RECORDS ADMINISTRATION

R0139096

DP-1506 ✓

TIS FILE
RECORD COPY

MC6800 CROSS-ASSEMBLER FOR THE PDP-8/E DIGITAL COMPUTER

R. J. SAND

SAVANNAH RIVER LABORATORY
AIKEN, SOUTH CAROLINA 29801

PREPARED FOR THE U.S. DEPARTMENT OF ENERGY UNDER CONTRACT AT(07-2)-1

NOTICE

This report was prepared as an account of work sponsored by the United States Government. Neither the United States nor the United States Department of Energy, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied or assumes any legal liability or responsibility for the accuracy, completeness or usefulness of any information, apparatus, product or process disclosed, or represents that its use would not infringe privately owned rights.

Printed in the United States of America

Available from

National Technical Information Service

U.S. Department of Commerce

5285 Port Royal Road

Springfield, Virginia 22161

Price: Printed Copy \$6.00; Microfiche \$3.00

726086 ✓

DP-1506

Distribution Category: UC-32

MC6800 CROSS-ASSEMBLER FOR THE PDP-8/E DIGITAL COMPUTER

R. J. Sand

Approved by

**D. Baker, Manager
Laboratory Operations and Services**

Publication Date: August 1978

**E. I. DU PONT DE NEMOURS AND COMPANY
SAVANNAH RIVER LABORATORY
AIKEN, SOUTH CAROLINA 29801**

PREPARED FOR THE U. S. DEPARTMENT OF ENERGY UNDER CONTRACT AT(07-2)-1

ABSTRACT

A cross-assembler has been developed to assemble Motorola MC6800 microprocessor programs on a Digital Equipment Corporation PDP-8/E minicomputer. This cross-assembler runs in 8K of core under the OS/8 operating system. The advantages of using the cross-assembler are the large user symbol table and the convenience and speed of program development. User's instructions for the cross-assembler are given. The design of the cross-assembler and examples of its use are described.

CONTENTS

Introduction	5
M68CA User's Manual	7
Introduction	7
Cross-Assembler Syntax	7
Symbols	7
Operators	9
Connectives	11
Expressions	12
Data Storage	12
Location Control	14
Using "Period" in an Expression	15
Program Preparation	15
Pseudo-Operations	16
Instruction Set	16
Operating Instructions	16
Error Diagnostics	17
Expression Evaluation Details	18
M68CA Cross-Assembler Design	30
Program Structure	30
Program Flowcharts	41
Module Functions	63
Memory Map	74
Conclusion	75
Acknowledgement	75
References	76
Appendix A - Example Programs	77
Appendix B - Binary Tape Format	83
Appendix C - Standard Symbols	85

LIST OF FIGURES

1.	The M68CA Cross-Assembler Program Structure	31
2.	Symbol Table Generation Structure	32
3.	Binary Tape Generation Structure	33
4.	Listing Generation Structure	34
5.	User Symbol Table Listing Generation Structure	35
6.	Evaluation Modules Structure	36
7a.	Service Modules Structure	37
7b.	Service Modules Structure	38
7c.	Service Modules Structure	39
7d.	Service Modules Structure	40
8.	Pass 0 Flowchart	42
9a.	Pass 1 Flowchart	43
9b.	Pass 1 Flowchart	44
9c.	Pass 1 Flowchart	45
9d.	Pass 1 Flowchart	46
9e.	Pass 1 Flowchart	47
10a.	Pass 2 Flowchart	48
10b.	Pass 2 Flowchart	49
10c.	Pass 2 Flowchart	50
10d.	Pass 2 Flowchart	51
10e.	Pass 2 Flowchart	52
10f.	Pass 2 Flowchart	53
11a.	Pass 3 Flowchart	54
11b.	Pass 3 Flowchart	55
11c.	Pass 3 Flowchart	56
11d.	Pass 3 Flowchart	57
11e.	Pass 3 Flowchart	58
11f.	Pass 3 Flowchart	59
11g.	Pass 3 Flowchart	60
12a.	Pass 4 Flowchart	61
12b.	Pass 4 Flowchart	62

MC6800 CROSS-ASSEMBLER FOR THE PDP-8/E DIGITAL COMPUTER

INTRODUCTION

The M68CA MC6800 cross-assembler program is for use in developing programs for the Motorola MC6800 microprocessor. The MC6800 is one of the leaders in a computer revolution that has been happening since Intel, Inc. introduced the C4004, a 4-bit parallel microprocessor, in 1971.

Since then, the electronics industry has become deeply involved in developing and refining the microcircuit technology required to produce the "computer on a chip" which is being designed into everything from toys to kitchen appliances to industrial controllers. In 1971, the C4004 sold for over \$200 each. In 1977, the 8-bit parallel microprocessor unit MC6800 sold for \$20. This dramatic price drop, with its accompanying high use rate, has produced a bottle neck in the design phase of electronic equipment.

The current generation of microprocessor circuits has made control equipment design and fabrication a combination of electronic hardware and software. Present trade journals and literature place more than half of total development costs in

the software realm. It is therefore advantageous to be able to convert the software as rapidly as possible from the human assembly language to the binary machine language necessary for system operation.

Microprocessors are by definition small and as such do not execute instructions as fast as some minicomputers. In developing software programs for microprocessors, it is beneficial to do so on larger, faster machines. This requires a cross-assembler, a program run on one computer for the purpose of translating instructions for a different computer.

The cross-assembler speeds up the programming process. But in a research environment where several different microprocessors may be used, cross-assemblers for each with nearly identical operating characteristics are almost a necessity. The M68CA MC6800 cross-assembler has been developed with these considerations in mind. It is one of four cross-assemblers running on a PDP 8/E disk based minicomputer. This document describes the operation of the M68CA cross-assembler program and provides information to aid in the maintenance and modification of the program.

M68CA USER'S MANUAL

INTRODUCTION

The M68CA MC6800 cross-assembler operates in a PDP-8* with an 8K core under OS/8* control and provides symbol table capacity for 768 symbols. The set of standard M68CA M6800 instructions requires 226 symbols, leaving 542 symbols for the user. The cross-assembler uses four passes:

- Pass 1 - defines the symbols in the program being assembled.
- Pass 2 - produces an object file of the program in binary format.
- Pass 3 - produces an assembly listing.
- Pass 4 - produces a symbol table file.

ASSEMBLER SYNTAX

In general, the cross-assembler syntax is free-form, allowing the user the ability to format his coding as he wishes. The few basic rules are:

- symbols identifying the current location must appear to the left of any other characters in the line and must be followed by a comma (,).
- grouping symbols via parentheses is not allowed in the syntax,
- evaluation of expressions is from left to right.

The accepted form of a coded statement is as follows:

`<symbol>,<expression>/<comment>`

It is not necessary to start a symbol at the left margin, nor is it necessary to start the expression immediately following the comma which terminates the symbol. The components of the statement may appear in combination or by themselves as the user desires.

SYMBOLS

Alphanumeric symbols

The cross-assembler forms instructions by combining symbol values and/or numerics as specified by the arithmetic and logical connectives. All symbols are stored in a symbol table which is created during Pass 1 of the assembly.

*PDP-8 and OS/8 are trademarks of Digital Equipment Corporation.

To evaluate an expression, the cross-assembler searches the symbol table for the given symbol and then substitutes the value found in the symbol table for the symbol. The values for each symbol in the expression and the numerics are then combined according to the rules of the cross-assembler.

A symbol consists of from one to six consecutive alphanumeric characters from the character sets A-Z and 0-9. The first character of a symbol must be alphabetic. The cross-assembler only uses the first six characters if the symbol contains more than six.

The cross-assembler allows the user to include other characters in a symbol, such as the "greater than," "less than," "open" and "closed" parentheses, and others. The user should take care not to include those characters already in use by the cross-assembler as indicated later in the text.

Symbols may be defined in two ways:

A line of the form

symbol,expression

The symbol will be given the value of the location into which the expression will be assembled. This method is used to assign symbolic values to particular locations in the program.

A line of the form

symbol=expression

The expression to the right of the equal sign is evaluated and the symbol is given this value.

Instruction symbols

The set of M68CA instructions is defined by 3- and 4-character alphabetic symbols. Each instruction generates one, two, or three 8-bit bytes to represent that instruction; the number of bytes required is called the instruction's attribute. Thus, if an instruction has an attribute of 2, when the expression containing the symbol for this instruction is evaluated, two bytes will result.

Addressing mode symbols

The set of M68CA addressing mode symbols is defined by 3-character alphabetic symbols. The symbols are:

IMS	Immediate Addressing, Single Byte
DIR	Direct Addressing
IND	Indexed Addressing
EXT	Extended Addressing
IMD	Immediate Addressing, Double Byte
Inherent is as implied and does not require definition	

Numbers

A number is represented by one to six hex digits whose value does not exceed 16 bits in length (i.e., less than 10000 hex). This number thus represented is used to specify an address (two bytes) or a quantity to be used in a load immediate instruction; i.e., as part of an expression and not to be confused with the numeric value used as a data value. Data value generation is explained later in this section under Operators.

Numbers must begin with a digit, 0-9 (e.g., FA1 is coded 0FA1, 27 is coded 27).

The number does not appear in the symbol table.

OPERATORS

The characters "minus" (-), "apostrophe" ('), "at" (@), "quote" ("), and "pound" (#) are the set of operators and are used to change the value of a symbol during an evaluation.

Minus (-)

The operator "minus" negates the 16-bit value of the symbol defined by data words 2 and 3. During evaluation, three bytes are affected even though only one byte or two bytes may be generated for the final output. For example, the load immediate instructions require only two bytes, but during the evaluation all three possible bytes are involved.

Word 1 is the instruction word; words 2 and 3 are the data words related to that instruction.

Apostrophe (')

The operator "apostrophe" reverses the second and third data words of the value of the symbol immediately following the operator.

At (@)

The operator "at" is used primarily to set up instruction symbols in the symbol table and to set up data words. To explain the function of the "at" operator, the user should recall the term "attribute", the number of bytes required by the instruction on evaluation. The attribute of each symbol is saved in the symbol table, and its value is determined by the use of the "at" operator in the symbol table evaluation during the assembly. The numeric value which immediately follows the "at" operator contains the resultant evaluation and the attribute, where the attribute is the rightmost digit. Thus, to redefine (or create) the hex equivalent for the extended jump instruction (attribute 3), the symbol source file would contain a line:

```
JMPE=@7E3
```

where

"@" is the operator.

7E is the hex representation for a JMP EXT instruction.

3 is the attribute (number of bytes required for the JMP EXT instruction).

Quote (")

The operator "quote" is used to convert a single character to its ASCII equivalent without the parity bit; i.e., the eighth level bit is stripped off. For example, if the user wished to compare data read from the ASR33 (with the eighth level bit cleared) to a constant, he would use the compare instruction, as follows:

```
"A" accumulator contains the character read  
CMPA IMS "$
```

The evaluation of "\$ is 24 (hex). If the data read were \$ (24), then the comparison would be true. If the user did not wish to strip the parity bit (eighth level bit), he could perform the comparison as follows:

```
"A" accumulator contains the character read  
CMPA IMS "$+80
```

Pound (#)

The "pound" operator is used to create data in the program. The user simply precedes the hex numeric value with "#", and the cross-assembler generates a one-byte constant. By adding a "T" after the hex value, the user causes the cross-assembler to generate a two-byte value (double-precision constant). The cross-assembler only accepts the first 6 numeric characters in the string; all other numeric characters are ignored. The user can add the quantities of the same byte count (single precision plus single precision, etc.). This is a data operator and is not to be used with an equal sign to evaluate a symbol.

Examples:

```
TABL:  #10F
        #0ABCDT
        #53+#172
        #37F2T+#0B8DT
```

CONNECTIVES

The set of characters plus (+), TAB, and SPACE are the formal connectives.

The operators described in the previous section also serve as connectives.

The formal connectives are equivalent and operate on two values, resulting in a new value. The rules for evaluating X connective Y are as follows:

- The attributes of X and Y are inclusive ORed.
- The instruction words of X and Y are inclusive ORed.
- The data words of X and Y are added with overflow from the low-order sum resulting in an increment of the high-order sum.

If an operator is used as a connective, it is applied to the symbol prior to the connective operation.

When the above rules are used, an expression with two symbols must have compatible attributes; i.e., if the attribute of the first symbol is 2 and the second symbol is 1, then the result will be 3, which in some cases may not be the desired result. For example, it is incorrect to use the data operator "#" with a load immediate instruction:

LDAA IMS #276

The result is a three-byte instruction, and the LDAA IMS instruction requires only two bytes. The proper coding would be:

LDAA IMS 276

where LDAA has an attribute of 2, IMS has an attribute of 0, and the numeric "276" has an attribute of 0.

EXPRESSIONS

Each line of the source program constitutes an "expression" - a group of symbols with operators joined together by connectives and governed by the rules given below. A line is terminated by the character "carriage return" (CR).

The following rules apply to the relationship between operators, symbols, and connectives when forming an expression:

A connective does not have to be present before an operator; e.g., xxx-yyy is valid.

A connective may be present between the operator and its associated symbol; e.g.,

```
LDAA IMS'500
LDAA IMS '+400
#(TAB)57
X=@++202
```

are all valid expressions.

Expressions are evaluated from left to right.

Parentheses are not allowed within an expression (except as a character in a symbol); e.g., there is no provision in the cross-assembler for distributive operations: A-(X+Y) is invalid.

DATA STORAGE

To store data in one or two words of memory, the numeric operator "pound" (#) is used. One-word data is generated by placing a pound sign (#) before the hex value to be put into the memory word. For example, to generate

the hex data word 82, the user would code "82." The result is one word with the value 82 in the current memory location. The cross-assembler generates the numeric value modulo $2^{*}8$ (256); i.e., the expression "6F55" would become "55" in memory.

To store two words of data, the user precedes the data with the pound sign and then immediately follows the data with a "T"; i.e., the constant "FC00" is coded as "#0FC00T." The cross-assembler generates double precision data modulo $2^{*}16$ (65536), using only the first six characters.

The statement format is as follows:

<symbol>,<1-6 hex digits>	(<symbol>optional) (hex value from 0-0FF)
<symbol>,<1-6 hex digits>T	(<symbol>optional) (hex value from 0-0FFFF)

If the user wishes to store the value of a symbol or the address of a symbol in a memory location, he uses the following statement format,

<symbol>,@<n><connective><expression>(<symbol>optional)

To store the value of xxx, where $xxx=4321$, then

xyz,@2 xxx

would store 4321 in two bytes, high-order byte first, at location xyz.

To store the address of yyy, then

xyz,@2 yyy

would store the address of yyy in two bytes, high-order byte first, at location xyz.

To store the high-order portion of the address, then

xyz,@1 yyy

would store the high-order portion in one byte location xyz.

To store the low-order portion of the address, then

```
xyz,@1 'yyy
```

would store the low-order portion in one byte at location xyz.

In this function, the "at" operator is used to assign an attribute to a line which otherwise would have an attribute of 0.

The user can generate a numeric equivalent for any ASCII character string by using the TEXT pseudo-op. The format of this statement is:

```
<symbol>,TEXT <nn> <any text up to 25 hex ASCII characters>
```

where nn(hex) indicates the number of characters which follow (the space between nn and the first character is not included in the count).

The ASCII data which is generated is loaded into memory starting at the current location and running consecutively for nn locations. The data does not include the parity bit (eighth level), which is left equal to zero. For example, the user wishes to generate the numeric code to print the message "BAD"; the following statement is used:

```
XYZ,TEXT 5 BAD(CR)(LF)
```

where the numeric values generated include the carriage return and line feed (the same carriage return and line feed that terminate the statement).

LOCATION CONTROL

Any line containing an asterisk (*) to the left of a (possible) slash will be treated as an origin statement, and the location counter will assume the value of the expression constituting the line.

For example,

```
*200
```

```
200    *
```

```
BEGIN=200
```

```
BEGIN*
```

all establish a value of 200 for the location counter.

In addition, there is a special symbol for the location counter - the period (.). This symbol may be used to reserve locations; for example:

```
*.+120
```

causes the location counter to be set to its present value plus 120.

The value of "." is equal to the location into which the first byte of the instruction will be assembled.

USING "PERIOD" IN AN EXPRESSION

The symbol "." may be used for expression addressing (e.g., `JMP EXT .+9`). However, this technique should be used only with the greatest care, since instructions occupy a variable number of bytes. For example:

```
JMP EXT .+2  
BSR SUBR1  
BSR SUBR2
```

would cause a jump to the last byte of the `JMP EXT` instruction, not to the `BSR SUBR2` instruction. `JMP EXT .+5` would jump to the `BSR SUBR2` instruction. Extensive use of relative addressing is not recommended programming technique. It creates problems when modifying existing programs.

PROGRAM PREPARATION

Source program files may be prepared by using the PDP-8 Editor. The TAB character may be used freely to line up listing columns. Pass 3 uses the TAB character to provide simulated tab stops every eight character spaces. The form feed character appended to each Editor page will cause a simulated "top of form" on the listing.

Each line of source code must be terminated with a CR/LF combination.

PSEUDO-OPERATIONS

TEXT Text is used to convert ASCII strings to their numeric equivalent.

\$ The dollar mark is used to mark the end of the source program.

INSTRUCTION SET

The set of machine instructions for the M68CA Assembly Language is very similar to the Motorola M6800 Assembly Language. The M68CA primary operand does not allow a space before the accumulator designation. The M68CA addressing modes are indicated by 3-character symbols as opposed to the special characters and automatic decisions by the Motorola M6800 Assembly Language. Labels (symbols with a comma as a terminator) and comments are optional on all statements. The general format for an instruction line, including the two options mentioned above, is:

<symbol>,<instruction expression>/<comment>

OPERATING INSTRUCTIONS

The cross-assembler is loaded from the system device by using the PDP-8 OS/8.

.R M68CA

It responds by activating the OS/8 Command Decoder which prints an asterisk.

.R M68CA
*

A command string is entered which indicates the binary and listing output devices and file names, and the input device and file name.

.R M68CA
*NAME1,NAME2<NAME.S

Assembly will begin after a carriage return. Output files are available on the indicated device when control returns to the OS/8 Keyboard Monitor. The

new files will be identified NAME1.BN, and NAME2.LS. The files may not be complete. The complete output files may be manipulated as desired by using OS/8 Peripheral Interchange Program (PIP). The binary file must be moved in the Image (/I) mode.

ERROR DIAGNOSTICS

Certain errors may be detected during an assembly. They are identified by a 2-character error identifier, printed on the listing device (teletype), along with the memory location, in the following format:

* error identifier AT memory location *

The following is a listing of the error identifiers and their meanings:

- ST Symbol table full. The cross-assembler returns to OS/8 Monitor on this error. Source program must be modified to reduce the number of symbols defined.
- UO Undefined origin expression. The expression defining an origin cannot be evaluated. The cross-assembler uses the present value of the location counter.
- UE Undefined expression. One or more terms of an expression are not defined at this point. They may be defined later in the assembly.
- RD Redefinition. A symbol already in the symbol table has been given a new definition by an expression containing an equal sign (=).
- MT Multiply defined symbol. The same symbol has been used more than once as a symbol.
- UI Undefined instruction. One or more terms of an expression generating binary output are undefined. The cross-assembler will generate a return to OS/8 Monitor if it does not have an attribute for the instruction.
- SL Source line too long. More than 72 characters appear between line terminators, exceeding the capacity of the cross-assembler source line input buffer. The cross-assembler returns to OS/8 Monitor on this error. The source program must be edited to shorten the line.

- IN Illegal number. A symbol beginning with a numeric character contains a non-numeric character. A position calling for only a number does not begin with 0-9.
- BE Branch error. An instruction attempted to branch to a location which exceeded the signed 7-bit distance (-125,+129) from the beginning of the next instruction.

EXPRESSION EVALUATION DETAILS

Symbols

The cross-assembler forms instructions by combining symbol values as specified by arithmetic and logical connectives. The cross-assembler does not distinguish between M68CA instruction symbols and user symbols in assembling a word. Thus, the user should not define any symbol which is identical to any symbol used to define a standard M68CA instruction symbol; e.g., TAB, IND, etc. (This error would be flagged during Pass 1.)

Every symbol requires four words in the symbol table, plus two words for data words associated with it. For example, an alphanumeric symbol which was being defined as "symbol, expression" at location 756 would require the basic four words plus two data words to contain the address. Therefore, this entry in the symbol table would require six words in the symbol table.

For the purpose of evaluating an expression, every symbol in the expression is given a three-word value. The first word of this value is referred to as the symbol's attribute (ATTR) and the remaining two words are referred to as the symbol's data words. The precise values these words have are defined in the following paragraphs.

In the formats and examples that follow, subscripts on symbols are used to denote bits in the values of the symbols. Bit positions are numbered from right to left (least significant to most significant bits).

Alphanumeric symbols

A symbol consists of from one to six consecutive alphanumeric characters from the character sets A-Z and 0-9. Other ASCII characters which are not standard cross-assembler characters can also be used in a symbol; e.g., <, >, {, and }. The first character of a symbol must be alphabetic. Any more than six characters in a symbol will be ignored.

Symbols may be defined in one of two ways:

1. A line of the form:

symbol, expression

The symbol will be given the value of the location into which the expression will be assembled. This method is used to assign symbolic values particular locations in the program.

When this symbol is used within an expression, its three-word value is:

ATTR=0

For example, if the symbol ABC were defined as ABC, INCB and were being assembled into location 423, its three-word value would be:

ATTR=0

2. A line of the form:

symbol=expression

The expression to the right of the equal sign will be evaluated, and the symbol will be given this value.

During evaluation, the three-word value of this symbol depends on the value of the expression. For example, if the symbol XYZ were defined as XYZ=1EF, its three-word value would be:

	ATTR=0	
		<div> <div>7</div> <div>0</div> <div> <div>+</div> <div>-----</div> <div>+</div> </div> <div> <div> </div> <div>0</div> <div> </div> </div> <div> <div> </div> <div>11101111</div> <div> </div> </div> <div> <div> </div> <div>00000001</div> <div> </div> </div> <div> <div>+</div> <div>-----</div> <div>+</div> </div> </div>
Data Word 1		
Data Word 2		
Data Word 3		

Instruction symbols

The set of M68CA instructions is defined by 3- and 4-character alphabetic symbols.

The value of a M68CA instruction consists of a 8-bit instruction code. The attribute of an instruction represents the number of 8-bit words required to make a complete instruction. For evaluation, the three-word value of an instruction would be:

	ATTR=length of the instruction	
		<div> <div>7</div> <div>0</div> <div> <div>+</div> <div>-----</div> <div>+</div> </div> <div> <div> </div> <div>instruction code</div> <div> </div> </div> <div> <div> </div> <div>0</div> <div> </div> </div> <div> <div> </div> <div>0</div> <div> </div> </div> <div> <div>+</div> <div>-----</div> <div>+</div> </div> </div>
Data Word 1		
Data Word 2		
Data Word 3		

For example:

The instruction BEQ has an instruction code of 27 and requires two words to complete the instruction; the instruction code and the immediate value. Its three-word value would be:

	ATTR=2	
		<div> <div>7</div> <div>0</div> <div> <div>+</div> <div>-----</div> <div>+</div> </div> <div> <div> </div> <div>00100111</div> <div> </div> </div> <div> <div> </div> <div>0</div> <div> </div> </div> <div> <div> </div> <div>0</div> <div> </div> </div> <div> <div>+</div> <div>-----</div> <div>+</div> </div> </div>
Data Word 1		
Data Word 2		
Data Word 3		

Addressing modes

The set of M68CA addressing modes are defined by three-character alphabetic symbols.

The value of a M68CA addressing mode symbol consists of an 8-bit code and is placed in bits 5 and 4 of Data Word 1 for evaluation

ATTR=0

Data Word 1
Data Word 2
Data Word 3

For example:

The extended mode has a code of 00110000; its three-word value would be:

ATTR=0

Data Word 1
Data Word 2
Data Word 3

Numbers

A number is represented by one to six hex digits whose value does not exceed 16 bits in length (i.e., less than 10000).

Numbers do not have symbol table entries but may be included as part of or all of an expression defining an alphanumeric symbol.

When a number appears in an expression, it has the following word configuration for the purpose of evaluating the expression:

ATTR=0

Data Word 1
Data Word 2
Data Word 3

Examples are shown later in this section.

Operators

The characters "minus" (-), "apostrophe" ('), "at" (@), "quote" ("), and "pound" (#) are the set of operators and are used to change the value of a symbol during evaluation.

Minus (-)

The operator - negates the 16-bit value of the symbol defined by data words 2 and 3. ATTR is not affected.

Apostrophe (')

The operator ' reverses the second and third data words of the value of the symbol. ATTR is not affected.

At (@)

The operator @ causes the value of the symbol to be configured as below:

The effect is to use the least significant 4 bits as an attribute and to place the next 8 most significant bits in the first data word of the configured value.

This operator is used primarily to set up instruction symbols in the symbol table. For example, the instruction extended jump has an attribute of 3 and an instruction code of 7E. To set up JMPE (redefined JMP EXT) in the symbol table, the following statement would be used:

JMPE=@7E3

Whenever JMPE appears in an expression, its value would be:

	7		0		ATTR=3
	+-----+				
@7E3=		01111110			Data Word 1
		0			Data Word 2
		0			Data Word 3
	+-----+				

Quote (")

The operator " is used to convert a single ASCII character to its hex equivalent. The parity bit (eighth level bit) is always 0. The character converted is the character immediately to the right of the operator. The value is configured as shown below:

	7		0		ATTR=0
	+-----+				
"A=		0			Data Word 1
		01000001			Data Word 2
		0			Data Word 3
	+-----+				

This operator is used primarily to provide the numeric values of ASCII characters for compare functions and load immediate functions. For example, if the user wishes to put the ASCII equivalent of an "E" in the A Accumulator, the following instruction would be used:

LDAA IMS "E

This is equivalent to:

LDAA IMS 45

By adding 80, the parity bit can be set:

LDAA IMS "E+80

Pound (#)

The # operator is used to generate numeric data (either one byte or two bytes) in the program.

The single byte is configured as shown:

The two-byte data is configured by following the least significant digit with a "T".

The data operator # should not be used in place of a simple numeric as part of an expression, because inconsistent results will be generated; e.g.,

LDAA IMS 10 is NOT equivalent to
LDAA IMS #10

The first expression has an attribute of 2, the latter of 3.

Connectives

The characters plus(+), TAB, and SPACE comprise the set of formal connectives. The operators described in the preceding paragraphs also serve as connectives. The connectives are equivalent and operate on two values, resulting in a new value. The new value is determined as follows:

The result of A<connective>W is:

ATTR=avw	(a) inclusive OR (w)
+-----+	
bvx	(b) inclusive OR (x)
c+y	(c) plus (y)
d+z	(d) plus (z)
+-----+	

If an overflow occurs when c plus y is calculated, the result of d plus z is incremented.

If an operator is used as a connective, it is applied to the symbol prior to the connective operation. For example to evaluate the instruction,

JMP EXT OFC00

JMP has the value of :

ATTR=2
+-----+
01001110
0
0
+-----+

EXT has the value of :

ATTR=1
+-----+
0110000
0
0
+-----+

The number OFC00 has the value of:

ATTR=0
+-----+
0
00000000
11111100
+-----+

The result of JMP EXT OFC00

ATTR=2	v	1	v	0	=	3
01001110	v	00110000	v	0	=	01111110
0	+	0	+	0	=	0
0	+	0	+	11111100	=	11111100

Thus, the assembled expression output would have the three-word value

7E
FC
00

The ATTR is the attribute and is only used to determine the number of words to output. If the attribute were 2, the third data word is omitted during the output of the object file and the assembly listing file. Since the attribute is 3, all data words are output in the order #1, #3, #2.

Expressions

Each line of the source program constitutes an "expression", a group of symbols with operators joined together by connectives and governed by the rules for forming expressions. A line is terminated by the characters "carriage return" (CR), or "form feed" (FF).

Expression evaluation examples

- a. Evaluate the expression LDAA IMS 7A

The instruction LDAA has a value of:

	ATTR=1	
LDAA=	10000110	(instruction code)
	0	
	0	

The addressing mode IMS has a value of

	ATTR=0	
IMS=	00000000	
	0	
	0	

The number 7A has a value of

7A=

ATTR=0			
		0	
		01111010	
		0	

Therefore the expression =

ATTR=1	v	0	v	0	=	1
10000110	v	0	v	0	=	10000110
0	+	0	+	01111010	=	01111010
0	+	0	+	0	=	0

Since the attribute is 2, the assembled expression output would be the two words:

86
7A.

b. Evaluate the expression STAB DIR 'XYZ

The instruction STAB has a value of:

STAB=

ATTR=2			
		11000111	
		0	
		0	

The addressing mode DIR has a value of

DIR=

ATTR=0			
		00010000	
		0	
		0	

Assume XYZ has the following value:

ATTR=0			
		0	
		(XYZ)bits 7-0	
		(XYZ)bits 15-8	

Low-order
High-order

ATTR=0

then		0		
'XYZ=		(XYZ)bits 15-8		High-order
		(XYZ)bits 7-0		Low-order

Therefore the expression =

ATTR=2	v	ATTR=0	v	0	=	2
11000111	v	00010000	v		=	
0		0		0	=	11010111
0		0		(XYZ)bits 15-8	=	(XYZ)bits 15-8
				(XYZ)bits 7-0	=	(XYZ)bits 7-0

This expression will be output in two 8-bit words to the output files; the first word will be the code for STAB DIR and the second word will be the high order byte of XYZ. The third word is dropped since the attribute is 2.

c. Evaluate the expression LDX IMD XYZ+3

The instruction LDX has a value of:

ATTR=2

LDX=		11001110	
		0	
		0	

The addressing mode IMD has a value of:

ATTR=1

IMD=		0	
		0	
		0	

Assume XYZ has the following value:

ATTR=0

XYZ=		0	
		(XYZ)bits 7-0	
		(XYZ)bits 15-8	

And the numeric +3 has a value of:

```

ATTR=0
+3=
+-----+
|    0    |
|00000011|
|    0    |
+-----+

```

Therefore the intermediate value of LDX IMD XYZ is

```

ATTR=2      v  ATTR=1      v      0      =      3
+-----+ +-----+ +-----+ +-----+
|11001110| v |    0    | v |    0    | = | 11001110 |
|    0    | v |    0    | + |(XYZ)bits 7-0| = |(XYZ)bits 7-0 |
|    0    | v |    0    | + |(XYZ)bits 15-8| = |(XYZ)bits 15-8|
+-----+ +-----+ +-----+ +-----+

```

With the numeric 3, the expression =

```

ATTR=3      v      0      =      3
+-----+ +-----+ +-----+ +-----+
| 11001110 | v |    0    | = | 11001110 |
|(XYZ)bits 7-0| + |    3    | = |(XYZ+3)bits 7-0 |
|(XYZ)bits 15-8| + |    0    | = |(XYZ+c)bits 15-8|
+-----+ +-----+ +-----+ +-----+

```

c=Carry from low-order sum

This expression will output to the output files three 8-bit words; the first word will be the code for a load index immediate, the second word will be the high-order byte, and the third word will be the low-order byte of XYZ+3.

M68CA CROSS-ASSEMBLER DESIGN

PROGRAM STRUCTURE

The structure of M68CA is patterned after the L Series cross-assembler written by Control Logic, Inc. This was done to enable convenient use and maintenance of a family of four cross-assemblers in use at the Savannah River Laboratory.

The structure diagrams help to visualize the organizational concepts of the M68CA cross-assembler. Each module is shown with its immediate service module. Beginning with the first diagram, hierarchy is illustrated in descending rank on succeeding pages.

THE M68CA CROSS-ASSEMBLER PROGRAM STRUCTURE

FIGURE 1

Symbol Table Generation Structure

FIGURE 2

Binary Tape Generation Structure

FIGURE 3

Listing Generation
Structure

FIGURE 4

User Symbol Table Listing Generation
Structure

FIGURE 5

Evaluation Modules
Structure

FIGURE 6

Service Modules
Structure

FIGURE 7a

Service Modules
Structure

FIGURE 7b

Service Modules
Structure

FIGURE 7c

Service Modules
Structure

FIGURE 7d

PROGRAM FLOWCHARTS

Flowcharts for Pass 0, Pass 1, Pass 2, Pass 3, and Pass 4 indicate the general program operation.

These charts give a more detailed accounting of the major program operation and can be used along with the program listing to understand the processing involved in assembling MC6800 programs.

Program labels are placed in the flowcharts to aid in correlating the program listing with the flowcharts.

Pass 0
Flowchart

FIGURE 8

Pass 1 Flowchart

FIGURE 9a

Pass 1 Flowchart

FIGURE 9b

Pass 1 Flowchart

FIGURE 9c

FIGURE 9d

Pass 1
Flowchart

FIGURE 9e

Pass 2 Flowchart

FIGURE 10a

Pass 2
Flowchart

FIGURE 10b

Pass 2
Flowchart

FIGURE 10c

Pass 2 Flowchart

FIGURE 10d

Pass 2 Flowchart

FIGURE 10e

Pass 2 Flowchart

FIGURE 10f

Pass 3 Flowchart

FIGURE 11a

Pass 3 Flowchart

FIGURE 11b

Pass 3 Flowchart

FIGURE 11c

Pass 3 Flowchart

FIGURE 11d

Pass 3 Flowchart

FIGURE 11e

Pass 3 Flowchart

FIGURE 11f

Pass 3
Flowchart

FIGURE 11g

Pass 4
Flowchart

FIGURE 12a

Pass 4
Flowchart

FIGURE 12b

MODULE FUNCTIONS

P0 - PASS 0

P0 initializes the I/O device flags, sets up the I/O device and format vectors, and calls the appropriate passes in order.

P0 Call:

P0 is entered by starting at location 200.

P1 - PASS 1

P1 evaluates the program's symbols and places them in the symbol table. If a symbol is being used as a label, it will use the location counter as the value of the symbol. If a symbol is being defined through the use of an equal sign, the expression to the right of the equal sign is evaluated and the resulting value is used in the symbol table.

P1 Call

P1 is entered via a JMP P1 from the P0 and returns to PASS2 in Pass 0.

P2 - PASS 2

P2 evaluates each instruction and punches the result for the binary tape in the Control Logic binary format.

P2 Call

P2 is entered via a JMP P2 and returns to PASS3 in Pass 0.

P3 - PASS 3

P3 evaluates each instruction and prints the result in the following format: location counter (2 double hex characters), attribute number of words of the instruction (each word is 2 hex characters), and the source line.

P3 Call

P3 is entered via a JMP P3 and returns to FINISH in Pass 0.

EVAL - EVALUATE LINE

EVAL evaluates a complete source line and places the three-word result in VALUE. It extracts one word at a time from the source line, adjusts the word's value according to the word's operator and ORs the first word of the accumulated value of the line, adds the second words of each, and also adds the third words.

EVAL Call

JMS EVAL
<error return>
<normal return>

EVSY - EVALUATE SYMBOL

EVSY evaluates the six-character symbol input in the following format in EVWBF:

+-----+		Terminators:
Terminator		0 Normal
Sign		1 =
1st Char. 2nd Char.		2 ,
3rd Char. 4th Char.		3 *
5th Char. 6th Char.		
+-----+		
		Signs:
		0 +
		1 -
		2 @
		3 ' _ _ _

and places the three-word result in EVVAL. If the symbol is numeric, it will convert it to two 8-bit binary words and place them right-justified, zero-filled in EVVAL+1 (low order) and EVVAL+2 (high order). If the symbol is a variable, it will retrieve its value from the symbol table. Before returning, the value in EVVAL is adjusted according to its sign.

EVSY Call

JMS EVSY
<error return>
<normal return>

GNL - GET NEXT LINE

GNL reads the next source line from the specified reader and places it, one character per word, in the source line buffer.

GNL Call

JMS GNL

GNW - GET NEXT WORD

GNW retrieves the next word in the specified buffer and places it and its terminator and sign in the specified buffer in the following format:

+-----+	
Terminator	
Sign	
1st Char.	2nd Char.
3rd Char.	4th Char.
5th Char.	6th Char.
+-----+	

Terminators:

0	Normal
1	=
2	,
3	*

Signs:

0	+
1	-
2	@
3	'

If the word is less than six characters, the remaining locations are blank-filled. If the word is greater than six characters, all characters after the sixth are ignored. If there is no next word, the terminator is set to -1. GNW increments the second call argument for its next entry before returning.

GNW Call

JMS GNW

<address of 5-word buffer to receive word>
<address of search starting point>

STE - SYMBOL TABLE ENTER

STE enters a symbol and its value into the next available entry in the symbol table if there is room. Each symbol table entry consists of the following six words:

1st Char.	2nd Char.
3rd Char.	4th Char.
5th Char.	6th Char.
VALUE	
VALUE	
VALUE	

STE Call

JMS STE
<address of packed symbol>
<address of three-word value>

STS - SYMBOL TABLE SEARCH

STS searches the symbol table for the specified symbol. If it finds the symbol, it sets AC equal to the address of the value. If it does not find the symbol, it sets AC = 0.

STS Call

JMS STS
<address of packed symbol>

WC - WORD COMPARE

WC compares two three-word long arrays. If they match, it sets AC = 0; if they do not match, it sets AC not equal 0.

WC Call

JMS WC
<address of 1st array>
<address of 2nd array>

WCC - SINGLE WORD COMPARE

WCC compares the word whose address is in WC1A with the word whose address is in WC2A. If the words match, it sets AC = 0 and returns via a JMP I WC; if they do not match, it sets AC not equal 0 and returns via a JMP I WC.

WCC Call

JMS WCC

SL - SCAN LINE

SL scans the source line buffer for the 8-bit character in the AC. If a match is found, AC is set to the address of the matched character; if no match is found, AC is not equal 0.

SL Call

JMS SL

GNWPK - GET NEXT WORD PACK

GNWPK is entered with location 17 equal to the address-1 of the two-word array to be packed and location 15 equal to the address-1 of the word where the packed characters are to be stored. GNWPK packs the two characters into one word, as follows:

1st Char.	2nd Char.
-----------	-----------

GNWPK Call

JMS GNWPK

BTAF - BINARY TO ASCII - 4 DIGITS

BTAF takes a three-word value, configures the bits as follows:

```
+-----+
|Word 3, bits 4-11 |Word 2, bits 4-11 |
+-----+
```

and then converts these 16 bits into 4 ASCII hex digits and places them, one per word, right-justified, zero-filled, high-order digit first, in the specified five-word array.

BTAF Call

JMS BTAF

<address of 5-word array>

<address of value>

ATB - ASCII TO BINARY

ATB converts a packed ASCII hex number whose maximum length is six characters into a 16-bit binary number and stores the value in a three-word specified array in the following format:

```
+-----+
|               |
|               |low-order 8 bits
|               |high-order 8 bits
|               |
+-----+
```

ATB Call

JMS ATB

<address of packed ASCII number>

<address of 3-word value array>

BTAS - BINARY TO ASCII - ATTRIBUTE NUMBER OF WORDS

BTAS assumes the three-word value it is to convert has the following format:

It then converts the attribute number of 8-bit data words into 2-digit ASCII hex numbers and stores them in the specified array, one digit per word, with a space between numbers.

BTAS Call

JMS BTAS

<address of array>

<address of 3-word value>

DNEG - DOUBLE PRECISION NEGATE

DNEG takes a 3-word value, configures the bits as follows:

then negates this configuration and replaces the original 8 bits in words 2 and 3 with the negated result.

DNEG Call

JMS DNEG

<address of 3-word value>

OCHN - ORDER CHANGE

OCHN takes a three-word value and swaps words 2 and 3.

OCHN Call

JMS OCHN
<address of value>

ATIN - ATTRIBUTE AND INSTRUCTION SET

ATIN takes a three-word value and moves the bits as follows:

- a. Bits 9-11 of word 2 moved to bits 0-2, word 1.
- b. Bit 3 of word 1=0.
- c. Bits 9-11 of word 3 moved to bits 4-6, word 1.
- d. Bits 4-8 of word 2 moved to bits 7-11, word 1.
- e. Word 2=0.
- f. Word 3=0.

ATIN Call

JMS ATIN
<address of value>

DIOR - INCLUSIVE OR

DIOR inclusively ORs the first words of two three-word values and places the result in the first word of the first value.

DIOR Call

JMS DIOR

<address of 1st value>

<address of 2nd value>

DADD - DOUBLE PRECISION ADD

DADD takes two three-word values and adds them in the following manner:

- a. Bits 4-11 of their second words, result stored in word 2 of the first value.
- b. Bits 4-11 of their third words, result stored in word 3 of the first value.
- c. If the addition of the second words causes an overflow, the result of the addition of their third words is incremented.

DADD Call

JMS DADD

<address of first value>

<address of second value>

LCI - LOCATION COUNTER INCREMENT

LCI increments the location counter (in both fields) by the amount found in bits 0-2 of the value.

LCI Call

JMS LCI

<address of value>

PC - PAGE CONTROL

PC controls the paging of printout. It allows 60 lines of text per page, and 6 blank lines between pages.

PC Call

JMS PC

PE - PAGE EJECT

PE ejects the present page of output.

PE Call

JMS PE

GP - GENERAL PRINTER

GP prints the specified number of characters from the specified buffer where the characters are stored right-justified, zero-filled.

GP Call

JMS GP

<address of buffer>

<negative length of buffer>

SLP - SOURCE LINE PRINTER

SLP prints the source line which is in the source line buffer.

SLP Call

JMS SLP

LLP - LISTING LINEPRINTER

LLP prints a listing line which consists of:

- a. location counter
- b. assembled code
- c. source line

LLP Call

JMS LLP

DMP - DIAGNOSTIC MESSAGE PRINTER

DMP prints the requested diagnostic message, whose number is in AC, in the following format:

<msg. code> AT <location>

DMP Call

JMS DMP

LTP - LEADER-TRAILER PUNCH

LTP punches the specified character the specified number of times.

LTP Call

JMS LTP

<negative number of characters>

<punch character>

BP - BINARY PUNCH

BP is entered with the type of block to be punched in the AC.

AC=0, ORIGIN BLOCK

AC=-1, END BLOCK

AC=+1, DATA BLOCK

For an origin block, it punches the buffer and sets the location counter into the starting address for the next set of data.

For an end block, it punches the buffer followed by the tape terminate block.

For a data block, it stores the data in the buffer.

BP Call

JMS BP

BBP - BINARY BUFFER PUNCH

BBP punches the start character, the starting address of the data, the number of data words, the data in the buffer, followed by the checksum. It then resets the buffer address so new data may be stored.

BBP Call

JMS BBP

MEMORY MAP

The memory is divided into several regions. The actual cross-assembler code occupies a large part of the first field. Variables and temporary buffers for the program code occupy the first page of the first and second field. These variables are essentially identical and facilitate access to them depending on where the program is actually operating on data. The OS/8 handlers reside in the second field as do the symbol tables. The layout of the memory map is as follows:

Location	General contents
20- 140	Program variables, buffers
144- 177	Constants
200- 5777	Program code
10020-10140	Program variables, buffers
12000-13177	OS/8 handlers
13200-14445	Permanent symbol table
14446-17775	User symbol table

CONCLUSIONS

This cross-assembler provides rapid assembly of MC6800 microprocessor assembly language programs. Large programs that take typically 5 minutes to assemble on a PDP 8/E disk system would theoretically take as much as 6 hours to assemble on a paper tape system in an M6800 system using a Teletype. The operation of the M68CA cross-assembler is straightforward, and documentation provides information to enable a programmer to maintain and modify the cross-assembler.

ACKNOWLEDGEMENT

The material in this report was accepted in partial fulfillment of the requirements for the degree of Master of Science in Electrical Engineering at the University of South Carolina, Columbia, South Carolina in May 1978.

REFERENCES

- L Series Cross Assembler (PDP 8 version)
Control Logic Incorporated, October 1974
- L Series Cross Assembler (OS/8 version)
University of South Carolina, July 1975
- M Series Cross Assembler (OS/8 version)
E I Du Pont De Nemours & Co Inc
Savannah River Laboratory, September 1975
- M6800 Microprocessor Programming Manual
Motorola Semiconductor Products Inc, 1975
- M6800 Microprocessor Applications Manual
Motorola Semiconductor Products Inc, 1975

APPENDIX A - EXAMPLE PROGRAMS

```

/
/      BUBBLESORT PROGRAM ASSEMBLED BY M68CA
/      RJS      11 NOV 77
/
02 00      02 00      *200
/
02 00      8E 01 00      X10,      LDS IMD 100      /INITIALIZE STACK
02 03      BD 02 09      JSR EXT X20      /SORT DATA
02 06      7E FC 00      JMP EXT OFC00      /RETURN TO MONITOR
/
02 09      B6 02 57      X20,      LDAA EXT I+1      /GET I
02 0C      B7 02 5A      STAA EXT J+1      /PUT I IN J
02 0F      FE 02 56      LDX EXT I      /PUT I IN INDEX
02 12      09      DEX      /INDEX-1
02 13      E6 4C      LDAB IND DATA      /GET DATA
02 15      7A 02 5A      X30,      DEC EXT J+1      /J-1
02 18      B6 02 5A      LDAA EXT J+1      /GET J
02 1B      81 00      CMPA IMS 0      /IS J ZERO?
02 1D      27 19      BEQ X80      /YES
02 1F      FE 02 59      X40,      LDX EXT J      /PUT J IN INDEX
02 22      A6 4C      LDAA IND DATA      /GET DATA
02 24      B7 02 58      STAA EXT TEMP      /PUT THIS DATA IN TEMP
02 27      F1 02 58      X50,      CMPB EXT TEMP      /IS B GREATER THAN TEMP?
02 2A      2A 09      BPL X70      /YES
02 2C      FE 02 59      X60,      LDX EXT J      /PUT J IN INDEX
02 2F      09      DEX      /INDEX-1
02 30      E7 4C      STAB IND DATA      /PUT LARGE NUMBER IN DATA
02 32      F6 02 58      LDAB EXT TEMP      /PUT SMALL NUMBER IN B
02 35      7E 02 15      X70,      JMP EXT X30      /NEXT J
02 38      FE 02 56      X80,      LDX EXT I      /PUT I IN INDEX
02 3B      09      DEX      /INDEX-1
02 3C      E7 4C      STAB IND DATA      /GET NEW DATA
02 3E      7A 02 56      DEC EXT I      /I-1
02 41      B6 02 56      LDAA EXT I      /GET I
02 44      81 00      CMPA IMS 0      /IS I ZERO?
02 46      27 03      BEQ X100      /YES, DATA SORTED
02 48      7E 02 09      X90,      JMP EXT X20      /NOT YET
02 4B      39      X100,      RTS
/
/      VARIABLES AND CONSTANTS
/
02 4C      24      DATA,      @1 24      /DATA TO SORT
02 4D      10      @1 10
02 4E      AB      @1 0AB
02 4F      49      @1 49
02 50      07      @1 7
02 51      AC      @1 0AC
02 52      15      @1 15
02 53      68      @1 68

```

02 54 35
 02 55 96
 02 56 00 0A
 02 58 00
 02 59 00 00

I,
 TEMP,
 J,
 \$

@1 35
 @1 96
 @2 0A
 @1 0
 @2 0

/NUMBER OF DATA WORDS

X10 0 00 02 00
 X20 0 00 02 09
 X30 0 00 02 15
 X40 0 00 02 1F
 X50 0 00 02 27
 X60 0 00 02 2C
 X70 0 00 02 35
 X80 0 00 02 38
 X90 0 00 02 48
 X100 0 00 02 48
 DATA 0 00 02 4C
 I 0 00 02 56
 TEMP 0 00 02 58
 J 0 00 02 59


```

/
/      SEARCH DATA FOR CHARACTER
/      ASSEMBLED BY M68CA
/      RJS      11 NOV 77
/
ODT=0FC00      /MONITOR LOCATION
*200
02 00 FC 00
02 00 02 00
02 00 CE 02 15      LDX IMD DATAS      /INIT INDEX TO 1ST SPOT
02 03 B6 02 20      LDAA EXT SC      /GET SEARCH CHARACTER
02 06 A1 00      FNDIT, CMPA IND 0      /DOES SC=DATA BYTE
02 08 27 08      BEQ DNE      /YES
02 0A 08      INX      /NEXT SPOT
02 08 8C 02 1F      CPX IMD DATAE      /END OF DATA YET?
02 0E 2E 02      BGT DNE      /YES
02 10 20 F4      BRA FNDIT      /TRY NEXT SPOT
02 12 7E FC 00      DNE, JMP EXT ODT      /RETURN TO MONITOR
/
02 15 41      DATAS, TEXT 0A ABCDEFGHIJ
02 16 42
02 17 43
02 18 44
02 19 45
02 1A 46
02 1B 47
02 1C 48
02 1D 49
02 1E 4A
02 1F 4B      DATAE, TEXT 1 K
02 20 46      SC, TEXT 1 F
$

```

```

ODT      0 00 FC 00
FNDIT    0 00 02 06
DNE      0 00 02 12
DATAS    0 00 02 15
DATAE    0 00 02 1F
SC       0 00 02 20

```

EXAMPLE OF AN ASSEMBLED
PROGRAM (with errors)

```
.R M68CA
*USCAPB,USCAPB<USCAPB.S
*UI AT FC07*
*MT AT FD08*
*IN AT FD0A*
*IN AT FD0A*
*UI AT FC07*
*BE AT FD06*
*IN AT FD0A*
*IN AT FD0A*
*UI AT FD37*
*UO AT FD39*
*UO AT FD39*
*UI AT FC07*
*BE AT FD06*
*IN AT FD0A*
*IN AT FD06*
*UI AT FD37*
*UO AT FD39*
*UO AT FD39*
```

```
R PIP
*TTY:<USCAPB.LS
```

		/		
		/	EXAMPLE PROGRAM	ASSEMBLED BY M68CA
		/	RJS	9 SEPT 77
		/		
	C0 06	ACIACS=0C006		/CONTROL STATUS
	C0 07	ACIADA=ACIACS+1		/DATA
		/		
	F8 00	*0F800		/---RAM---
F8 00	00 00	VAR1, @2 0		/2 BYTE VARIABLE
F8 02	00	VAR2, @1 0		/1 BYTE VARIABLE
F8 03	00	VAR3, #0		/1 BYTE VARIABLE
F8 04	00 00	VAR4, #0T		/2 BYTE VARIABLE
	F8 26	*.+20		/MAKE ROOM FOR STACK
F8 26	00	STACK, @1 0		/PROGRAM STACK
	FC 00	*0FC00		/---EPROM---
		/		
FC 00	8E F8 26	LABEL, LDS IMD STACK		/INITIALIZE STACK POINTER
FC 03	86 00	LDAA IMS VAR1		/LOW ORDER BYTE OF VAR1
FC 05	C6 F8	LDAB IMS 'VAR1		/HIGH ORDER BYTE OF VAR1
UI AT FC07				
FC 07	FF 00 00	LDA A IMS VAR1		/SPACE NOT ALLOWED
FC 0A	81 52	CMPA IMS "R		/NUMERIC FOR ASCII
FC 0C	7E FD 06	JMP EXT CONTIN		
FC 0F	4D	MSG, TEXT 7 MESSAGE		/MESSAGE
FC 10	45			
FC 11	53			
FC 12	53			
FC 13	41			
FC 14	47			
FC 15	45			
	FD 06	*.+0F0		
BE AT FD06				
FD 06	27 00	CONTIN, BEQ LABEL		/BRANCH TOO FAR BACKWARDS
FD 08	86 AA	VAR2, LDAA IMS 0AA		/VAR2 MULTIPLY DEFINED
IN AT FD0A				
IN AT FD0A				
FD 0A	41	TEXT 0X APPLE		/X NOT A NUMBER
FD 0B	50			
FD 0C	50			
FD 0D	4C			
FD 0E	45			
FD 0F	09			
FD 10	3B			
FD 11	58			
FD 12	20			
FD 13	4E			
FD 14	4F			
FD 15	54			
FD 16	20			
FD 17	41			
FD 18	20			
FD 19	4E			
FD 1A	55			
FD 1B	4D			

```

FD 1C 42
FD 1D 45
FD 1E 52
FD 1F 0D
FD 20 0A
FD 21 59
FD 22 20
FD 23 44
FD 24 45
FD 25 46
FD 26 49
FD 27 4E
FD 28 45
FD 29 44
FD 2A 0D
FD 2B 0A
FD 2C 0D
FD 2D 0A
FD 2E 52
FD 2F 0D
FD 30 0A
FD 31 20
FD 32 20
FD 33 20
FD 34 20
FD 35 20
FD 36 20
*UI AT FD37*
FD 37 86 00

```

```

*UO AT FD39*
      00 00
*UO AT FD39*
      00 00

```

```

/          LDAA IMS APPLE  /APPLE NOT A SYMBOL
/
*FE00          /NUMBERS START WITH 0-9
*APPLE        /UNDEFINED SYMBOL
/
/
$

```

```

ACIACS 0 00 C0 06
ACIADA 0 00 C0 07
VAR1    0 00 F8 00
VAR2    0 00 F8 02
VAR3    0 00 F8 03
VAR4    0 00 F8 04
STACK   0 00 F8 26
LABEL   0 00 FC 00
MSG      0 00 FC 0F
CONTIN  0 00 FD 06

```

*

APPENDIX B - BINARY TAPE FORMAT

The binary formatted output of the cross-assembly program represents each byte of assembled instructions and data in a single frame of punched tape.

The assembled data is divided into "blocks". The leader and trailer are blank. The format of a block is shown below:

The checksum is the sum modulo 256 of data bytes. The end of the binary tape is denoted by BF hex.

APPENDIX C - STANDARD SYMBOLS

<u>MNEMONIC</u>	<u>CODE</u>	<u>ATTRIBUTE</u>
INHERENT ADDRESSING		
NOP	01	1
TAP	06	1
TPA	07	1
INX	08	1
DEX	09	1
CLV	0A	1
SEV	0B	1
CLC	0C	1
SEC	0D	1
CLI	0E	1
SEI	0F	1
SBA	10	1
CBA	11	1
TAB	16	1
TBA	17	1
DAA	19	1
ABA	1B	1
BRA	20	2
BHI	22	2
BLS	23	2
BCC	24	2
BCS	25	2
BNE	26	2
BEQ	27	2
BVC	28	2
BVS	29	2
BPL	2A	2
BMI	2B	2
BGE	2C	2
BLT	2D	2
BGT	2E	2
BLE	2F	2
TSX	30	1
INS	31	1
PULA	32	1
PULB	33	1
DES	34	1
TXS	35	1
PSHA	36	1
PSHB	37	1
RTS	39	1
RTI	3B	1
WAI	3E	1
SWI	3F	1

NEGA	40	1
COMA	43	1
LSRA	44	1
RORA	46	1
ASRA	47	1
ASLA	48	1
ROLA	49	1
DECA	4A	1
INCA	4C	1
TSTA	4D	1
CLRA	4F	1
NEGB	50	1
COMB	53	1
LSRB	54	1
RORB	56	1
ASRB	57	1
ASLB	58	1
ROLB	59	1
DECB	5A	1
INCB	5C	1
TSTB	5D	1
CLRB	5F	1

ADDRESSING MODES

IMS	00	0
DIR	10	0
IND	20	0
EXT	30	1
IMD	00	1

TWO PART INSTRUCTIONS

NEG	40	2
COM	43	2
LSR	44	2
ROR	46	2
ASR	47	2
ASL	48	2
ROL	49	2
DEC	4A	2
INC	4C	2
TST	4D	2
JMP	4E	2
CLR	4F	2
SUBA	80	2
CMPA	81	2
SBCA	82	2
ANDA	84	2
BITA	85	2
LDAA	86	2
STAA	87	2
EORA	88	2
ADCA	89	2

ORAA	8A	2
ADAA	8B	2
CPX	8C	2
JSR	8D	2
BSR	8D	2
LDS	8E	2
STS	8F	2
SUBB	C0	2
CMPB	C1	2
SBCB	C2	2
ANDB	C4	2
BITB	C5	2
LDAB	C6	2
STAB	C7	2
EORB	C8	2
ADCB	C9	2
ORAB	CA	2
ADDB	CB	2
LDX	CE	2
STX	CF	2