size is final "." value

        name FCS /testprogram/

        start equ *
              leax buffer,u get address of buffer
              clr temp
              inc temp
              ldd 80
        LOOP  clr 0,X+
              SUBD 1
              BNE LOOP
              OS9 F$EXIT return to OS9
              EMOD
        pgmlen equ * program size is address of last byte +1
```

EMOD

This directive closes or ends an OS9 memory module. The current CRC value is output as object code. The assembler must have previously used the MOD directive to correctly generate an OS9 module.

OS9

This directive is used as a convenient way to generate OS9 system calls. It has an operand which is a byte value to be used as a request code (function number). The output is equivalent to the instruction sequence:

```
SWI2
FCB operand
```

EXAMPLES:

```
OS9 I$READ
OS9 F$EXIT
```

ASSEMBLER OPERATION

Pass 1

Pass one of the assembler is used to build the symbol table. It is used to resolve forward refewrences in pass two.

Pass 2

Pass two is the main pass of the assembler. The following is a list of the operation performed.

1. Error messages printed.
2. Assembled source listing printed if LIS flag is on.
3. Object code is generated, if BIN flag is on.
4. Closing subroutine of the CPM is executed.
5. Error and symbol count displayed
6. Warning count if any displayed
7. Phasing error if any difference in the last assembled address between pass 1 and 2.
8. LAST ASSEMBLED ADDRESSES if pass 1 and pass 2 are listed if the 'LIS' option is off.
9. Sorted symbol table printed if SYM flag is on.

CUSTOMIZING THE ASSEMBLER

The file 'SCCZERO' is a set of EQUATES that are used by the CPM for communication with the assembler. See the section on NEW CPU PERSONALITY MODULES for details on writing new CPM'S.

There are several things in the assembler which may be changed. They are all located at the beginning of the program. These include a set of vectors, title control, page eject control, default options, flag characters, and the default symbol length. These are defined in the supplied file named OPTION.TXT, which allows you to set, assemble, and append these values to the assembler to create a special version. One of the more useful vectors are the binary object code output vectors which when changed could allow you to send the object code to a serial port, cassette tape, an EPROM programmer, or UGH, memory. The assembler's third title which displays "++++++++ CRASMB V4.0 by LLOYD I/O, All Rights Reserved ++++++++" can be redefined to reflect your own permanent third title.

APPENDING THE ASSEMBLER AND A MODULE

In Flex any of the modules may be appended to the assembler to create a special version of the assembler which when called will already be set up for the CPM needed. This CANNOT be done in OS9.

Since the CPM's are position independent a special process must be followed before the new assembler will work correctly. Step one is to use the following syntax to obtain a map of the assembler.

```
MAP CRASMB
```

The Flex command MAP.CMD must exist on the system disk and the assembler (CRASMB.CMD) must exist on the working drive. A loading map of the assembler will be displayed much like:

```
Loads from $0100 to $2A6F
Transfer address is  $0100
```

Step two is to get a map of the CPM that is to be appended to the assembler. (MAP I6809.BIN for example). The MAP.CMD which is supplied with the assembler will default the file spec extension to CMD. So the exact CPM file spec must be used, including the extension. The map could be displayed as:

Loads from $0000 to $065B

Now add the ending address of the assembler to the ending address of the CPM. (for example $2A6F + $065B = $30CA.) The assembler must now assemble a short file containing basically nothing while the CPM to be appended must be loaded by the assembler. The loading of the CPM must be done by the assembler to get the CPM loaded at the correct address. (The syntax: CRASMB <dummy file spec.>,,<CPM file spec.>+BLS can be used to do that.) When the assembler is done, SAVE the memory from $0100 to the answer of the addition of the ending addresses of assembler and the CPM, and give it a transfer address of $0100. (For example SAVE AS69.CMD,0,30CA,0 will save assembler as a 6809 assembler: note these are not the actual addresses.)

ERROR MESSAGES

The 6809 version of the cross assembler supports 22 error messages which are always printed BEFORE the offending line. They are as follows:

0. <user error message pointed to by X>
1. MULTIPLY DEFINED SYMBOL
2. UNDEFINED SYMBOL
3. ILLEGAL CHARACTER(S)
4. UNRECOGNIZABLE MNEMONIC OR MACRO
5. MNEMONIC REQUIRES A LABEL
6. RELATIVE BRANCH TOO LONG
7. ILLEGAL ADDRESSING MODE
8. USE <PATH> OR LIB <PATH> NESTING LEVEL TOO DEEP
9. OUT OF MEMORY WHILE DEFINING A MACRO
10. OUT OF MEMORY WHILE CALLING A MACRO
11. ILLEGAL REGISTER SPECIFIED
12. ABORTING..MEMORY OVERFLOW
13. MNEMONIC CANNOT HAVE A LABEL
14. PARAMETER SUBSTITUTION ERROR
15. ILLEGAL NESTING
16. ILLEGAL EXPRESSION
17. ILLEGAL 'CPM' CALL
18. MULTIPLY DEFINED MACRO
19. UNBALANCED PARENTHESIS IN EXPRESSION
20. ILLEGAL LABEL OFFSET SPECIFIED
21. ILLEGAL SYMBOL CHARACTERS
22. PHASING ERROR DETECTED

23-255. <user table of error messages pointed to by X>

Error code 0 will print the message pointed to by the index register (X) as a string of characters followed by a $04 (ETX). Error code 23 to 255 will search and print a user defined error code table and messages which is pointed to by the index register (X). The table follows the format...<error code number (single byte)> <error message address (double byte)>... to end of table which is marked by error code zero (0).

Error messages are printed in the following manner:

```
***  message     IN LINE # xxxx OF "file name.ext"
```

The line number is the actual line number of the line which was read from the current file, even if it is a nested library file. This feature tells you exactly which line and which file caused the error.

ADDITIONAL FEATURES

The 6800 CPM supports the extra mnemonics BHS and BLO, which are the logical equivalents of BCC and BCS, respectively. The extra mnemonics are easier to remember and use.

The 6909 CPM supports the mnemonics:

```
ABA     ---     STA B ,-S;ADD A,S+
CBA     ---     STA B ,-S;CMP A,S+
CLC     ---     ANDCC $FE
CLF     ---     ANDCC $BF
CLI     ---     ANOCC $EF
CLV     ---     ANDCC $FD
CLZ     ---     ANDCC $FB
CPX     ---     CMPX
DES     ---     LEAS -1,S
DEX     ---     LEAX -1,X
INS     ---     LEAS 1,S
INX     ---     LEAX 1,X
SBA     ---     STA B,-S;SUB A,S+
SEC     ---     ORCC $01
SEF     ---     ORCC $40
SEI     ---     ORCC $10
SEV     ---     ORCC $02
SEZ     ---     ORCC $04
TAB     ---     TFR A,B ; TSTA
TAP     ---     TFR A,CC
TBA     ---     TFR B,A ; TSTA
TPA     ---     TFR CC,A
TSX     ---     TFR S,X
TXS     ---     TFR X,S
WAI     ---     CWAI $EF

SETDP   ---     SET THE DIRECT PAGE VALUE AND FLAG
                SETDP <EXPRESSION> or SETDP ON or SETDP OFF
REG     ---     Set up a symbolic value to the stack push
                pull bits for PC,U,Y,X,DP,B,A,CC (D=A and B)
```

Examples of SETDP and REG

```
        SETDP   0
        SETDP   OFF
        SETDP   $E8
        SETDP   ON

ALLR    REG     $FF
SOMER   REG     X,Y,D
SOMES   REG     CC,A
```

Extended or direct addressing modes can be forced, but normally the assembler tries to use direct before extended. This is done by comparing the upper byte of a 16 bit extended address operand to the SETDP value. If they are not equal, or the SETDP is OFF, then extended addressing instructions have to be used. You can force extended addressing modes by preceding the operand with a '>' greater than sign, or direct addressing modes by preceding the operand with a '<' less than sign. Examples:

```
STA  >$80      force extended
STA  <PASS     force direct
```

In addition, some accumulator opcodes allow for the form for example:

STA A, STA B, STAA, STAB, STA, and STB. These include ST, OR, and LD. Also mnemonics that call for a register specification character(s) may be followed by a space before the register name. These types of mnemonics usually will have the register name printed where the normal position of acc. A or B was for the 6800 assembler. Some instructions require an addressing mode which may be indexed. Most of those are indicated by the use of a comma (,) within the operand expression. For example "PCR" will not work for a 0,PCR value, it must be ",PCR".

The Z-80 CPM has changed the Z-80 'SET' mnemonic to 'BSET' for bit set, because of a conflict between it and the assembler directive 'SET'.

Also note that since this assembler supports parenthesized expressions, any instruction which would have used parentheses must use brackets '[' and ']'. For example the Z-80 CPM uses [SP],HL instead of (SP),HL.

NEW CPU PERSONALITY MODULES

CRASMB exists because the need for a versatile modular macro assembler was seen. To that end this documentation is provided to enable the user to write his own CPM for the CPU of his choice. It is intended to be used in conjunction with source code to one of the CPM'S.

Please note the available CPM's included in this package, and up-coming in the near future. Currently there exists CPM's for the 6800-2, 6801-3, 6805, 6809, 6502, CDP1802, 8080-8085, and Z80. All run on 6809 systems. Appendix A lists the file names for assembling each CPM for Flex and OS9. Appendix 3 lists the main assembly file for the Flex 6809 CPM. Appendix C is a partial list of some of the library files used by the 6909 CPM. Appendix D is a listing of the main assembly file for the OS9 version of the 6809 CPM. The only differences between Flex and OS9 CPM's are the main assembly file and the library files for the assembler's storage area, the system date title set up, and the OS9 system definitions call. The library files for the main part of the modules are identical.

The assembler directive CRO is used to call a new CPM. It is assumed the file being specified is a binary program specifically written for this assembler and is written in POSITION INDEPENDENT code. When loading in the new CPM in Flex, it is checked for the loading address (which must be zero ($0000). If the starting load address is anything other than the expected loading address, an error will be invoked. If the file specified does not exist, control will be returned back to the disk operating system. The error will be "ILLEGAL 'CPM' CALL".

For OS9 the operation of linking or if need be loading-linking is mostly handled by OS9 system calls. The assembler is given the start of the module, where-ever it is in memory, and its starting execution addresses and it stores these in the storage memory (which is common between the assembler and module both by the direct page register and the user stack pointer.) If the CPM is found in the memory directory, it is not loaded from disk, which means that if it is not found in the memory directory, and attempt will be made to load it from the file specified in the path list following the CRO directive. Program linkage between the module and assembler is maintained by run time pointers on the user stack and direct page registers.

In either case (of Flex or OS9 versions) there are three long branch to subroutines which are at the start of the CPM. The assembler assumes these instructions and will jump to each routine as the call is needed. After the three CPM vectors are six ASSEMBLER vectors used to access subroutines within the assembler.

```
        LBRA INTERP     Start of subroutine.
                        Return equal if mnemonic found and assembled
                        Return NOT equal if mnemonic not found
        LBRA INTINT     Initialize flags and values.
                        Executed when CRO is found and at the
                        beginning of pass 1 and 2.
        LBRA CLOSE      Process any closing functions.
                        For instance report some type
                        of message that indicates the
                        status of relocatable code, etc.
                        must exit with an 'RTS'.
TREES   JMP [CPMTRE,U] Search mnemonic table
PERROR  JMP [CPMPER,U] Print error by number
OBS     JMP [CPMOBS,U] Save byte of object code
SKIP    JMP [CPMSKI,U] Skip past spaces in line
OUTLIN  JMP [CPMOUT,U] Get the character in line
EXPRES  JMP [CPMEXP,U] Evaluate 16 bit expression
```

When assembling a new CPM, the main assembly file calls the library files used to establish the correct assembler storage equates, the operating system equates and the above listed vectors. It will also force the starting locations of the CPM to the correct address. All processing is done through offsets from the user stack pointer (,U). The file T6800.TXT may be used as an example for proper syntax, which is partially listed in appendix C. Appendix B contains a typical main assembly source file for the Flex CPM's and appendix D contains a typical main assembly source file for the OS9 CPM'S.

Appendix A lists main disk files for assembling the various CPM'S. They are short files which call the necessary LIBrary files for the needed assembling subroutines and mnemonic tables. These are listed so you may make sure you have the correct source files for the Flex or OS9 CPM's if purchased.

The following paragraphs describe the three CPM subroutines, and the six ASSEMBLER subroutines. The PAGEFLEX and PAGEOS9 source files contain some equates which are used by the ASSEMBLER subroutines which must be set up properly before calling the routine. They also contain some equates for storage areas which may be used by the CPM in any way needed. The addressing mode normally used to access them is also given.

INTERP - Search and process mnemonic

This is the first vector in the module. The assembler calls this and the next three vectors which are long branches to the correct subroutines.

This is the main mnemonic processing search and process subroutine. It handles the mnemonic table lookup, assembler search call, and multi-way branch to any of several instruction type subroutines. It must return a NOT-EQUAL condition, if the mnemonic was not found. The assembler subroutine TREES is used to perform the search, and it returns an EQUAL condition if the mnemonic was found. Once the mnemonic is found, a branch to a 'type' subroutine which may handle several instructions of the same general type, (such as instructions with no operands), may call some of the other assembler subroutines to process operands, and save object code. Errors may also be reported.

INTINT - Module initialization

This routine is called at the start of pass 1 and pass 2. and when a module is loaded or linked. All of the supplied CPM's set the system date into the page title area under the page number. The 6809 CPM set up flags to handle direct page instructions. The 6502, 8080, and Z-80 CPM's set the OPT FDB option so that FDB-directives will generate interchanged bytes of code.

CLOSE - Report errors,... etc.

This subroutine is called by the assembler when both passes are completed. Currently none of the supplied CPM's do anything, but errors or other types of information may be displayed. This should occur at the end of the listing.

TREES - LBSR TREES

This subroutine is the mnemonic look-up search routine. It performs a binary tree search or serial search for the mnemonic in the current line of source code. Upon entry the index register (X) must point to the start of the table, and accumulator A (items) must contain the count of mnemonics in the table. The variables 'BNAML' (count of characters to compare) and 'ITLEG' (byte count of each mnemonic, separator character, and data table address, which are equal amounts for each mnemonic in the search table, known as the record length), must be set to the appropriate values for the table being searched. If the table contains less than 11 mnemonics a serial search will be done because of the overhead required in a binary tree search. Variable length mnemonics or directives may be in the same table, and when jumping to 'TREES' the count of items (acc. A) must be less than 11.

The flag 'FCMODE' must be set none zero or else the address found with a mnemonic will be jumped to as a subroutine. (Note: this may be a desired function.) Upper and lower case characters in current assembly line are considered the same, assuming the lookup table to be in all upper case.

When the search is complete it returns with a status of NOT EQUAL if the mnemonic is NOT found. At this point the CPM should return control to the assembler, or another table may be searched. If the status of EQUAL is returned to the CPM, then the mnemonic was found and the index register contains the offset address of the data table for that mnemonic. (Note: 'offset address' in the index 'X' register must be adjusted to get the run time address.) The CASMBMAC.TXT file contains a macro which should be used to change the index register. (It adds the program counter to the index register. Check its listing for more information.) The mnemonic table must follow the format of:

```
FCC /??????/,0      or    FCC /??????./
FDB DATA                  FDB DATA
```

In the above example there are six characters per mnemonic. There may be from one to six characters, the separator character and then the address of a data table. Above, a period (.) is used as a separator. (A separator may be a period or a value of 0 to $1F or greater than $7F.)

The above syntax is followed for each mnemonic with the same length until all mnemonics have been defined. The table must be arranged in alphabetical order, or the binary search routine will not work.

For each change in mnemonic length, a different search must be performed with the specified comparison and record lengths on a different mnemonic table.

Appendix C contains the first part of the 6809 CPM to give an example of how to use this routine, and return control to the assembler.

PERROR - LBSR PERROR

This routine prints an error message by the number in accumulator A. It also increments the error counter which is printed at the end of assembly. Error messages are printed in pass 2 and always BEFORE the offending line. Error numbers from 1 to 22 are assembler errors. Error number 0 will print the error message pointed to by the index register (X). Error numbers 23 to 255 will search a user defined table of error codes. This table is pointed to by the index register (X) when calling this routine, and the table must follow the format of <error code number>, <error code message address>. If an error code of zero is found, it marks the end of the table. The format is:

```
FCB 22
FDB ERR22
```

```
    FCB 23
    FDB ERR23
    FCB 0 MARK END OF TABLE
```

OBS - LBSR OBS

To save any object code information, load accumulator A with the data, and then use this subroutine. The program counter (PICK) will be incremented in addition to saving the code.

SKIP - LBSR SKIP

Sometimes finding the starting point of information in the source code line is necessary for further processing by a CPM. This routine will find a space, then skip all spaces to the first non-space character in the input line. If a space is not found before the end of the line is found, then SKIP returns in a NOT EQUAL condition. If a space was found then an EQUAL condition is returned.

OUTLIN - LBSR OUTLIN

To get the next character from the source code line, the CPM may use this routine. If the end of the line is found a NOT EQUAL condition is returned, else an EQUAL condition is returned.

EXPRES - LBSR EXPRES

This subroutine will resolve an expression. LX must point to the first character of the expression. If the expression is resolved without error, the result is returned in the index register with an EQUAL condition. If an error occurs, such as an undefined symbol or illegal character for a specified base, a condition of NOT EQUAL is returned along with the error code. (Error numbers 2 and 16 respectively.)

CPMBUF - CPMBUF,U (LEAX CPMBUF,U)

This is a 64 byte scratch pad area which may be used for any purpose by the CPM. Its main purpose it to give the current user his own scratch area.

CPMTIT - CPMTIT,U (LEAX CPMTIT,U)

This is a 40 character buffer which is used as the second part of a two part title by the assembler as a header on each page when the PAG option is turned on. Currently the CPM's get the system date and convert it to a string such as 'March 18, 1992'. The string is followed by a $04 (eot). A close look at the file 'TIME.TXT' will show how this is done.

PASS - TST PASS,U

This is the pass number flag. If it is zero, then pass one is in operation, but if it is non-zero then pass two is in operation.

BNAML - STA BNAML,U

This 8 bit register is used for the TREES subroutine to establish the length of the comparison characters to use. A maximum of six (6) may be used. It must be set to the appropriate value before using the TREES routine.

ITLEG - STA ITLEG,U

This 9 bit register holds the total length of bytes used for each mnemonic, separator character, and table address. It is referred to as the record size.

FERS - TST FERS,U

This flag is used as a general purpose error flag. It is treated as a local variable only. That means the CPM cannot use it from one call to another and expect it to retain its previous value.

TYPE - STA TYPE,U

This variable is treated as global in nature. It is set to the addressing mode type and expression type from the following table by the CPM before returning to the assembler.

0 No code, no expression
1 No expression used
2 Expression used
3 EQU with expression
4 Accumulator referenced, No expression
5 Accumulator referenced with expression
6 FDB, FCB types
7 Accumulator with indexed expression
8 Indexed expression
9 FCC only type
10 Indexed expression with post byte and offset
11 Accumulator version of number 10

The TYPE variable is used by the assembler to determine the how to print the assembled line. If the value is negative the opcode will be treated as a 16 bit instruction.

The format for printing the opcode is as follows:

```
IIII PP DDDD AAAA
 !   !   !    !
 !   !   !    ----- absolute address of a relative instruction
 !   !   ---------- 8 or 16 bits of data
 !   -------------- indexing post byte
 ------------------ 8 or 16 bit instruction
```

FCMODE STA FCMODE,U

This flag is used by the routine TREES to determine if the table address associated with a mnemonic is an address to execute or an address of an opcode table. If it is zero, TREES will call that address as a subroutine.

PICK - LDD PICK,U

This 16 bit variable is the assembly program counter. It is set by the directive ORG and incremented any time object code data is saved by the routine OBS. It may be used by the CPM for such things as determining addressing distances, etc.

RAMK - LDD RAMK,U

This is the current storage counter. It is accessed by the '.' as opposed to the '*' for the program (instruction) counter.

LX - LDX LX,U

This is the current position in the source code line. It is used by OUTLIN, SKIP, TREES, and EXPRES. Note that it contains the actual address of the current position, and is not an offset. The syntax 'LDA [LX,U]' will get the current character into accumulator A.

GREAT - STA GREAT,U

This character variable is normally set to a space. It is printed when the list option is on, after the line number and before the address. The 6809 CPM sets it to a '>' if short branches could be used for long branches. Also it is set to a 'W' when extended addressing modes are used when the OS9 (position independent code) flag is on. It is set to a 'D' for instructions that affect the storage counter, like the RMB or ORG pseudo directives. You may set it to any printable character between a space ($20) and a tilde ($7E).

RELADD - STX RELADD, U

The CPM's which use some kind of relative addressing modes display the absolute value of the address expression after the last byte of the code field and just before any possible label. The value displayed there is the value in this 2 byte variable. However the flag PRELAD,U must be set to a non-zero value to indicate to the assembler that the RELADD,U value is to be displayed.

PRELAD - STA PRELAD,U

This is the flag which must be set so that the value in RELADD,U will be display in its field.

FBBF - STA FBBF,U

This is the flag, which when set will reverse the order of the two bytes of code generated from the FDB directive. The 6502, 8080, and Z-80 CPM's set this flag.

ADACC - STA ADACC,U

This variable is used by a CPM subroutine that checks for a reference to an accumulator. The routine can clear it first, then check for the reference type. For example, the 6800 CPM sets it to one for accumulator A or to two for acc. B.

ADMODE - STA ADMODE,U

Most CPU types have several addressing modes. The 6800 CPM sets this variable to one of three values depending on the addressing mode. They are 1 for immediate, 2 for indexed, and 3 for extended.

The next four 16 bit variables may be used by the CPM. The supplied CPM's use these in the following manner.

XX1 - STX XX1,U

XX1 is used for the storage of the object code table address after returning from TREES.

XX2 - STX XX2,U

The results from an expression evaluations are stored here.

XX3 - STX XX3,U

The value of a branch instruction is stored here. It is checked for a branch-to-far condition.

XX4 - STX XX4,U

The current source code line pointer (LX) is stored here temporarily while checking for a reference to an accumulator.

WARN - LDD WARN,U

This is a counter which a CPM may update to reflect any type of warning messages which have been reported. When pass 2 is complete the value stored here is checked and if found to be not equal to zero, its value will be printed as "TOTAL WARNINGS :". The 6809 CPM counts the warnings generated from using extended addressing modes while the assembler is in the position independent code generation mode (OPT OS9).

PCTMOY - LDX PCTMOY,U

This five byte buffer holds the PC (PICK) value and symbol name address (in symbol table) and a flag that indicates whether or not the symbol value was set by a assembler directive. PCTMOY+0 is the value, PCTMOY+2 is the symbol address, and PCTMOY+4 is the flag which is set if a label is found in a source line, and must be reset if the value at PCTMOY+O is not to be set into that symbol.

WSYM - LEAX WSYM,U

This is the mnemonic and symbol name storage area where comparisons are made. The assembler subroutine TREES uses this buffer for the binary tree search, but the input line for the serial search. The serial search is used to find difficult mnemonics and operands. The source code the Z80 CPM shows extensive use of the TREES search routine.

The following is an outline of the steps used to write a CPM.

I. Study the CPU instruction set.

- A. Determine all address modes.
- B. Determine like instructions.
- C. Classify instructions of equal length mnemonics
  1. There will be only a few different subroutines to handle all instruction types.
  2. Form a map of the instructions.
     a. Address modes Vertically.
     b. Like instructions horizontally.

II. Build or Edit the mnemonic tables.

- A. Build the mnemonic records in alphabetical order.
  1. The spelling.
  2. The separator character.
  3. The mnemonic op-code table address.
- B. Build the op-code tables for each mnemonic.
  1. Like instructions have like structure so the same CPM subroutine may handle them.
  2. The following general structure is recommended
     a. Interpreter subroutine address.
     b. Instruction op-codes.
  3. See the source code for one of the CPM'S.

III. Write or encode the subroutines to handle each type of instruction.

- A. The 6800 CPM has 9 routines.
- B. The 6502 CPM has 11 routines.
- C. The 6805 CPM has 7 routines.
- D. The 68O9 CPM has 20 routines.
- E. The 1802 CPM has 5 routines.
- F. The 8080 CPM has 11 routines.
- G. The Z-80 CPM has 19 routines.

IV. Write or encode the main CP-M as follows.

- A. Set FCMCDE as not equal.
- B. Set BNAML to the length of the mnemonics.
- C. Set ITLEG to BNAML plus one for the separator character and plus two for the op-code table address. Set to 6 for three character mnemonics.
- D. Set the index register (X) to the address

of the alphabetized mnemonic table.

E. Set accumulator A to the number of mnemonics in the table.

F. Jump to subroutine TREES.

G. If NOT EQUAL condition returned then EXIT.

H. If EQUAL condition returned then do this.
   1. Save the index register in XX1.
   2. Clear TYPE.
   3. Clear ADACC.
   4. Load the index register with the first two bytes (instruction handler address) of the opcode table. (LDX 0,X)
   5. Then jump to the address in the index register. (JMP 0,X)

I. The instruction handler subroutine will now execute and translate the instruction in accordance with the addressing mode found.

V. When the instruction is assembled the instruction handler must return control back to the assembler with the following instructions.

A. CLR A (Set EQUAL CONDITION)
B. RTS (Return from subroutine)

The reason an EQUAL condition must be returned is that if not the assembler will assume that the mnemonic was not found by a call to the CPM.

A close study of the source code for the CPM's purchased with this product will answer many questions the programmer may have concerning writing the CPM.

APPENDIX A

FILE FLEX	OS9 (from device directory)	DESCRIPTION
COMMON		
SCCMACRO.TXF	CPMS/SCCMACRO	(macro definitions)
SCCZERO.TXT	CPMS/SCCZERO	(assembler storage)
SCCITIME.TXT	CPMS/SCCOTIME	(Date (time) title setup)
6800		
SC6800.TXT	CPM6800/SC6800	MAIN ASSEMBLY FILE
SC6800T.TXT	CPM6800/SC6800T	LIBRARY FILE
SC6800B.TXT	CPMS/SCBRANCH	(Relative branch routines)
SC6800M.TXT	CPM6800/SC6800M	LIBRARY FILE
6801		
SC680l.TXT	CPM6801/SC6801	MAIN ASSEMBLY FILE
SC6801T.TXT	CPM6801/SC6801T	LIBRARY FILE
SC6801B.TXT	CPMS/SCBRANCH	(Relative branch routines)
SC6801M.TXT	CPM6801/SC6801M	LIBRARY FILE
6805		
SC6805.TXT	CPM6805/SC6805	MAIN ASSEMBLY FILE
SC6805T.TXT	CPM6805/SC6805T	LIBRARY FILE
SC6805B.TXT	CPMS/SCBRANCH	(Relative branch routines)
SC6805M.TXT	CPM6805/SC6805M	LIBRARY FILE
6809		
SC6809.TXT	CPM6809/SC6809	MAIN ASSEMBLY FILE
SC6809T.TXT	CPM6809/SC6809T	LIBRARY FILE
SC6809S.TXT	CPM6809/SC6809S	LIBRARY FILE
SC6809U.TXT	CPM6809/SC6809U	LIBRARY FILE
SC6809B.TXT	CPM6809/SC6809B	LIBRARY FILE
SC6809M.TXT	CPM6809/SCS809M	LIBRARY FILE
6502		
SC6502.TXT	CPM6502/SC6502	MAIN ASSEMBLY FILE
SC6502T.TXT	CPM6502/SC6502T	LIBRARY FILE
SCS502B.TXT	CPMS/SCBRANCH	(Relative branch routines)
SC6502M.TXT	CPM6502/SC6502M	LIBRARY FILE
1802		
SC1802.TXT	CPM1802/SC1802	MAIN ASSEMBLY FILE
SC1802T.TXT'	CPM1802/SC1802T	LIBRARY FILE
SC1802M.TXT	CPM1802/SC1802M	LIBRARY FILE

8080

SC8080.TXT	CPM8080/SC8080	MAIN ASSEMBLY FILE
SC8080T.TXT	CPM8080/SC8080T	LIBRARY FILE
SC8080M.TXT	CPM8080/SC8080M	LIBRARY FILE

Z80

SCZ80.TXT	CPMZ80/SCZ80	MAIN ASSEMBLY FILE
SCZ80T.TXT	CPMZ80/SCZ80T	LIBRARY FILP
SCZ80L.TXT	CPMZ80/SCZ80L	LIBRARY FILE
SCZ80U.TXT	CPMZ80/SCZ80U	LIBRARY FILE
SCZ80M.TXT	CPMZ80/SCZ80M	LIBRARY FILE

APPENDIX B

```
 NAM 6809 CPM MODULE for FLEX
 STTL VERSION 4.0  (c) 1983 By LLOYD I/O
 OPT PAG,EXP
 PAG


* (C) COPYRIGHT 1983, BY LLOYD I/O

* 6809 CPM MODULE FOR CRASMB
* FOR CRASMB VERSIONS 4.X

* WRITTEN BY

* FRANK L. HOFFMAN
* COPYRIGHT 1983
*
* LLOYD I/O
* 19535 NE GLISAN
* PORTLAND, OR 97230



 SETDP 0 DIRECT PAGE = ZERO

FRANK  EQU     1 FLEX VERSION FLAG

       LIB     SCCMACRO
       LIB     SCCZERO

* ++++++++++++++++++++++++++++++++++++++++++++++
*                                              +
*     START OF PROGRAM          'I6809'        +
*                                              +
* ++++++++++++++++++++++++++++++++++++++++++++++

       ORG     0

START  FDB     USER

       LBRA    INTERP  PROCESS MNEMONICS
       LBRA    INTINT  INITIALIZE
       LBRA    CLOSE   WRAP UP

TREES  JMP     [CPMTRE,U] DO MNEMONIC SEARCH
PERROR JMP     [CPMPER,U] PRINT ERROR BY NUMBER
OBS    JMP     [CPMOBS,U] SAVE OBJECT CODE
SKIP   JMP     [CPMSKI,U] SKIP PAST SPACES
OUTLIN JMP     [CPMOUT,U] GET NEXT CHARACTER
EXPRES JMP     [CPMEXP,U] EVALUATE AN EXPRESSION
```

```
NAMEO   FCC     /FLH/ INITIALS

        LIB     SC6809T
        LIB     SC6809S
        LIB     SC6809U
        LIB     SC6809B
        LIB     SCCITIME
        LIB     SC6809M

USER    EQU     *
        END
```

APPENDIX C

```
* FILE NAME 'T6809'

* THIS IS THE 6809
* MNEMONIC INTERPETER

* IT GENERATES THE
* OBJECT CODE IN 6809
* MACHINE CODE FROM
* THE STANDARD 6809
* MNEMONIC SET.

DPR EQU CPMSUF+0 DIRECT PAGE VALUE
DPF EQU CPMBUF+2 DIRECT PAGE INUSE FLAG
POST EQU  CPMBUF+3 POST BYTE INSTRUCTION
INDF EQU  CPMBUF+4 INDIRECT FLAG
NEGF EQU  CPMBUF+5 NEGATIVE FLAG
BITF EQU  CPMBUF+6 8 BIT OK IF EQUAL TO ZERO
STAF EQU  CPMBUF+7 STORE OR LOAD FLAG FOR MNEMONICS
PCRF EQU  CPMBUF+8 PROGRAM RELATIVE FLAG
COMF EQU  CPMBUF+9 COMMA, 'A' OR 'B' FOLLOWS FLAG
CCUF EQU  CPMBUF+10 USE 'ORCC' OR 'ANDCC'
HIF EQU CPMBUF+11 LONG BRANCH FLAG
BRATST EQU CPMBUF+12 BRANCH ERROR STATUS
FORCE EQU CPMBUF+14 FORCED EXTENDED OR DIRECT ADDRESSING


INTERP EQU
 LDA SFF
 STA FCMODE,U MNEMONIC MODE
 STA CCUF,U SET NO 'CC' ('OR' OR 'AND')
 LEAX MNEM2,PCR GET TABLE
 LBSR EXEC BINARY TREE SEARCH
 BEQ INT1 FOUND, SO PROCESS
 LEAX MNEM3,PCR
 LBSR EXEC
 BEQ INT1
 LEAX MNEM4,PCR
 LBSR EXEC
 BEQ INT1
 CLR A SET TO NON-ORGANIZED TABLES
 LEAX MSETDP,PCR 'SETDP' or 'REG'
 LBSR TREES DO SERIAL SEARCH
 BNE INT2
 ADJUST START GET REAL ADDRESS
 JSR O,X GO PROCESS IT
 CLR A
```

```
INT2 EQU
 RTS
INT1 EQU *
 CLR TYPE,U SET NO TYPE
 CLR ADACC,U NO ACCUMULATOR OPERATION
 ADJUST START GET REAL ADDRESS
 STX XX1,U SAVE ACTUAL ADDRESS
 LDX O,X GET ROUTINE ADDRESS
 ADJUST START GET REAL ADDRESS
 JSR 0,X EXECUTE IT
 CLRA
 RTS

EXEC EQU  EXECUTE SEARCH
 LDB O,X+ GET RECORD LENGTH
 STB ITLEG,U
 SUB B 3 CALCULATE MNEMONIC LENGTH
 STB BNAML,U
 LDA 0,X+ GET MNEMONIC COUNT
 LBRA TREES DO SEARCH

* INITIALIZE

INTINT EQU
 LDA $FF
 STA DPF,U SET DIRECT PAGE OFF
 LDX 0
 STX DPR,U SET ZERO DIRECT PAGE
 LBRA TIME

CLOSE EQU*
 RTS

*============================== *
.
.
.


* FILE NAME 'M6809'

* M6809 MNEMONIC LOOK-UP
* TABLES

* -------------------------

* INSTRUCTION TYPE ASSIGNMENT
```

```
* TYPE DESCRIPTION
*
*  1   INHERENT ONLY (1 BYTE)
*  2   BRANCH RELATIVE (2 BYTE)
*  3   LONG BRANCH RELATIVE (4 BYTE)
*  4   A or B by I,D,X,E
*  5   A, B, or D by I,D,X,E
*  6   A, B, or CC by I,D,X,E (CC by I only)
*  7   A, B, or M by D,X,E (A or B inherent only)
*  8   All Registers by I,D,X,E (except CC)
*  9   Register to Register (8 bit to 8 bit, 16 to 16 bit)
*  10  JSR, JMP D,X,E
*  11  ST BY D.E,X
*  12  CPX D,E,I,X
*  13  LD BY D,E,I,X
*  14  Indexed only (LEA)
*  15  STACK S OR U PUSH, OR PULL
*  16  SOFTWARE INTERRUPTS
*  17  DES,DEX,INX,INS
*  18  6800 TRANSFERS (TAB,TBA.... etc.)
*  19  CWAI ONLY (IMMEDIATE ADDRESSING MODE ONLY)
*  20  SAVE BYTES (1-255 BYTES) (TYPE 6)
*
* -------------------------

*  2 CHARACTER MNEMONICS

MNEM2 EQU * 3
  FCB  5 RECORD LENGTH
  FCB  3 RECORD COUNT
  FCC  /LD./
  FDB  CLD-START
  FCC  /OR./
  FDB  COR-START
  FCC  /ST./
  FDB  CST-START
  FCB  0 END MARKER

*  3 CHARACTER MNEMONICS

MNEM3 EQU
  FCB  6 RECORD LENGTH
  FCB  79 RECORD COUNT
  FCC  /ABA./
  FDB  CABA-START
  FCC  /ABX./
  FDB  CABX-START
  FCC  /ADC./
  FDB  CADC-START
  FCC  /ADD./
  FDB  CADD-START
```

```
  FCC /AND./
  FDB CAND-START
  FCC /ASL./
  FDB CASL-START
.
.
.
.
.
.
.
.
.
*  OP CODES

CABX FDB T1-START
  FCB  $3A

CADC FDB T4-START
  FCB  $99,$B9,$89,$A9
  FCB  $D9,$F9,$C9,$E9

CADD FDB T5-START
  FCB  $99,$99,$8B,$AB
  FCB  $DB,$FB,$CB,$EB
  FCB  $03,$F3,$C3,$E3

CAND FDB T6-START
  FCB  $94,$B4,$84,$A4
  FCB  $04,$F4,$C4,$E4
  FCB  $1C

CASL FDB T7-START
  FCB  $48,$58,$08,78,$68

CASR FDB T7-START
  FCB  $47,$57,$07,$77,$67

CBCC FD9 BRANCH-START
  FCB  $24

CBCS FDB BRANCH-START
  FCB  $25

CBEQ FDB BRANCH-START
  FCB  $27
```

APPENDIX D

```
 NAM 6809 CPM MODULE for OS9
 STTL VERSION 2.0 (c) 1983 By LLOYD I/O
 OPT PAG,EXP
 PAG

* This program ust be assembled by CRASMB or OSM.

* (C) COPYRIGHT 1983, BY LLOYD I/O

* 6809 CPM MODULE FOR CRASMB

* WRITTEN BY

* FRANK L. HCFFMAN
* COPYRIGHT 1983
*
* LLOYD I/O
* 19535 NE GLISAN STREET
* PORTLAND, OR 97230


 SETDP 0 DIRECT PAGE = ZERO

 OPT OS9

        USE     CPMS/SCCMACRO
        USE     CPMS/SCCZERO

*++++++++++++++++++++++++++++++++++++++++
*                                       +
*  START OF PROGRAM       'I6809'       +
*                                       +
*++++++++++++++++++++++++++++++++++++++++

        MOD     PGMLEN,NAMEO,$21,$81,START,0

START   EQU *
        LBRA    INTERP PROCESS MNEMONICS
        LBRA    INTINT INITIALIZE
        LBRA    CLOSE   WRAP UP

TREES   JMP     [CPMTRE,U]  DO MNEMONIC SEARCH
PERROR  JMP     [CPMPER,U]  PRINT ERROR BY NUMBER
OBS     JMP     [CPMOBS,U]  SAVE OBJECT CODE
SKIP    JMP     [CPMSKI,U]  SKIP PAST SPACES
CUTLIN  JMP     [CPMOUT,U]  GET NEXT CHARACTER
EXPRES  JMP     [CPMEXP,U]  EVALUATE AN EXPRESSION
```

```
NAMEO  FCS     /I6809/ PROGRAM NAME

       USE     CPM6809/SC6809T
       USE     CPM6809/SC6809S
       USE     CPM6809/SC6809U
       USE     CPM6809/SC6809B
       USE     CPMS/SCCOTIME
       USE     CPM6809/SC68098M

       EMOD

PGMLEN EQU *

       END
```

UPDATE NOTES
for
CRASMB (version 5.0)

NEW DIRECTIVES AND FUNCTIONS

SLL

Meaning: To start a chain of local labels which are keyed off a standard or global label. Local labels are any legal symbol or label but begin with a question mark ("?"). They are very useful within macros and library files. The syntax is:

```
label SLL
```

where 'label' is any global label. Local labels of the same name may be used within another bracketed local label area. They actually take up two different areas in memory when the program is being assembled. The SLL-ELL pair may appear any number of times in a program, but may only be nested up to 100 deep.

Examples:

```
   ESUE   SLL
          ... code ...
   ?SAY   EQU *   DEFINE A LOCAL LABEL AREA AS THE 'PC
          ... code ...
          BNE ?SAY access a local label in the operand
          BRA ?DONE  AGAIN
          ... code ...
   ?DONE  EQU *
          ELL
```

ELL

Meaning: This is the closing bracket for the current level of local labels in use. It must have a matching SLL directive. See the above example.

INI

Meaning: This directive will output the byte value of the expressions which follow it directly to the output device.

The syntax is:
INI expression 1, [expression 2, expression 3.....]

Examples:
```
INI 30     Turn compressed on for a Microline u83A
INI 29,0,0 Turn compressed off and send two nulls
```

Macro parameter length and substring function.

The length function (Syntax: &&macro number) replaces the three characters of the call with the decimal value of the actual length of the parameter specified. Either the current macro's parameters or the assembler command line parameters may be used.

The sub-string, mid-string function (syntax: &&[macro number, start expression,length expression]) returns a part of the parameter specified which may be the command line parameters or macro parameters. The two ampersands and opening and closing brackets are a part of the syntax. Both expressions are any legal expression, even containing a call to the length function.

Examples:
```
FCB &&(1,1,1)     return the first character of parameter 1
FCC '&&1'         form constant string of the length digits
FCC "&&[3,&&3-2,3] string of the last 3 characters
```

Both of these functions are very handy for breaking down parameters into the parts needed without having to use two parameters to do the same thing. It is possible to write cross-assembly macros which use the exact manufacture's mnemonic set and addressing mode syntax.

FLEX --- CONTROL Q.QUIT

The Flex version supports the 'quit and return to DOS' function. Typing 'Control Q' during Pass 1 or Pass 2 will make the assembler return to Flex. This is a built in function to OS9 and is not a part of the OS9 version since it is already supported.