Software Services

The 68XX Specialists for Application and System Software

SSB 6809 ASSEMBLER

We Sell Capabilities...

COPYRIGHT NOTICE

WARRANTEE INFORMATION

The SSB ASSEMBLER is provided AS IS without warrantee. Reasonable care has been taken to insure that the software operates as described in this manual. If you find a situation in which the assembler does not operate as described, please contact Smoke Signal Broadcasting. We will attempt to correct any errors brought to our attention, but we make no gaurantee to do so.

TABLE OF CONTENTS

INTRODUCTION

The SSB 6809 Assembler for DOS69 is a versatile and powerful assembler for the 6809 computer system. The assembler will permit very large source files (including multiple files) to be assembled, producing a binary file on disk. A number of assembly-time options are provided. The name of the source file(s) to be assembled, unless otherwise specified, default(s) to a filename bearing a type 1 extension. Likewise, binary output filenames will default to type 0 extensions. Since DOS69 initially defines a type 0 extension as .BIN and a type 1 extension as .TXT, extension types and their literal equivalents will be used interchangeably throughout the remainder of this manual.

In addition to all the standard 6809 mnemonics, the assembler will accept and translate all 6800 mnemonics to 6809 object code. This allows programs written for the 6800 to be moved to the 6809 with minimum of difficulty.

INVOKING THE ASSEMBLER

The assembler is invoked using the ASM09 command, which uses the general syntax of:

ASM09 <FILENAME> [,<FILENAME>...] [/<OPTION> [[,] <OPTION>...]]

The name of the file(s) to be assembled default(s) to the base name bearing a type 1 extension and the drive currently assigned as the work drive. If the "O=XXX" option (specifies object file filename) is not included in the command line, the binary file will be assigned the same 'name' as the first source file which is being assembled, but will be assigned a type 0 extension. ASM09 will allow up to 20 source files to be assembled at one time. The default filename extensions can be overiden by simply defining the extension explicitly. That is, if FILE.A is specified, then the file FILE.A will be operated upon rather than FILE.TXT or FILE.BIN. Refer to the DOS69 documentation for information regarding extension type definitions, work drive assignments, and the use of the SET command for defining such parameters.

COMMAND OPTION SWITCHES

There are many option switches which can be included in the option list. The option list is separated from the file specs by a '/'. Each option code is represented by a single letter, with multiple options specified as a list of unseparated option switches. Options of the form "i=xxx", if not the last option on the command line, must be seperated by a ",". This is to delimit the end of the string. Below is a list of the available options and their meanings. a "-" preceeding an option switch will result in that switch being turned off. (note that some switches default to "ON"...).

ASM COMMAND LINE OPTION SWITCHES

OPTION	DEFAULT	FUNCTION
A	-A	assemble into memory note: the assembler will not overlay itself
G	-G	print code generated by FCB, FDB, & FCC directives
H	-H	get initial heading from console
K	-K	use formfeeds (instead of linefeeds) to get to the top of the next page
L	-L	generate listing (may use L=XXX form) listing file defaults to XXX.LST where XXX is filename of 1st file in command line
M	-M	list error messages
N=ddd	N=78	set printer line length to ddd(dec.). 49 < ddd < 121
O	O	create object file. (may use O=XXX form) object file defaults to XXX.BIN where XXX is file-name of 1st file in command line.
P=ddd	P=66	set number of lines per printed page & enable paging
S	-S	print symbol table
T	-T	put system's time / date into page headings requires moniter to have a time / date routine. (see appendix E)
X	-X	print cross reference concordance

EXAMPLES

1. ASM09 TEST
2. ASM09 TEST/LS
3. ASM09 0:TEST/O=1:TEST.$,SL
4. ASM09 TEST/L-O
5. ASM09 PROG0.TXT,PROG1.TXT,PROG2.TXT
6. ASM09 1:EQU.SYS,PROG3.TXT,2:PROG4.TXT/0=PROG.$

The first example would assemble the source file TEST.TXT from the working drive and would check to see if TEST.BIN exists on the the same drive. If TEST.BIN was not found, then the assembler will create TEST.BIN. However, if TEST.BIN existed, the assembler would overwrite TEST.BIN with new object code. The second example would do the same thing as the first, except that listing would be enabled & the symbol table would be printed.

Example 3 would cause assembly of TEST.TXT (found on drive 0) and would create a "transient command" file TEST.$ (on drive 1).

The 4th example would cause file TEST.TXT (on the current working drive) to be assembled for a listing only.

Example 5 shows how to assemble several files together to form a single object file. Successful multiple assembly requires only the last input file to contain an 'END' directive. Note also that the same rules for multiple and undefined symbol detection apply between files as well as within files. In addition, if the name of the binary file is not specified, the binary file will take on the name of the first file given in the file specs. This example would place the binary output in a file named 'PROG0.BIN' on the working drive.

The command line of example 6 shows again how multiple files can be assembled, with input files being read from different drives, and a transient command file PROG.$ being the binary output file located on the working drive. Note here how an equate file, for example, can be included in the assembly process.

LISTINGS

To have the listing printed on a printer rather than appearing on the system terminal, simply precede the the ASM09 command with the "P," prefix (see 'P' command in the DOS69 description). This is assuming that the appropriate PRINT.SYS file exists in the operating system, and has been run prior to an attempt to use the printer. For example:

```
DOS: P ASM09 TEST/L
```

This would cause the assembly listing of the source file TEST.TXT to be printed on the system printer. Note that an option switch of "/L=," would route the listing to a disk file (due to the "="), and that this file would have the same filename as the first source file in the command (but with an extension of ".LST" (due to the "," in "/L=,")).

NOTE:

Listing output (whether to a file, the console, or to the printer) can be stopped, restarted or aborted by the standard control characters (see appendix E).

For a complete description of the format of the listing which will be produced by ASM09, please refer to appendix D.

ASSEMBLER DESCRIPTION

The SSB 6809 Assembler was written for maximum flexibility for DOS69 disk system users. As always, flexibility adds complexity and therefore the user is advised to read the following application notes before attempting to use the assembler.

It is assummed that the user is familiar with assembly language and, in particular, the mnemonics of the 6809 assembly language. Those who are not should refer to the "M6809 Microprocessor Programming Manual" or the "M6809 Programming Reference Manual", both available from your Motorola distributor.

The source language (input) for the SSB 6809 Assembler consists of a subset of the 7-bit ASCII (American Standard Code for Information Interchange, 1968) character set. In all cases the parity bit (most significant bit) of each character must be 0.

SOURCE STATEMENT FORMAT

Each source statement may include up to 5 fields: a sequence number, a label (or "*" for a comment line), an operation, an operand, and (optionally) a comment.

SEQUENCE NUMBERS

The sequence number field is an optional field provided as a programming convenience. The sequence number field starts at the beginning of the source line, and consists of up to five decimal digits. The value of the number must be less than 65536. Sequence numbers (if supplied) must be followed by a space. ASM09 will automatically supply sequence numbers if the first character of the first source line of the first file is not an ASCII numeric digit ("0" : "9").

Although sequence numbers are optional, they must be consistently used or not used for the entire assembly. If the first source statement has a sequence number, then every succeeding source statement must also have a sequence number. If the first source statement does not have a sequence number, then no other source statement may be numbered.

SOURCE STATEMENT FORMAT

(continued)

LABEL FIELD

The label field occurs as the first field of a source statement. The label field can take one of the following forms:

1. An asterisk (*) as the first character in the label field indicates that the rest of the source statement is comment. Comments are ignored by the Assembler, and are printed on the source listing only for the programmer's information.

2. A space as the first character indicates that the label field is empty. The line has no label and is not a comment.

3. A symbol chaacter as the first character indicates that the line has a label. Symbol characters are the upper case letters A-Z, digits 0-9, and the special characters, period (.), dollar sign ($), and underscore (_). Symbols consist of one to six characters, the first of which must be alphabetic or the special character, period (.). Certain special symbols are reserved by the Assembler, and will cause an error to be generated if they appear in a label field. These reserved symbols are: A, B, and X, CC, D, DP, PC, PCR, S, U, and Y.

   A symbol may occur only once in the label field unless it is used with the SET directive. If a symbol does occur more than once in a label field, then each reference to that symbol will be flagged with an error.

   With the exception of some directives, a label is assigned the value of the program counter of the first byte of the instruction or data being assembled.

   Each unique label, undefined symbol, and external reference symbol in a program is allocated a ten-byte block in the symbol table. In addition, a ten-byte block is allocated for every four references to a symbol, if the cross reference option is in effect.

SOURCE STATEMENT FORMAT

(continued)

OPERATION FIELD

The operation field occurs after the label field, and must be preceded by at least one space. The operation field must contain a symbol. Thus, the rules governing labels apply to the operation field as well. Entries in the operation field may be one of two types:

Opcode

These correspond directly to the machine instructions. The operation code includes the "A" or "B" character for the accumulator specification. For compatibility with other 6800 / 6809 assemblers, a single space may separate the operation code from the accumulator designator. For example, "LDA A" is the same as "LDA A" is the same as "LDAA" . Although ASM09 recognizes the above instruction forms, the propr form for the 6809 instruction "load accumulaor A" is "LDA".

Directive

These are special operation codes known to ASM09 which control the assembly process rather than being translated into machine instructions. (see section entitled "ASSEMBLER DIRECTIVES")

SOURCE STATEMENT FORMAT

(continued)

OPERAND FIELD

The operand field's interpretation is dependant on the contents of the operation field. The operand field, if required, must follow the operation field, and must be preceded by at least one space. The operand field may contain a symbol, expression, or a combination of symbols and expressions separated by commas. The operand field of machine instuctions is used to specify the addressing mode of the instruction, as well as the operand of the instruction.

The following operand formats exist:

Operand Format	6809 Addressing Mode
no operand	accumulator and inherent
<expression>	direct, extended, or relative
#<expression>	immediate
<expression>,X	indexed
<<expression>	direct
><expression>	extended
[<expression>]	extended indirect
<expression>,R	indexed
<<expression>,R	8 bit offset indexed
><expression>,R	16 bit offset indexed
[<expression>,R]	indexed indirect
<[<expression>,R]	8 bit offset indirect
>[<expression>,R]	16 bit offset indirect
Q+	auto increment by 1
Q++	auto incremet by 2
[Q++]	auto increment indirect (by 2)
-Q	auto decrement by 1
--Q	auto decrement by 2
[--Q]	auto decrement indirect (by 2)
W1[,W2,...Wn]	immediate

Where R is one o the registers PCR, S, U, X, or Y, and Q is one of the registers S, U, X, or Y. Wi (i=1 to n) is one of the symbols A, B, CC, D, DP, PC, S, U, X, or Y.

6809 ADDRESSING MODES

The 6809 includes some instructions which require no operands. These instructions are self-contained, and employ the inherent addressing or the accumulator addressing mode. For more information, refer to appendix B.

IMMEDIATE ADDRESSING

Immediate addresssing refers to the use of one or two bytes of information that immediately follow the operation code in memory. Immediate addressing is indicated by preceding the operand field with the pound sign or number sign (#) e.g., #<expression>. The expression following the "#" will be assigned one or two bytes of storage, depending on the instruction. All instructions referencing the accumulator "A" or "B", or the condition code register "CC", will generate a one-byte immediate value. Also, immediate addressing used with the PSHS, PULS, PSHU, and PULU instructions generates a one-byte immediate value. Immediate operands used in all other instructions generate a two-byte value.

The register list operand does not take the form #<expression> but still generates one byte of immediate data. The form of the operand is:

```
R1 [,R2,...,Rn]
```

where Ri (i-1 to n) is one of the symbols A, B, CC, D, DP, PC, S, U, X or Y. The number and type of symbols vary, depending on the specific instruction.

For the instructions PSHS, PULS, PSHU, and PULU, any of the above register names may be included in the register list. The only restriction is that "U" cannot be specified with PSHU or PULU, and "S" cannot be specified with PSHS or PULS. The one-byte immediate value assigned to the operand is determine by the registers specified. Each register name sets a bit in the immediate byte as follows:

```
PC  U,S  X   Y   DP  B,D A,D  C   <--- register(s)
 7   6   5   4   3   2   1   0  <--- bit set
```

For the instructions EXG and TFR, exactly two of the above register names must be included in the register list. The other restriction is the size of the registers specified. For the EXG instruction, the two registers must be the same size. For the TFR instruction, the two registers must be the same size, or the first can be a 16-bit register and the second an 8-bit register. In the case where the transfer is from a 16-bit register to an 8-bit register, the least significant 8 bits are transferred. The 8-bit registers are A, B, CC, and DP. The 16-bit

registers are D, PC, S, U, X, and Y. The one-byte immediate value assigned to the operand is determined by the register names. The most significant four bits of the immediate byte contain the value of the first register name; the least significant four bits contain the value of the second register as shown by the following table:

Register specified	value (hex)	(binary)	size 8b	16b
"D"	0	0000		*
"X"	1	0001		*
"Y"	2	0010		*
"U"	3	0011		*
"S"	4	0100		*
"PC"	5	0101		*
"A"	8	1000	*	
"B"	9	1001	*	
"CC"	A	1010	*	
"DP"	B	1011	*	

RELATIVE ADDRESSING

Relative addressing is used by branch instructions. There are two forms of the branch instruction. The short branch can only be executed within the range -126 to +129 bytes relative to the first byte of the branch instruction. The actual branch offset is put into the second byte of the branch instruction. The long branch can execute in the full range of addressing from 0000-FFFF (hexadecimal) because a two-byte offset is calculated and put into the operand field of the branch instruction. The offset is the two's complement of the difference between the location of the byte immediately following the branch instruction and the location of the destination of the branch.

DIRECT AND EXTENDED ADDRESSSING

Direct and extended addresssing utilize one (direct) or two (extended) bytes to contain the address of the operand. Direct and extended addressing are indicated by having only an expression in the operand field (i.e., <expression>). Direct addressing will be used by ASM09 whenever possible. Regardless of the criteria described above, it is possible to force the Assembler to use the direct addressing mode by preceding the operand with the "<" character. Similarly extended addresssing can be forced by preceding the operand with the ">" character. These two operand forms are: <<expression> and ><expression>. There is no restriction on the latter form. It will always generate extended addressing. If direct addressing is forced, the most significant byte of the expression is compared with the direct page pseudo register. If they are not the same, a warning message is generated. The user must ensure that the direct page register is set up at execution time.

INDEXED ADDRESSING

Indexed addressing is relative to one of the index registers. The general form is <expression>,R. The address is calculated at the time of instruction execution by adding the value of <expression> to the current contents of the index register. The other general form is [<expression>,R]. In this indirect form, the address is calculated at the time of instruction execution by first adding the value of <expression> to the current contents of the index register, and then retrieving the two bytes from the calculated address and address+1. This two-byte value is used as the effective address of the operand. The allowable forms of indexed addressing are described below. Appendix B describes the format of the post-byte (i.e., the byte immediately following the opcode) for each of the indexed addressing modes.
In the description below, R refers to one of the index registers S, U, X, or Y.

The accumulator offset mode allows one of the accumulators to be specified instead of an <expression>. Valid forms are:

<acc>,R and [,<acc>,R]

where <acc> is one of the accumulators A, B, or D. This form generates a one-byte operand (post-byte only). When accumulator A or B is specified, sign extension occurs prior to adding the value in the accumulator to the index register. The valid forms for the automatic increment/decrement mode are shown below. For each row, the three entries shown are equivalent.

R+	,R+	0,R+
-R	,-R	0,-R
R++	,R++	0,R++
--R	,--R	0,--R
[R++]	[,R++]	[0,--R]

In this form, the only valid expression is 0. Like the accumulator offset mode, this form generates a one-byte operand (post-byte only). The valid forms for the expression offset mode are:

R	,R	<expression>,R
[R]	[,R]	[<expression>,R]
<R	<,R	<<expression>,R
<[R]	<[,R]	<[<expression>,R]
<R	>,R	><expression>,R
>[R]	>[,R]	>[<expression>,R]

The "<" and ">" characters force an 8-bit or 16-bit offset, respectively,and are described below. If no expression is specified, only the post byte of the operand is generated. If an expression with a value in the range -16 to +15 is specified without indirection, a one-byte

operand is generated which contains the expression's value as well as the index register indicator. At execution time, the expression's value is expanded to 16 bits with sign extension before being added to the index register. All other forms will generate a post-byte, as well as either a one-byte or two-byte offset which contains the value of the expression. The size of the offset is determined by the type and size of the expression. Expressions with values in the range -128 to +127 generate an 8-bit offset. If an expression that follows the above rules contains a symbol that is referenced before it has been defined, the instruction will be assembled using a 16-bit offset in order to avoid phasing errors. All other cases will result in a 16-bit offset being generated. In the case where an 8-bit offset is generated, the value is expanded to 16 bits with sign extension at execution time. Regardless of the criteria described above, it is possible to force the Assembler to generate an 8-bit offset by preceding the operand with the "<" character. Similarly, a 16-bit offset can be forced by preceding the operand with the ">" character. There is no restriction on the ">"form. It always generates a post-byte followed by a 16-bit offset. If an 8-bit offset is forced and the expression has a value outside of the range -128 to +127, a byte overflow error is generated.

The valid forms for the program counter relative mode are exactly the same as the expression offset mode, with the exception that the index register specificaion must be "PCR". However, the manner in which the offset is generated by ASM09 differs. The assembler generates a relative address which is then used as the 8-bit or 16-bit offset following the post-byte. The relative address is the two's complement of the difference between the location of the byte immediately following the indexed instruction and the value of the expression. If the relative address calculated is not in the range -128 to +127, or if the expression references a symbol that has not yet been defined, a two-byte offset is generated after the post-byte. A one byte offset is generated if the relative address is in the range -128 to +127. Like the expression offset mode, a one-byte offset can be forced by preceding the operand with a "<". A ">" forces a two-byte offset. A byte overflow error is generated if a one-byte offset is forced when the relative address is not in the range -128 to +127.

EXTENDED INDIRECT ADDRESSING

The extended indirect mode has the form:

[<expression>]

Although extended indirect is a logical extension of the extended addressing mode, this mode is implemented using an encoding of the post-byte under the indexed addressing mode. A post-byte is generated, as well as a two-byte offset which contains the value of the expression.

EXPRESSIONS

An expression is a combination of symbols, constants, algebraic operators, and parentheses. The expression is used to specify a value which is to be used as an operand. Expressions follow the conventional rules of algebra.

OPERATORS

The precedence of the various operators is as follows. Paranthetical expressions are evaluated first, with the innermost parentheses being processed before the outer ones. Next, the multiplication (*), division (/), and all two-character operators have precedence. Of lowest precedence are the addition (+) and subtraction (-) operators. Unary minus can only occur at the beginning of an expression or immediately before a left parenthesis. Unary minus is equivalent in evaluation to putting a zero directly before the minus sign. For example the following expressions are all equivalent:

```
-TAG*INDEX+3
0-TAG1*INDEX+3
-(TAG1*INDEX)+3
```

Operators of the same precedence are evaluated from left to right. All intermediate results in the computation of an expresssion are truncated to a 16-bit integer value. The result of an expression is also a 16-bit integer. Operators can operate on numeric constants, single character ASCII literals, and symbols. In addition to the normal operators for multiplication, division, addition, and subtraction, ASM09 recognizes certain two character operators. The operators are infix operators and have the same precedence as multiplicaion or division. Each two-character operator begins with an exclamation point (!) and takes two operands. The following two-character operators are defined:

!^ - exponentiation

The left operand is raised to the power specified by the right operand. If the right operand is zero, the resulting value will be "1", regardless of the value of the left operand.

!. - logical AND

Each bit in the left operand is logically "ANDed" with the corresponding bit in the right operand.

!+ - inclusive OR

Each bit in the left operand is inclusively "ORed" with the corresponding bit in the right operand.

!X - exclusive OR

Each bit in the left operand is exclusively "ORed" with the corresponding bit in the right operand.

!< - shift left

The left operand is shifted to the left by the number of bits specified by the right operand. The left operand is zero-filled from the right.

!> - shift right

The left operand is shifted to the right by the numer of bits specified by the right operand. The left operand is zero-filled from the left.

!L - rotate left

The left operand is rotated left by the number of bits specified by the right operand. The most significant bit is rotated into the least significant bit position of the left operand.

!R - rotate right

The left operand is rotated right by the number of bits specified by the right operand. The least significant bit is rotated into the most significant bit position of the left operand.

SYMBOLS

Each symbol is associated with a 16 bit integer value which is used in place of the symbol during the expresssion evaluation. The asterisk (*) used in an expression as a symbol represents the current value of the location counter (the first byte of a multi-byte instruction).

CONSTANTS

Constants represent quantities of data that do not vary in value during the execution of a program. The numeric constants can be in one of four bases: decimal, hexadecimal, binary, or octal.

A decimal constant consists of a string of numeric digits. The value of a decimal constant must fall in the range 0-65535, inclusive. Optionally, decimal constants may be preceded by the ampersand character (&). The following example shows both valid and invalid decimal constants:

VALID	INVALID	REASON INVALID
12	123456	more than 5 digits
12345	12.3	invalid character
65201	67800	out of range (> 65535)

A hexadecimal constant consists of a maximum of four characters from the set of digits (0-9) and the upper case alphabetic letters (A-F), and is preceded by a dollar sign ($). Hexadecimal constants can also be designated by being succeded by the letter "H". In this case, the first digit of the hexadecimal constant must be a numeric so that the constant can be distinguished from a symbol name. Hexadecimal constants must be in the range $0000 to $FFFF. The following example shows both valid and invalid hexadecimal constants:

VALID	INVALID	REASON INVALID
$12	ABCD	no preceding "$"
0ABCDH	$G2A	invalid character
$001F	$2F018	too many digits

A binary constant consists of a maximum of 16 ones or zeros preceded by a percent sign (%). Binary constants can also be represented by a series of ones and zeros succeeded by the letter "B". The following examle shows both valid and invalid binary constants:

VALID	INVALID	REASON INVALID
%00101	1010101	missing percent
%1	%10011000101010111	too many digits
10100B	%210101	invalid digit

An octal constant consists of a maximum of six numeric digits, excluding the digits 8 and 9, preceded by a commercial at-sign (@). Octal constants can also be designated by ending in the letter "O" or "Q". Octal constants must be in the the ranges @0 to @177777. The following example shows both valid and invalid octal constants:

VALID	INVALID	REASON INVAID
@17634	@2317234	too many digits
377Q	@277272	out of range
1776000	23914Q	invalid character

Character constants can be used in expressions if they are single characters. Character constants are preceded by a single quote. Any character, including the single quote, can be used as a character constant. The following example shows both valid and invalid character constants:

VALID	INVALID	REASON INVALID
'*	'VALID	too long

COMMENT FIELD

The last field of an ASM09 source statement is the comment field. This field is optional and is only printed on the source listing for documentation purposes. The comment field is separated from the operand field (or from the operation field if no operand is required)

AUTO FIELDING

ASM09 performs automatic output fielding. No matter what the source file looks like in terms of field spacing, the output will automatically tab each field into a columnar form.

ASSEMBLER DIRECTIVES

In addition to the 6809 mnemonics, this assembler supports 16 assembler directives or pseudo-ops. These pseudo-ops are listed below along with a brief description. More detailed descriptions follow:

FCC	form constant character
FCB	form constant byte
FDB	form double byte
SPC	insert spaces in output listing
OPT	activates or deactivates assembler options
PAGE	skip to next page of output
ORG	define new origin (PC)
EQU	assign value to symbol
END	signal end of source program
NAM	specify name
TTL	specify title
RMB	reserve memory bytes
BSZ	reserve block of storage (fill with $00's)
SETDP	set direct-page pseudo register
REG	define register list
SET	assign temporary value to a symbol

FCC

The function of FCC is to create character strings for messages or tables. The character string 'text' is broken down to ASCII, one character per byte. The two allowable formats are shown below.

```
    label FCC  count,text
or
    label FCC  delimiter text same delimiter
```

where count is any decimal number. In the case where a number is used as a delimiter, the first character of text must not be a comma. The character limit of any FCC statement is 255. The use of label is optional.

FCB

The FCB pseudo-op caused an axpression to be evaluated and the resultant 8 bits placed in memory. Usage is shown below:

```
label FCB  expression 1,expression 2,...expression N
```

Each expression is separated by a comma with a maximum of 255 expressions per FCB statement. The label is optional.

FDB

The function of the FDB directive is identical to the FCB except 16 bit quantities are assembled, i.e., two bytes generated for each expression. The required format is shown below:

```
label FDB  expression 1,expression 2,...expression N
```

where the label is optional.

SPC

The SPC operator causes the specified number of spaces to be inserted in the output listing. The format is shown below:

```
SPC  expression
```

Notice that no label is allowed. If 'expression' evaluates to zero, one space is inserted. The operator SPC itself does not appear in the output listing. If PAGE mode is selected, SPC will not cause spacing past the top of the next page.

OPT

The directive OPT is used to activate or deactivate the assembler options from within the source program. The format is shown below. Note that no label is allowed, and no code is generated.

```
OPT  option1,option2,...optionN
```

The allowable options are:

SYM print sorted symbol table after listing.
NOS do not print the symbol table (default).

GEN print code generated by FCB, FDB, and FCC.
NOG print only one line for each FCB, FDB, or FCC (default).

LIS print the assembled source listing. (if "/L" in command)
NOL supress the printing of the assembly listing (default).

PAG enable page formatting and numbering (default).
NOP disable page mode.

CRE enable printing of cross-reference concordance at end of listing. This option must be specified before the first symbol is encountered in the source program.

LLE=ddd set printer line length (defaults to 78, max is 120).

P=ddd set printer page length (defaults to 66, max is 128).

WAR enable printing of warning messages (default).
NOW suppress warnings

OBJ enable writing of object module (default)
NOO disable writing of object module

If contradicting options appear, the last one appearing takes precedence. All options take effect simultaneously at the beginning of pass 2. The default options specified take effect unless the user specifies a particular option. Only the first 3 characters of an option name are significant and multiple options are separated by a comma.

PAGE

The PAGE operator, if the PAGE option is on, causes a page eject and subsequently causes the title (if any) and page number to be printed at the top of the next page. No label is allowed and no code is produced.

ORG

The ORG operator, whose format is shown below, causes a new origin address (PC) for the code following.

```
ORG  expression
```

No label is allowed and no code is produced. If no ORG appears an origin of 0000 is assumed

EQU

EQU is used to equate a symbol to an expression as shown below. A label is required and no code is generated. Only one level of forward referencing is allowed and the equate must not be recursive.

```
label  EQU  expression
```

No code is produced by EQU.

END

This operator signals the assembler that the end of the source input has occured. No label is allowed and no code is generated.

A second use of the END statement allows for the assignment of a transfer address to the binary file created. This can be accomplished by putting a label or value in the 'operand field' of the END statement. As an example, suppose the program you are assembling is to start executing at location $100, and in the source file you have the label START on the statement which is ORGed at $100. The END statement should now include this label in its operand field in order to assign $100 as the transfer address, as shown on the next page.

```
        ORG $100
START   LDX $007D

        program here

        END START
```

or

```
        END $0100
```

NAM

The NAM directive inserts a module name into the heading line. It has no effect on the object code.

TTL

This operator is used to assign a title to be printed at the top of all pages if the PAGE option is on. If the PAGE option is off, this operator has no effect. The format, as shown below allows up to 45 characters in the title. No label is allowed.

```
        TTL   text for the title
```

No code is generated. If more than one NAM or TTL operator appears the last one encountered will be printed on the next page.

RMB

This operator causes the assembler to reserve memory for data storage. No code is generated, therefore the contents of the reserved memory locations are undefined at run time. The label is optional as shown below:

```
     label RMB  expression
```

where 'expression' is a 16 bit quantity.

BSZ

This directive causes ASM09 to reserve a block of memory whose lengtn is given by the operand field. All bytes thus reserved will be set to $00.

SETDP

This directive causes ASM09 to set an internal psuedo register to the value of the low order 8 bits of the expression in the operand field. If the high order 8 bits are not all zeroes, a byte overflow error will occur. If the high order 8 bits of any memory reference match the contents of this register, then ASM09 will assemble that instruction using the direct addressing mode (unless the ">" is used, in which case the extended addressing mode will be used). If a forward reference is made to a symbol (e.g. RMB's at the end of the program), ASM09 will assemble that instruction using the extended addressing mode (unless the "<" is used, in which case the direct addressing mode will be used).

NOTE:

If direct page psudeo register does not agree with the high order byte of the forced direct address, an addressing error will occur.

It is up to the programmer to insure that the DP register is set up appropriatly at execution time to correctly reach the desired memory region.

REG

This directive translates a list of register names into a mask which may be used in the post-byte of certain instructions. The operand field must contain a non-duplicated list of symbols of the set A, B, CC, D, DP, PC, S, U, X, OR Y. An error will occur if both U and S are specified at the same time.

SET

This directive will assign a value to a symbol. Its function is the same as "EQU", however, a symbol may be redefined after being "SET".

ASSEMBLER OPERATION

Pass 1

The first pass is used to build the symbol table which is used to resolve forward references.

Pass 2

During the second pass, several things may happen. If the LIST option is on, the assembled source listing is printed with error messages, if any. If the LIS option is off, only offending source lines and their corresponding error messages are printed. If the SYM option is on, a symbol table will be printed at the end of pass 2. If the CREF option is in effect, this symbol table will contain all of the symbols' definition & reference points. Note that this will eat a lot of paper -- you won't often need a cross reference concordance...

APPENDIX A

(CHARACTER SET)

The character set recognized by ASM09 is a subset of ASCII. The ASCII code is shown in the following figure. The following characters are recognized by ASM09:

1. The upper case letters A thru Z
2. The digits 0 thru 9.
3. Four arithmetic operators: +, -, *, & /.
4. The special 2-character expression operators:
   !^, !>, !<, !X, !., !+, !R, & !L
5. Parentheses in expression: (&).
6. The special symbol characters: underscore (_), period (.), and dollar sign ($). Only the period may be used as the first character of a symbol.
7. The characters used as prefixes for constants and addressing modes:
   - # immediate addressing
   - $ hexadecimal constant
   - & decimal constant
   - @ octal constant
   - % binary constant
   - ' character constant
8. The characters used as suffixes for constants and addressing modes:
   - ,X indexed addressing
   - H hexadecimal constant
   - O octal constant
   - Q octal constant
   - . decimal constant
   - % binary constant
   - ' character constant
9. Three seperator characters: space,carriage return, & comma.
10. The character "*" to indicate comments. Comments may contain any printable character.
11. The special symbols "A" & "B" to denote the accumulators, "*" to indicate the value of the location counter, "PCR","S","U","X", & "Y" to indicate indexed addressing in the operand field, "D" to specify the 16-bit double accumulator, and "CC" & "DP" to specify the control registers.
12. The special characters "[", & "]" used to indicate indirect addressing, "<" to force direct addressing or to force 8 bit offset mode, ">" to force extended addressing or to force 16 bit offset mode for indexed addresses, and "+" & "-" to indicate auto increment / decrement.

ASCII CODE

	2	3	4	5	6	7
Ø	sp	Ø	@	P	`	p
1	!	1	A	Q	a	q
2	"	2	B	R	b	r
3	#	3	C	S	c	s
4	$	4	D	T	d	t
5	%	5	E	U	e	u
6	&	6	F	V	f	v
7	'	7	G	W	g	w
8	(	8	H	X	h	x
9	)	9	I	Y	i	y
A	*	:	J	Z	j	z
B	+	;	K	[	k	[
	,	<	L	\	l	
D	-	=	M	]	m	
E	.	>	N	^	n	~
F	/	?	O	_	o	del

APPENDIX B

(6809 INSTRUCTIONS)

The following table lists the special symbols used in the description of the 6809 instruction set:

=	left side of equal sign is replaced by right side
[]	evaluate contents first; grouping
()	the contents of
M()	the contents of the memory location addressed by the enclosed expression
+	arithmetic addition
-	arithmetic subtraction
*	arithmetic multiplication
and	boolean and
effad	6809 effective address
or	boolean inclusive or
xor	boolean exclusive or
L>	logical shift right by number of bits specified
L<	logical shift left by number of bits specified
A>	arithmetic shift right by number of bits specified
A<	arithmetic shift left by number of bits specified
R>	rotate right by number of bits specified
R<	rotate left by number of bits specified
$nn	the hexadecimal number nn
n	a bit value of n (0 or 1)
nn	an 8 bit value of nn (00 : $FF)
nnnn	a 16 bit value of nnnn (0000 : $FFFF)
aa	8 bit address
aaaa	16 bit address
A	accumulator A
B	accumulator B
C	carry condition code (bit 0 of CC)
CC	condition code register
D	16 bit dual acumulator A,B
F	fast interrupt condition code (bit 6 of CC)
H	half carry condition code (bit 5 of CC)
I	interrupt condition code (bit 4 of CC)
ii	8 bit immediate operand
iiii	16 bit immediate operand
N	sign condition code (bit 3 of CC)
P	program counter
rl	register list
rr	8 bit relative branch address
rrrr	16 bit relative branch address
S	stack pointer
U	user stack pointer
V	overflow condition code (bit 1 of CC)
X	index register
xx	8 bit indexed addressing offset
xxop	opcode depends on index mode
Y	index register
Z	zero condition code (bit 2 of CC)

(6809 8 bit arithmetic instructions)

Mnemonic	Operand	Opcode	Function	Status F H I N Z V C
ABX	--	3A	X=(X)+(B)	- - - - - - -
ADCA	ii	89	A=(A)+ii+(C)	- T - T T T T
	aa	99	A=(A)+m(aa)+(C)	
	xxop	A9	A=(A)+xxop+(C)	
	aaaa	B9	A=(A)+M(aaaa)+(C)	
ADCB	ii	C9	B=(B)+ii+(C)	- T - T T T T
	aa	D9	A=(A)+M(aa)	
	xxop	E9	B=(B)+xxop	
	aaaa	F9	B=(B)+M(aaaa)+(C)	
ADDA	ii	8B	A=(A)+ii	- T - T T T T
	aa	9B	A=(A)+M(aa)	
	xxop	AB	A=(A)+xxop	
	aaaa	BB	A=(A)+M(aaaa)	
ADDB	ii	CB	B=(B)+ii	- T - T T T T
	aa	DB	B=(B)+M(aa)	
	xxop	EB	B=(B)+xxop	
	aaaa	FB	B=(B)+M(aaaa)	
ADDD	iiii	C3	D=(D)+iiii	- - - T T T T
	aa	D3	D=(D)+M(aa:aa+1)	
	xxop	E3	D=(D)+xxop	
	aaaa	F3	D=(D)+M(aaaa:aaaa+1)	
DAA	--	19	binary to BCD conv.	- - - T T T T
MUL	--	3D	D=(A) * (B)	- - - - T - T
SBCA	ii	82	A=(A)-ii-(C)	- - - T T T T
	aa	92	A=(A)-M(aa)-(C)	
	xxop	A2	A=(A)-xxop-(C)	
	aaaa	B2	A=(A)-M(aaaa)-(C)	
SBCB	ii	C2	B=(B)-ii-(C)	- - - T T T T
	aa	D2	B=(B)-M(aa)-(C)	
	xxop	E2	B=(B)-xxop-(C)	
	aaaa	F2	B=(B)-M(aaaa)-(C)	
SEX	--	1D	Sign extension of B into A	- - - T T Ø -
SUBA	ii	80	A=(A)-ii	- ? - T T T T
	aa	90	A=(A)-M(aa)	
	xxop	A0	A=(A)-xxop	
	aaaa	B0	A=(A)-M(aaaa)	
SUBB	ii	C0	B=(B)-ii	- ? - T T T T
	aa	D0	B=(B)-M(aa)	
	xxop	E0	B=(B)-xxop	
	aaaa	F0	B=(B)-M(aaaa)	
SUBD	iiii	83	D=(D)-iiii	- - - T T T T
	aa	93	D=(D)-M(aa:aa+1)	
	xxop	A3	D=(D)-xxop	
	aaaa	B3	D=(D)-M(aaaa:aaaa+1)	
NEG	aa	00	M(aa)=Ø-M(aa)	- ? - T T ? T
	xxop	60	xxop=Ø-xxop	
	aaaa	70	M(aaaa)=Ø-M(aaaa)	
NEGA	--	40	A=Ø-(A)	- ? - T T ? T
NEGB	--	50	B=Ø-(B)	- ? - T T ? T

(6809 logical instructions)

Mnemonic	Operand	Opcode	Function	Status F H I N Z V C
ANDA	ii	84	A=(A) and ii	- - - T T 0 -
	aa	94	A=(A) and M(aa)	
	xxop	A4	A=(A) and xxop	
	aaaa	B4	A=(A) and M(aaaa)	
ANDB	ii	C4	B=(B) and ii	- - - T T 0 -
	aa	D4	B=(B) and M(aa)	
	xxop	E4	B=(B) and xxop	
	aaaa	F4	B=(B) and M(aaaa)	
ANDCC	ii	1C	CC=(CC) and ii	? ? ? ? ? ?
ASL	aa	08	M(aa)=M(aa) A< 1	- ? - T T ?
	xxop	68	xxop=xxop A< 1	
	aaaa	78	M(aaaa)=M(aaaa) A< 1	
ASLA	--	48	A=(A) A< 1	- ? - T T ?
ASLB	--	58	B=(B) A< 1	- ? - T T ?
ASR	aa	07	M(aa)=M(aa) A> 1	- ? - T T ?
	xxop	67	xxop=xxop A> 1	
	aaaa	77	M(aaaa)=M(aaaa) a> 1	
ASRA	--	47	A=(A) A> 1	- ? - T T ?
ASRB	--	57	B=(B) A> 1	- ? - T T ?
BITA	ii	85	(A) and ii	- - - T T O
	aa	95	(A) and M(aa)	
	xxop	A5	(A) and xxop	
	aaaa	B5	(A) and M(aaaa)	
BITB	ii	C5	(B) and ii	- - - T T O
	aa	D5	(B) and M(aa)	
	xxop	E5	(B) and xxop	
	aaaa	F5	(B) and M(aaaa)	
CLR	aa	0F	M(aa)=0	- - - 0 1 0
	xxop	6F	xxop=0	
	aaaa	7F	M(aaaa)=0	
CLRA	--	4F	A=0	- - - 0 1 0
CLRB	--	5F	B=0	- - - 0 1 0
COM	aa	03	M(aa)=M(aa) xor $FF	- - - T T 0
	xxop	63	xxop=xxop xor $FF	
	aaaa	73	M(aaaa)=M(aaaa) xor $FF	
COMA	--	43	A=(A) xor $FF	- - - T T 0
COMB	--	53	B=(B) xor $FF	- - - T T 0
EORA	ii	88	A=(A) xor ii	- - - T T 0
	aa	98	A=(A) xor M(aa)	
	xxop	A8	A=(A) xor xxop	
	aaaa	B8	A=(A) xor M(aaaa)	
EORB	ii	C8	B=(B) xor ii	- - - T T 0
	aa	D8	B=(B) xor M(aa)	
	xxop	E8	B=(B) xor xxop	
	aaaa	F8	B=(B) xor M(aaaa)	

(6809 logical instructions)

(continued)

Mnemonic	Operand	Opcode	Function	Status F H I N Z V C
ORA	ii	8A	A=(A) or ii	- - - T T Ø -
	aa	9A	A=(A) or M(aa)	
	xxop	AA	A=(A) or xxop	
	aaaa	BA	A=(A) or M(aaaa)	
CRB	ii	CA	B=(B) or ii	- - - T T Ø -
	aa	DA	B=(B) or M(aa)	
	xxop	EA	B=(B) or xxop	
	aaaa	FA	B=(B) or m(aaaa)	
ORCC	ii	1A	CC=(CC) or ii	? ? ? ? ? ? ?

(6809 compare / test instructions)

Mnemonic	Operand	Opcode	Function	Status F	H	I	N	Z	V	C
CMPA	ii	81	(A)-ii	-	?	-	T	T	T	T
	aa	91	(A)-M(aa)							
	xxop	A1	(A)-xxop							
	aaaa	B1	(A)-M(aaaa)							
CMPB	ii	C1	(B)-ii	-	?	-	T	T	T	T
	aa	D1	(B)-M(aa)							
	xxop	E1	(B)-xxop							
	aaaa	F1	(B)-M(aaaa)							
CMPD	iiii	1083	(D)-iiii	-	-	-	T	T	T	T
	aa	1093	(D)-M(aa:aa+1)							
	xxop	10A3	(D)-xxop							
	aaaa	10B3	(D)-M(aaaa:aaaa+1)							
CMPS	iiii	118C	(S)-iiii	-	-	-	T	T	T	T
	aa	119C	(S)-M(aa:aa+1)							
	xxop	11AC	(S)-xxop							
	aaaa	11BC	(S)-M(aaaa:aaaa+1)							
CMPU	iiii	1183	(U)-iiii	-	-	-	T	T	T	T
	aa	1193	(U)-M(aa:aa+1)							
	xxop	11A3	(U)-xxop							
	aaaa	11B3	(U)-M(aaaa:aaaa+1)							
CMPX	iiii	8C	(X)-iiii	-	-	-	T	T	T	T
	aa	9C	(X)-M(aa:aa+1)							
	xxop	AC	(X)-xxop							
	aaaa	BC	(X)-M(aaaa:aaaa+1)							
CMPY	iiii	108C	(Y)-iiii	-	-	-	T	T	T	T
	aa	109C	(Y)-M(aa:aa+1)							
	xxop	10AC	(Y)-xxop							
	aaaa	10BC	(Y)-M(aaaa:aaaa+1)							
TST	aa	0D	M(aa)-00	-	-	-	T	T	0	-
	xxop	6D	xxop-00							
	aaaa	7D	M(aaaa)-00							
TSTA	--	4D	(A)-00	-	-	-	T	T	0	0
TSTB	--	5D	(B)-00	-	-	-	T	T	0	0

(6809 load / pull instructions)

Mnemonic	Operand	Opcode	Function	Status F H I N Z V C
EXG	rl	1E	exchange 2 registers	? ? ? ? ? ? ?
TFR	rl	1F	transfer register	? ? ? ? ? ? ?
LDA	ii	86	A=ii	- - - T T Ø -
	aa	96	A=M(aa)	
	xxop	A6	A=xxop	
	aaaa	B6	A=M(aaaa)	
LDB	ii	C6	B=ii	- - - T T Ø -
	aa	D6	B=M(aa)	
	xxop	E6	B=xxop	
	aaaa	F6	B=M(aaaa)	
LDD	iiii	CC	D=iiii	- - - T T Ø -
	aa	DC	D=M(aa:aa+1)	
	xxop	EC	D=xxop	
	aaaa	FC	D=M(aaaa:aaaa+1)	
LDS	iiii	10CE	S=iiii	- - - T T Ø -
	aa	10DE	S=M(aa:aa+1)	
	xxop	10EE	S=xxop	
	aaaa	10FE	S=M(aaaa:aaaa+1)	
LDU	iiii	CE	U=iiii	- - - T T Ø -
	aa	DE	U=M(aa,aa+1)	
	xxop	EE	U=xxop	
	aaaa	FE	U=M(aaaa:aaaa+1)	
LDX	iiii	8E	X=iiii	- - - T T Ø -
	aa	9E	X=M(aa:aa+1)	
	xxop	AE	X=xxop	
	aaaa	BE	X=M(aaaa:aaaa+1)	
LDY	iiii	108E	Y=iiii	- - - T T Ø -
	aa	109E	Y=M(aa:aa+1)	
	xxop	10AE	Y=xxop	
	aaaa	10BE	Y=M(aaaa:aaaa+1)	
LEAS	xxop	32	S=effad xxop	- - - - - - -
LEAU	xxop	33	U=effad xxop	- - - - - - -
LEAX	xxop	30	X=effad xxop	- - - - T - -
LEAY	xxop	31	Y=effad xxop	- - - - T - -
PULS	rl	35	pull register(s) from M(S)	? ? ? ? ? ? ?
PULU	rl	37	pull register(s) from M(U)	? ? ? ? ? ? ?

(6809 store / push instructions)

Mnemonic	Operand	Op-code	Function	Status F H I N Z V C
STA	aa	97	M(aa)=(A)	- - - T T 0 -
	xxop	A7	xxop=(A)	
	aaaa	B7	M(aaaa)=(A)	
STB	aa	D7	M(aa)=(B)	- - - T T 0 -
	xxop	E7	xxop=(B)	
	aaaa	F7	M(aaaa)=(B)	
STD	aa	DD	M(aa:aa+1)=(D)	- - - T T 0 -
	xxop	ED	xxop=(D)	
	aaaa	FD	M(aaaa:aaaa+1)=(D)	
STS	aa	10DF	M(aa:aa+1)=(S)	- - - T T 0 -
	xxop	10EF	xxop=(S)	
	aaaa	10FF	M(aaaa:aaaa+1)=(S)	
STU	aa	DF	M(aa:aa+1)=(U)	- - - T T 0 -
	xxop	EF	xxop=(U)	
	aaaa	FF	M(aaaa:aaaa+1)=(U)	
STX	aa	9F	M(aa:aa+1)=(X)	- - - T T 0 -
	xxop	AF	xxop=(X)	
	aaaa	BF	M(aaaa:aaaa+1)=(X)	
STY	aa	109F	M(aa:aa+1)=(Y)	- - - T T 0 -
	xxop	10AF	xxop=(Y)	
	aaaa	10BF	M(aaaa:aaaa+1)=(Y)	
PSHS	rl	34	push register(s) onto M(S)	- - - - - - -
PSHU	rl	36	push register(s) onto M(U)	- - - - - - -

(6809 shift / rotate instructions)

Mne-onic	Oper-and	Op-code	Function	Status F	H	I	N	Z	V	C
LSL	aa	08	M(aa)=M(aa) A< 1	-	?	-	T	T	?	T
	xxop	68	xxop = xxop A< 1							
	aaaa	78	M(aaaa) = M(aaaa) A< 1							
LSLA	--	48	A=(A) A< 1	-	?	-	T	T	?	T
LSLB	--	58	B=(B) A< 1	-	?	-	T	T	?	T
LSR	aa	04	M(aa)=M(aa) L> 1	-	-	-	0	T	-	T
	xxop	64	xxop=xxop L> 1							
	aaaa	74	M(aaaa)=M(aaaa) L> 1							
LSRA	--	44	A=(A) L> 1	-	-	-	0	T	-	T
LSRB	--	54	B=(B) 1> 1	-	-	-	0	T	-	T
ROL	aa	09	M(aa)=M(aa) R< 1	-	-	-	T	T	?	T
	xxop	69	xxop=xxop R< 1							
	aaaa	79	M(aaaa)=M(aaaa) R< 1							
ROLA	--	49	A=(A) R< 1	-	-	-	T	T	?	T
ROLB	--	59	B=(B) R< 1	-	-	-	T	T	?	T
ROR	aa	06	M(aa)=M(aa) R> 1	-	-	-	T	T	?	T
	xxop	66	xxop=xxop R> 1							
	aaaa	76	M(aaaa)=M(aaaa) R> 1							
RORA	--	46	A=(A) R> 1	-	-	-	T	T	?	T
RORB	--	56	B=(B) R> 1	-	-	-	T	T	?	T

(6809 branching instructions)

Mnemonic	Operand	Opcode	Function	Status F H I N Z V
BCC	rr	24	P=(P)+2+rr iff (C)=0	- - - - - - -
BCS	rr	25	P=(P)+2+rr iff (C)=1	- - - - - - -
BEQ	rr	27	P=(P)+2+rr iff (Z)=1	- - - - - - -
BGE	rr	2C	P=(P)+2+rr iff (N) xor (V)=0	- - - - - - -
BGT	rr	2E	P=(P)+2+rr iff (Z) or [(N) xor (V)]=0	- - - - - - -
BHI	rr	22	P=(P)+2+rr iff (C) xor (Z)=0	- - - - - - -
BHS	rr	24	P=(P)+2+rr iff (C)=0	- - - - - - -
BLE	rr	2F	P=(P)+2+rr iff (Z) or [(N) xor (V)]=1	- - - - - - -
BLO	rr	25	P=(P)+2+rr iff (C)=1	- - - - - - -
BLS	rr	23	P=(P)+2+rr iff (C) or (Z)=1	- - - - - - -
BLT	rr	2D	P=(P)+2+rr iff (N) xor (V)=1	- - - - - - -
BMI	rr	2B	P=(P)+2+rr iff (N)=1	- - - - - - -
BNE	rr	26	P=(P)+2+rr iff (Z)=0	- - - - - - -
BPL	rr	2A	P=(P)+2+rr iff (N)=0	- - - - - - -
BRA	rr	20	P=(P)+2+rr (always)	- - - - - - -
BRN	rr	21	P=(P)+2 (a 2 byte no-op)	- - - - - - -
BVC	rr	28	P=(P)+2+rr iff (V)=0	- - - - - - -
BVS	rr	29	P=(P)+2+rr iff (V)=1	- - - - - - -
JMP	aa	0E	P=00aa	- - - - - - -
	xxop	6E	P=xxop	
	aaaa	7E	P=aaaa	
LBCC	rrrr	1024	P=(P)+4+rrrr iff (C)=0	- - - - - - -
LBCS	rrrr	1025	P=(P)+4+rrrr iff (C)=1	- - - - - - -
LBEQ	rrrr	1027	P=(P)+4+rrrr iff (Z)=1	- - - - - - -
LBGE	rrrr	102C	P=(P)+4+rrrr iff (N) xor (V)=0	- - - - - - -
LBGT	rrrr	102E	P=(P)+4+rrrr iff (Z) or [(N) xor (V)]=0	- - - - - - -
LBHI	rrrr	1022	P=(P)+4+rrrr iff (C) xor (Z)	- - - - - - -
LBHS	rrrr	1024	P=(P)+4+rrrr iff (C)=0	- - - - - - -
LBLE	rrrr	102F	P=(P)+4+rrrr iff (Z) or [(N) xor (V)]=1	- - - - - - -
LBLO	rrrr	1025	P=(P)+4+rrrr iff (C)=1	- - - - - - -
LBLS	rrrr	1023	P=(P)+4+rrrr iff (C) or (Z)=1	- - - - - - -
LBLT	rrrr	102D	P=(P)+4+rrrr iff (N) xor (V)=1	- - - - - - -
LBMI	rrrr	102B	P=(P)+4+rrrr iff (N)=1	- - - - - - -
LBNE	rrrr	1026	P=(P)+4+rrrr iff (Z)=0	- - - - - - -
LBPL	rrrr	102A	P=(P)+4+rrrr iff (N)=0	- - - - - - -
LBRA	rrrr	16	P=(P)+3+rrrr (always)	- - - - - - -
LBRN	rrrr	1021	P=(P)+4 (a 4 byte no-op)	- - - - - - -
LBVC	rrrr	1028	P=(P)+4+rrrr iff (V)=0	- - - - - - -
LBVS	rrrr	1029	P=(P)+4+rrrr iff (V)=1	- - - - - - -

(6809 subroutine & special function instructions)

Mnemonic	Operand	Opcode	Function	Status F	H	I	N	Z	V	C
BSR	rr	8D	M(S-2)=(P)+2; P=(P)+2+rr; S=(S)-2	-	-	-	-	-	-	-
JSR	aa	9D	M(S-2)=(P)+2; P=00aa; S=(S)-2	-	-	-	-	-	-	-
	xxop	AD	M(S-2)=(P)+2; P=xxop; S=(S)-2							
	aaaa	BD	M(S-2)=(P)+3; P=aaaa; S=(S)-2							
LBSR	rrrr	17	M(S-2)=(P)+3; P=(P)+3+rrrr; S=(S)-2	-	-	-	-	-	-	-
SWI	--	3F	M(S-2)=(P)+1; P=M($FFFA); S=(S)-2	-	-	-	-	-	-	-
SWI2	--	103F	M(S-2)=(P)+2; P=M($FFF4); S=(S)-2	-	-	-	-	-	-	-
SWI3	--	113F	M(S-2)=(P)+2; P=M($FFF2); S=(S)-2	-	-	-	-	-	-	-
RTI	--	3B	P=M(S); S=(S)+2	?	?	?	?	?	?	?
RTS	--	39	P=M(S); S=(S)+2	-	-	-	-	-	-	-
DEC	aa	0A	M(aa)=M(aa)-1	-	-	-	T	T	?	-
	xxop	6A	xxop=xxop-1							
	aaaa	7A	M(aaaa)=M(aaaa)-1							
DECA	--	4A	A=(A)-1	-	-	-	T	T	?	-
DECB	--	5A	B=(B)-1	-	-	-	T	T	?	-
INC	aa	0C	M(aa)=M(aa)+1	-	-	-	T	T	?	-
	xxop	6C	xxop=xxop+1							
	aaaa	7C	M(aaaa)=M(aaaa)+1							
INCA	--	4C	A=(A)+1	-	-	-	T	T	?	
INCB	--	5C	B=(B)+1	-	-	-	T	T	?	
NOP	--	12	P=(P)+1 (1 byte no-op)	-	-	-	-	-	-	-
CWAI	ii	3C	clear & wait for interrupt	?	?	?	?	?	?	?
SYNC	--	13	Syncronize	?	?	?	?	?	?	?

(POST BYTE)

(indexed addressing)

the value of the post-byte (the 1st byte following the opcode) for instructions using the indexed addressing mode is determined by the format of the operand. Two formats exist: Simple indexing and complex indexing. Simple indexing is used when the operand is of the form:

<exp>,R

where <exp> is an expression in the range -16 to 15 but not equal to zero, and R is one of the index registers "S", "U", "X", or "Y". All other indexed addressing modes use the complex indexing format. The two post-byte formats are:

(simple)	(complex)
0 R R X X X X X	1 R R I T T T T

where R R = 00 for X register (or PCR if TTTT = 110x)
01 for Y register
10 for U register
or 11 for S register

and T T T T = 0000 for single auto increment (R+)
0001 for double auto increment (R++)
0010 for single auto decrement (R-)
0011 for double auto decrement (R--)
0100 for 0 offset or no offset
0101 for register B is offset (B,R)
0110 for register A is offset (A,R)
1000 for 8 bit offset (aa,R)
1001 for 16 bit offset (aaaa,R)
1011 for register D is offset (D,R)
1100 for 8 bit offset with PCR
1101 for 16 bit offset with PCR
or 1111 for extended indirect addressing

and I = 0 for direct addressing
or 1 for indirect addressing

(X X X X X is a 5 bit 2's compliment offset)

APPENDIX C

(ERROR MESSAGES)

173 Invalid use of direct mode indicator. The direct mode indicator ("<") was used in the indirect extended addressing mode. The "<" is ignored.

174 Invalid auto increment/decrement format. Single auto increment or decrement was used in the indirect mode, or more than 2 "+" or "-" encountered.

175 Invalid index register format. One of the accumulators "A", "B", or "D" was used as the offset in the indexed mode, but was not followed by one of the index registers "S", "U", "X", or "Y".

176 Invalid expression for PSH/PUL The immediate expression following one of the instructions "PSHS", "PULS", "PSHU", or "PULU" contained symbols defined with other than the "REG" directive, contained an operator other than "!+" or contained no symbol following the "#".

177 Incompatable register for PSH/PUL instruction. The register list for the PSHS/PULS instructions cannot contain the register "S", and the register list for the PSHU/PULU instructions cannot contain the register "U".

178 Invalid register operand specification. An undefined register name was encountered in a register list or there are not exactly two register names in the register list for a TFR/EXG instruction or no register list was specified for a PSH/PUL instruction.

179 Incompatable register pair. The register pair of an EXG instruction was not the same size or the register pair of a TFR instruction specified a transfer from an 8 bit register to a 16 bit register.

202 Label or opcode error. The label or opcode symbol does not begin with an alphabetic character or a period.

205 Label error. The statement's label field is not terminated with a blank. This Usually occurs if an invalid character is used in the label.

207 Undefined opcode. The symbol in the opcode field is not a valid 6809 (or 6800) mnemonic nor is it a valid ASM09 directive.

208 Branch out of range. The operand resulted in a forward offset of >129 bytes or a backward offset of >126 bytes from the first byte of the branch instruction.

209 Illegal addressing mode. The specified addressing mode in the operand field is not valid for this instruction type.

210 Byte overflow. The operand's value exceeded 1 byte (8 bits). The most significant 8 bits of the 16 bit value of the expression must be all ones or all zeroes for a one byte field. (due to sign propagation of 2's compliment numbers)

211 Undefined symbol. A symbol was referenced in an operand field but has never occured in the label field of a correctly assembled instruction.

212 Directive operand error. A syntax error was detected in the operand field of a ASM09 directive.

214 FCB directive syntax error. The structure of the FCB is incorrect.

215 FDB directive syntax error. The structure of the FDB is incorrect

216 Directive operand error. The directive's operand field is missing, terminated by an invalid character, or an expression in the operand field contains an invalid operator.

217 Option error. An option in the operand field of the OPT directive is undefined.

219 No "END" statement. The end of the last source file has been encountered before an "END" directive. ASM09 has supplied the "END".

220 Phasing error. The value of the program counter during pass II is different from the value during pass I for the same instruction.

221 Symbol table overflow. The symbol table has overflowed. This is a FATAL pass I error which will cause ASM09 to suspend assembly.

222 Reserved symbol used. One of the reserved symbols ("A", "B", "X", "Y", "U", "S", "D", "CC", "DP", "PC", or "PCR") appeared in the label field or in the operand field of an instruction. These symbols can only be used in the operation field to modify a root mnemonic ("A" or "B") or in the operand field to specify indexed addressing.

223 Invalid label for directive. Depending on the directive, there must be or must not be a symbol in the label field of this instruction.

226 Illegal parenthesis. The parentheses "(" or ")" in an expression do not balance. EG there is/are more of one than the other.

227 Too many digits in a numeric constant. An overflow has occured during the evaluation of a numeric constant. May also occur if a sequence number is present / missing on a line in a file which has not / has sequence numbers.

229 Invalid starting execution address. The starting execution address given in the expression on the "END" statement is not valid.

233 Symbol name too large. A symbol of greater than 6 characters was incountered.

234 Multiply defined symbol. A reference was made to a multiply defined symbol.

235 Memory error. The "OPT MEM" directive was used and object code was going to be written into non-existant memory or into the memory occupied by ASM09. The write operation has been aborted & no further writes will be done.

236 Program counter overflow. The program counter overflowed beyond $FFFF.

237 invalid terminator for sequence number. The character following a user supplied sequence number was not a blank.

241 Illegal symbol used in an expression. An undefined forward referend symbol was used in an expression. This can occur in the PCR indexing mode.

242 OPT directive error. Conflicting options were specified or the "OPT CRE" directive was encountered after the first symbol had been placed into the symbol table.

244 Illegal page or line length. The "OPT LLE=nnn" or "OPT P=nnn" directive(s) was(were) used with "nnn" being outside the allowable range.

247 Invalid operand terminator. The character following the legal part of an operand is not a valid terminator (usually a CR or a space). May also be caused by invalid indirect pairing (ie "[" & "]" not balanced).

(WARNING MESSAGES)

1 Long branch not required. A long branch instruction was used to branch to an address within the range of -126 to +129. Although the long branch instruction could be changed to a short branch, it could result in other out-of-range short branches.

2 Extended addressing should be used. Direct addressing was forced by using the "<" indicator. However, the direct-page psuedo register (set by the "SETDP" directive) indicated that the location was accessable via the extended addressing mode.

3 Duplicate register specification. The same register was specified more than once in a register list. Register "D" specified with either "A" or "B" will give this warning.

4 Possible SETDP expression error. The most significant byte of an expression in a "SETDP" directive was not zero.

5 Extended addressing should have been used. Direct addressing was forced by using the "<" indicator.

6 Possible transfer error. The TFR instruction was used to transfer from a 16 bit register to an 8 bit register. The result of such a transfer is to move the lower 8 bits & to truncate the upper 8 bits.

APPENDIX D

(LISTING DESCRIPTION)

COLUMN	CONTENTS
1-5	Source line number
8:11	Program Counter
13:14	1st (or only) byte of op-code
15:16	2nd byte of op-code (if any)
18:19	1st byte of operand
20:21	2nd byte of operand (if any)
	for non-branch, non-indexed instructions:
18-19	1st byte of operand
20:21	2nd byte of operand (if any)
	for non-branch, indexed instructions:
18:19	index post-byte
21:22	1st byte of operand
23:24	2nd byte of operand (if any)
	for branch instructions:
18:19	1st byte of relative branch offset
20:21	2nd byte of offset (if any)
23:26	absolute address of target
	for 6800-equivalent instructions:
18:19	2nd byte of translated instruction
21:22	3rd byte of instruction (if any)
23:24	4th byte of instruction (if any)
	for directives like BSZ, EQU, ORG, ETC. :
23:26	value of reduced operand expression
28:33	Label
35:40	Operation
41:48	Operand; longer operands extend into comment field
50:120	Comment(s)

CROSS REFERENCE CONCORDANCE FORMAT

COLUMN	CONTENTS
1	Symbol's "type": U = udefined symbol M = multiply defined symbol S = "set" symbol
4:7	Hexadecimal value of symbol
9:14	The symbol
16:?	List of source line numbers where the symbol has been referenced. An "*" appears after the line number of a symbol's definition. If the symbol is undefined, the "*" will appear after the line number of the last reference to the symbol.

APPENDIX E

(USEFUL INFORMATION)

1) USAGE OF SYSTEM'S DATE & TIME

In order to use the system's date / time facility, the vector at $D2E2 (in a "$C000" system) must point to somewhere other than the system's warm-start routine. ASM09 checks this, and if the vector is an effective "warm start", the "T" option is ignored. This vector should read the date / time hardware or do whatever is necessary to get a date & time string into the system's parameter table. These strings must be 15 or less bytes long (for the date) and 8 or less bytes long (for the time) and both must be terminated by a $00. See the DOS69 manual for more details. In order to use the strings (initialized with "SET") with out having a date / time driver (or hardware), it is suggested that you change the vector (at $D2E2 in a $C000 system) to:

```
$39,$00,$00
```

This will cause ASM09 to presume that the date / time has been read. ASM09 will then retrive the bogus date / time which is in the system's tables. See the DOS69 listing for more detailed information. i

2) I/O BREAK, PAUSE, & RESUME

ASM09 will moniter the keyboard for a keypress during assembly. If one is detected, then the key is checked. If this key is one of the three control characters (see the DOS69 manual for information on changing the characters) defined by the system's parameter table, then ASM09 will take the appropriate action. If any other key is pressed, it will be ignored. The three pre-defined keys are:

char	function
^C	causes assembly to be aborted
esc	stops the assembly (waits for start or abort)
esc	starts the assembly again

This is useful to "throttle" the listing if it is going to a CRT.