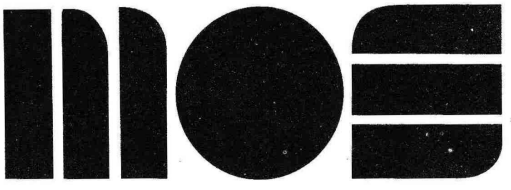


MICROCOMPUTERS

CROSS ASSEMBLER MANUAL
PRELIMINARY

Publication Number 6500-60P

MCS6500

MICROCOMPUTER FAMILY

CROSS ASSEMBLER MANUAL

PRELIMINARY

AUGUST 1975

First Edition

MOS TECHNOLOGY, INC.
950 Rittenhouse Road
Norristown, PA. 19401

TABLE OF CONTENTS

I. INTRODUCTION

This manual describes the assembly language and assembly process for programs for the MCS-650X series of microprocessors. Several assemblers are available for program development and while they are all slightly different in detail of use they are essentially the same in substance.

The process of translating a mnemonic or symbolic form of a computer program to actual machine code is called an assembly, and a program which performs the translation is an assembler. The symbols used and rules of association for those symbols are the assembly language. In general one assembly language statement will translate into one machine instruction. This distinguishes an assembler from a compiler which may produce many machine instructions from a single statement. An assembler which executes on a computer other than the one for which code is generated is called a cross-assembler. Use of cross-assemblers for program development for microprocessors is common since often a microcomputer system has fewer resources than are needed for an assembler.

Normally digital computers use the binary number system for representation of data and instructions. Computers understand only ones and zeroes corresponding to an "on" or "off" state. Human users on the other hand find it difficult to work with the binary number system and hence use a more convenient representation such as octal (base 8), decimal (base 10), or hexadecimal (base 16). Two representations of the MCS-650X operation to "load" information into an "accumulator" are shown below:

```
10101001   (binary)
   A9      (hexadecimal)
```

An instruction to move the value 21 (decimal) to the accumulator is:

A9 15 (hexadecimal)

Users still find numeric representations of instructions tedious to work with and hence have developed symbolic representations. For example the preceeding instruction might be written as:

LDA #21

In this case LDA is a symbol for A9, Load the Accumulator. A computer program used to translate the symbolic form LDA to numeric form A9 is called an assembler. The symbolic program is referred to as source code and the numeric program is the object code. Only object code can be executed on the processor.

Each machine instruction to be executed has a symbolic name referred to as an opcode (operation code). The opcode for "store the accumulator" is STA. The opcode for "transfer accumulator to index X" is TAX. There are 55 opcodes for the MCS-650X processors (listed in section II). A machine instruction in assembly language consists of an opcode and perhaps operands which specify the data on which the operation is to be performed.

Instructions may be labelled for reference by other instructions as shown in

L2 LDA #12

The label is L2, the opcode is LDA, and the operand is #12. At least one blank must separate the three parts (fields) of the instruction. Additional blanks may be inserted for ease of reading. Instructions for the MCS-650X processors have at most one operand and many have none. In these cases the operation to be performed is completely specified by the opcode as in CLC (Clear the Carry Bit).

Programming in assembly language requires learning the instruction set (opcodes), addressing conventions for referencing data, the data structures within the processor, as well as the structure of assembly language programs.

II. INSTRUCTION FORMAT

Assembler instructions for the MCS-650X are of two basic types according to function:

1. Machine instructions
2. Assembler directives

Machine instructions correspond to the 55 operations implemented on the MCS-650X processors. The instruction format is:

(label) opcode (operands) (comments)

Fields are bracketed to show that they are optional. Labels and comments are always optional and many operation codes (opcodes) such as RTS (Return from Subroutine) do not require operands. A typical instruction showing all four fields is:

```
LOOP    LDA    BETA,X    FETCH BETA INDEXED BY X
```

A field is defined as a string of characters separated by a blank space or tab character or characters. The list of opcodes for the MCS-650X processors is shown in Table 1.

A label is an alphanumeric string of from one to six characters, the first of which must be alphabetic. A label may not be any of the 55 opcodes and also may not be any of the special single characters A, S, P, X, or Y. These special characters are used by the assembler to reference the Accumulator (A), Stack pointer (S), Processor status (P), and index registers X and Y respectively. A label may begin in any column provided it is the first field of an instruction. Labels are used on instructions as branch targets and on data elements for reference in operands.

Table 1.

MCS650X MICROPROCESSOR INSTRUCTION SET - OP CODES

ADC	Add with Carry to Accumulator
AND	"AND" to Accumulator
ASL	Shift Left One Bit (Memory or Accumulator)
BCC	Branch on Carry Clear
BCS	Branch on Carry Set
BEQ	Branch on Zero Result
BIT	Test Bits in Memory with Accumulator
BMI	Branch on Result Minus
BNE	Branch on Result not Zero
BPL	Branch on Result Plus
BRK	Force an Interrupt or Break
BVC	Branch on Overflow Clear
BVS	Branch on Overflow Set
CLC	Clear Carry Flag
CLD	Clear Decimal Mode
CLI	Clear Interrupt Disable Bit
CLV	Clear Overflow Flag
CMP	Compare Memory and Accumulator
CPX	Compare Memory and Index X
CPY	Compare Memory and Index Y
DEC	Decrement Memory by One
DEX	Decrement Index X by One
DEY	Decrement Index Y by One
EØR	Exclusive-or Memory with Accumulator
INC	Increment Memory by One
INX	Increment X by One
INY	Increment Y by One
JMP	Jump to New Location
JSR	Jump to New Location Saving Return Address
LDA	Transfer Memory to Accumulator
LDX	Transfer Memory to Index X
LDY	Transfer Memory to Index Y
LSR	Shift One Bit Right (Memory or Accumulator)
NØP	Do Nothing - No Operation
ØRA	"OR" Memory with Accumulator
PHA	Push Accumulator on Stack
PHP	Push Processor Status on Stack
PLA	Pull Accumulator from Stack
PLP	Pull Processor Status from Stack
RØL	Rotate One Bit Left (Memory or Accumulator)
RTI	Return From Interrupt
RTS	Return From Subroutine
SBC	Subtract Memory and Carry from Accumulator
SEC	Set Carry Flag
SED	Set Decimal Mode
SEI	Set Interrupt Disable Status
STA	Store Accumulator in Memory
STX	Store Index X in Memory
STY	Store Index Y in Memory
TAX	Transfer Accumulator to Index X
TAY	Transfer Accumulator to Index Y
TSX	Transfer Stack Register to Index X
TXA	Transfer Index X to Accumulator
TXS	Transfer Index X to Stack Register
TYA	Transfer Index Y to Accumulator

The operands portion of an instruction specifies either an address or a value. An address may be computed by expression evaluation and the assembler allows considerable flexibility in expression formation. An assembly language expression consists of a string of names and constants separated by operators +, -, *, and / (add, subtract, multiply, and divide). Expressions are evaluated by the assembler to compute operand addresses. Expressions are evaluated left to right with no operator precedence and no parenthetical grouping. Note that expressions are evaluated at assembly time and not execution time.

Any string of characters following the operands field is considered to be comments and is listed but not further processed. If the first non-blank character of any record is a semi-colon (;) the record is processed as a comment. On instructions which require no operand, comments may follow the opcode. At least one separating character (space or horizontal tab) must separate the fields of an instruction.

There are eight assembler directives used to reserve storage and direct information to the assembler. Seven have symbolic names with a period as the first character. The eighth, a symbolic equate, uses an equals sign (=) to establish a value for a symbol. A list of the directives is given below and their use is explained in a later section.

.BYTE .WORD .DBYTE .PAGE .SKIP .OPT .END =

Labels and symbols other than directives may not begin with a period.

A typical MCS-650X assembler program segment is shown on the following page. This example is given primarily to show the form of the information output by the assembler. An annotated example is given in later sections.

```
213  076A  20 60 09  ALPHA     JSR   GETINS            FIND START OF NEXT INSTR
214  076D  A9 00               LDA     #0
215  076F  85 ID               STA     EFLAG
216  0771  85 1E               STA     DFLAG           NO DATA OR EFFECTIVE ADDR YET
217                     ; PICK UP THE OPCODE AND BREAK IT INTO ITS PARTS
218  0773  A5 14               LDA     OPCODE
219  0775  29 03               AND     #%11
220  0777  85 13               STA     GROUP           BITS 1,0 = GROUP CODE
221  0779  A5 14               LDA     OPCODE
222  077B  29 FC               AND     #%11111100
223  077D  4A                  LSR     A
224  077E  85 10               STA     B72
225  0781  AA                  TAX
227  0782  29 07               AND     #%111
228  0784  85 12               STA     B42
229  0786  8A                  TXA
230  0787  4A                  LSR     A
231  0788  4A                  LSR     A
232  0789  4A                  LSR     A
233  078A  85 11               STA     B75
234  078C  20 79 09            JSR     SETUP           GET DATA FROM IT
235                     ; SEE IF WE HAVE A LABEL TO PRINT
236  078F  AF 15               LDA     IADR
237  0791  85 27               STA     NUMBER
238  0793  AF 16               LDA     IADR+1
239  0795  85 28               STA     NUMBER+1
240  0797  20 IC 08  BETA      JSR   NUM             PRINT CURRENT P.C.
```

line number — memory address — object code — label — opcode — operand — comments

Example 1. Segment of an MCS-650X program.

Symbolic

Perhaps the most common operand addressing mode is the symbolic form as in:

```
LDA   BETA        PUT BETA VALUE IN ACCUMULATOR
```

In the example BETA references a byte in memory that is to be loaded into the accumulator. BETA is an address at which the value is located. Similarly in the instruction

```
LDA   ALPHA+BETA
```

the address ALPHA+BETA is computed by the assembler and the value at the computed address is loaded into the accumulator.

Memory associated with the MCS-650X processors is segmented into pages of 256 bytes each. The first page, page zero, is treated differently by the assembler and by the processor for optimization of memory storage space. Many of the instructions have alternate operation codes if the operand address is in page zero memory. In those cases the address requires only one byte rather than the normal two. For example if BETA is located at byte 4B in page zero memory then the code generated for

```
LDA   BETA
```

is A5 4B. This is called page zero addressing. If BETA is at 01 3C in memory page one the code generated is AD 3C 10 This is an example of absolute addressing. Thus, to optimize storage and execution time a programmer should design with data areas in page zero memory whenever possible. Note that the assembler makes decisions on which form to use based on operand address computation.

Constants

Constant values in assembly language can take several forms as needed by the programmer. If a constant is other than decimal a prefix character is used to specify type.

$	(Dollar sign) specifies hexadecimal
@	(Commercial at) specifies octal
%	(Percent) specifies binary
'	(Apostrophe) specifies an ASCII literal character in immediate instructions

The absence of a prefix symbol indicates decimal value. In the statement

```
LDA  BETA+5
```

the decimal number 5 is added to BETA to compute the address. Similarly

```
LDA   BETA+ $5F
```

denotes that the hexadecimal value 5F is to be added to BETA for the address computation.

The immediate mode of addressing is signified by a # (Pounds sign) followed by a constant. For example

```
LDA   #2
```

specifies that the decimal value 2 is to be put into the accumulator. Similarly

```
LDA   #'G
```

will load the ASCII character G into the accumulator.

Immediate mode addressing always generates two bytes of machine code, the opcode and the value to be used as operand. Note that constant values can be used in address expressions and as values in immediate mode addressing. They can also be used to initialize locations as explained in a later section on assembler directives.

Relative

There are 8 conditional branch instructions available to the user. An example is

```
BEQ   START        IF EQUAL BRANCH TO START
```

which might typically follow a compare instruction. If the values compared are equal a transfer to the instruction labelled START is made. The branch address is a one byte positive or negative offset which is added to the program counter during execution. At the time the addition is made the program counter is pointing to the next instruction beyond the branch instruction. Thus, a branch address must be within 129 bytes forward or 125 bytes backward from the conditional branch instruction. An error will be flagged at assembly time if a branch target falls outside the bounds for relative addressing. Relative addressing is not used for any other instructions.

Implied

Twenty-five instructions such as TAX (Transfer Accumulator to Index X) require no operand and hence are single byte instructions. Thus, the operand addresses are implied by the operation code.

Three instructions ASL, LSR, and ROL are special in that the accumulator, A, can be used as an operand. In this special case these three instructions are treated as implied mode addressing and only an operation code is generated.

Indexed

Operands may be indexed with values in registers X and Y. Indexing is indicated by a comma and appropriate letter following the operand. For example

```
LDA   BETA,Y
```

The value in register Y is added to BETA to form the address of the operand. Not all instructions can be indexed and on some indexing may be permitted with one register but not the other. Refer to Table 2 for allowable addressing modes.

Indexed indirect

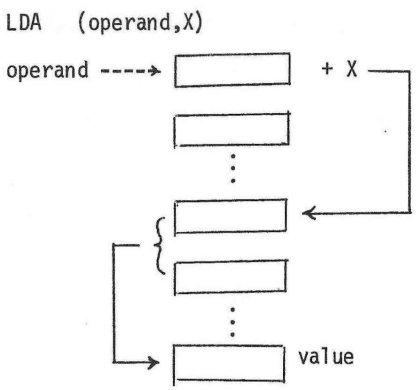
In this mode the operand address is a location in page zero memory which contains the address to be used as an operand. An example is:

LDA (BETA,X)

The parentheses around the operand indicate it is indirect mode. In the above example the value in index register X is added to BETA. That sum must reference a location in page zero memory. During execution the high order byte of the address is ignored thus forcing a page zero address. The two bytes starting at that location in page zero memory are taken as the address of the operand. For purposes of illustration assume the following:

BETA is 12
X contains 4
Locations 0017 and 0016 are 01 and 25
Location 0125 contains 37

Then BETA + X is 16, the address at location 16 is 0125. The value at 0125 is 37 and hence the instruction LDA (BETA,X) loads the value 37 into the accumulator. This form of addressing is shown in the illustration below.



Indirect indexed

Another mode of indirect addressing uses index register Y and is illustrated by:

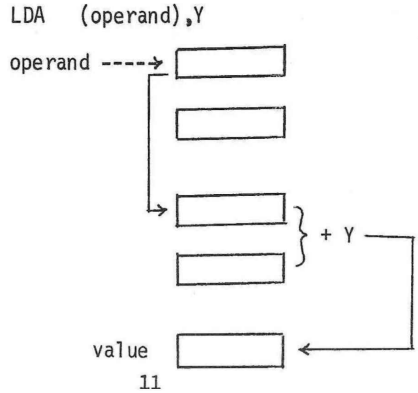
LDA (GAMMA),Y

In this case GAMMA references a page zero location at which an address is to be found. The value in index Y is added to that address to compute the actual address of the operand. Suppose for example that:

GAMMA is 38 (hexadecimal)
Y contains 7
Locations 0039 and 0038 are 00 and 54
Location 005B contains 126

Then the address at 38 is 0054 and 7 is added to this giving an effective address 005B. The value at 005B is 126 which is loaded into the accumulator.

In indexed indirect the index X is added to the operand prior to the indirection. In indirect indexed the indirection is done and then the index Y is added to compute the effective address. Indirect mode is always indexed except for a JMP instruction which allows an absolute indirect address as exemplified by JMP (DELTA) which causes a branch to the address at location DELTA. The indexed indirect mode of addressing is shown in the illustration below.



	Immediate	Page zero	Absolute	Page zero indexed by X	Absolute indexed by X	Absolute indexed by Y	Indirect indexed
ACD	x	x	x	x	x	x	x
AND	x	x	x	x	x	x	x
ASL (1)		x	x	x	x		
BIT		x	x				
CMP	x	x	x	x	x	x	x
CPY	x	x	x				
CPX (2)	x	x	x				
DEC		x	x	x	x		
EOR	x	x	x	x	x	x	x
INC		x	x	x	x		
JMP (3)			x				x
JSR			x				
LDA	x	x	x	x	x	x	x
LDX (2)	x	x	x	x	x		
LDY	x	x	x	x	x		
LSR (1)		x	x	x	x		
ORA	x	x	x	x	x	x	x
ROL (1)		x	x	x	x		
SBC	x	x	x	x	x	x	x
STA		x	x	x	x	x	x
STX		x	x	x			
STY		x	x	x			

(1) Accumulator A can also be an operand
(2) Indexing with Y
(3) Indirect is absolute indirect and not indexed

The 8 conditional branches use relative addressing.
The 25 other instructions not in this table use implied addressing.

Table 2. Instruction addressing modes.

III. ASSEMBLER DIRECTIVES

There are eight directives which are used to control the assembly process, define values or initialize memory locations. Assembler directives always appear in the opcode field of an instruction and thus might be considered as assembly time opcodes instead of execution time opcodes. The directives are: .BYTE, .WORD, .DBYTE, .OPT, .PAGE, .SKIP, .END and equates which are denoted by the equals sign =. All directives which are proceded by the period may be abbreviated to the period and three characters if desired (eg. .BYT).

.BYTE is used to reserve one byte of memory and load it with a value. The directive may contain multiple operands which will store values in consecutive bytes. ASCII strings may also be generated by enclosing the string with quotes.

```
HERE    .BYTE  2
THERE   .BYTE  1, $F, @3, %101, 7
ASCII   .BYTE  'ABCDEFH'
```

Note that numbers may be represented in the most convenient form. In general, any valid MCS650X expression which can be resolved to eight bits may be used in this directive. If it is desired to include a quote in an ASCII string, this may be done by putting two quotes in the string;

```
.BYTE 'JIM"S CYCLE'
```

could be used to print:

```
JIM'S CYCLE
```

.WORD is used to reserve and load two bytes of data at a time. Any valid expression, except for ASCII strings, may be used in the operand field.

```
HERE   .WORD  2
THERE  .WORD  1, $FF03, @3
WHERE  .WORD  HERE, THERE
```

The most common use for .WORD is to generate addresses as shown in the above example labelled "WHERE" which stores the 16 bit addresses of "HERE" and "THERE". Addresses in the MCS650X are fetched from memory in the order low byte, high byte,

and therefore .WORD generates the values in this order. The hexadecimal portion of the second example above ($FF03) would be stored 03,FF. If this order is not desired, the following directive is used.

.DBYTE is exactly like .WORD except the bytes are stored in high byte, low byte order.

```
.DBYTE   $FF03
```

will generate FF,03. Thus, fields generated by .DBYTE may not be used as indirect addresses.

= is the EQUATE directive and is used to reserve memory locations, reset the program counter (*), or assign a value to a symbol.

```
HERE      *=*+1          reserve one byte
WHERE     *=*+2          reserve two bytes
*=$200                   set program counter
NB=8                     assign value
MB=NB+%101               assign value
```

The = directive is very powerful and can be used for a wide variety of purposes.

Expressions must not contain forward references or they will be flagged as an error.

For example,

```
* = C + D - E * F
```

would be legal if C, D, E and F are all defined but would be illegal if any of the variables were a forward reference. Note also that expressions are evaluated in strict left to right order.

.PAGE is used to cause an immediate jump to top of page and may also be used to generate or reset the title printed at top of page.

```
.PAGE   'THIS IS A TITLE'
.PAGE
.PAGE   'NEW TITLE'
```

If a title is defined, it will be printed at the top of each page until it is redefined or cleared. A title may be cleared with: .PAGE ' '.

.SKIP is used to generate blank lines in a listing. The directive will not appear but its position may be found in a listing since it is treated as a valid input "card" and the card number printed on the left side of the listing will jump by two when the next line is printed.

```
.SKIP    2       skip two blank lines
.SKIP    3*2-1   skip five lines
.SKIP    ONELIN
```

.OPT is the most powerful directive and is used to control generation of output fields, listings and expansion of ASCII strings in .BYTE directives

```
.OPT   XREF, ERRORS, COUNT, LIST, MEMORY, GENERATE
.OPT   NOXREF, NOERRORS, NOCOUNT, NOLIST, NOMEMORY, NOGENERATE
```

The operand fields in this directive are only scanned for the first three characters, thus all fields may be shortened to:

```
.OPT   XRE,ERR,COU,LIS,MEM,GEN
.OPT   NOX,NOE,NOC,NOL,NOM,NOG
```

Also valid is:

```
.OPT   CNT
```

Default settings are:

```
.OPT   NOCNT,XREF,MEM,LIST,ERR,NOGEN
```

The individual .OPT operands are:

(1) XREF [NOXREF] controls whether a full cross reference listing will be printed. A symbol table will always be printed (unless NOLIST is used, see below).

(2) ERRORS [NOERRORS] is used to control creation of a separate error file. The error file contains the source line in error and the error message. This facility is normally of greatest use to time-sharing users who have limited print capacity. The error file may be turned on and examined until all errors have been corrected. The listing file may then be examined. Another possibility is to run with:

```
.OPT    ERRORS, NOLISTING
```

until all errors have been corrected and then make one more run with

```
.OPT    NOERRORS, LISTING
```

(3) COUNT [NOCOUNT] is used to generate a count of times each instruction has been used in a program and data on the number of symbols, bytes of code generated, etc. which are mainly of use to batch users who might have to recompile the assembler if they desire to assemble very large programs. An instruction count can be very useful to indicate if certain instructions which might be useful are not being used due to lack of familiarity with the entire instruction set. CNT may also be used for COUNT.

(4) LIST [NOLIST] is used to control the generation of the listing file which contains source input, errors and warnings, code generated, symbol table and instruction count if enabled.

(5) MEMORY [NOMEMORY] is used to control generation of the memory file which is used as an interface between the assembler and the simulator and various loader programs. The memory file contains information about symbols, line numbers and code generated and is described in detail elsewhere in this document.

(6) GENERATE [NOGENERATE] is used to control printing of ASCII strings in the .BYTE directive. The first two characters will always be printed and further characters will be printed (normally two bytes per line) if GENERATE is used.

.END should be the last statement in a program and is used to signal the physical end of the program. Its use is optional but highly recommended for program documentation.

VI. Output files

There are three output files generated by the assembler. Each file is optional through use of the .OPT assembler directive. The listing file contains the program list, symbol table and instruction count. The error file contains all error lines and errors. The interface file contains the interface to the simulator.

A. Listing file

The listing file will be produced unless the NOLIST option is used on the .OPT assembler directive. This file is made up of three sections: program, symbol table and instruction count.

1. Program

This listing will always be produced unless the NOLIST option is selected. It contains the source statements of the program along with the assembled code. Errors and warnings appear after erroneous statements. For an explanation of error codes see part B in this section.

At the end of the program is a count of the errors and warnings found during the assembly. An example of this section is shown below.

```
CARD # LOC     CODE        CARD

   1                      CR=15
   2                      LF=12
   3                      ; LOW CORE DATA AREAS
   4  0000  E7 06         TEMTBL   .WORD  G3TEM, G1TEM
   5  0002  E7 05
   6                      GROUP=B10
   7  0004  00            THI      .BYTE 0
   8  0005  00            TLO      .BYTE 0
   9  0006  EA EA EA      3PER     .WORD 0
*****  ERROR ** LABEL DOESN'T BEGIN WITH ALPHABETIC CHARACTER - NEAR COLUMN 1
  10  0009  B1 0E         NEXT     LDA     (SAVIL)Y
   .
   .
   .
 269  07C9  C9 3B                  CMP     #';
 270  07CB  F0 EA                  BEQ     DONE
*****  ERROR ** UNDEFINED SYMBOL - NEAR COLUMN   18
 280                            .END
```

```
END OF MOS/TECHNOLOGY 650X ASSEMBLY VERSION 4
NUMBER OF ERRORS =   2, NUMBER OF WARNINGS =   0
```

2. Symbol table

The symbol table will always be produced unless the NOLIST option is used. It contains a list of all symbols used in the program, their value and the line they are defined in. The cross-reference listing is part of the symbol table and is produced unless the NOXREF option is used. It contains cross-references for each symbol. Symbols that are undefined are flagged as such with cross-references remaining in the listing. Part of a symbol table listing with cross-references is shown below.

```
SYMBOL     VALUE  LINE DEFINED        CROSS-REFERENCES

AGAIN      093C         369     374
BLANK      07F6         292     247  274  285
DONE      *UNDEFINED*     0     270
EFLAG      001D          25
```

3. Instruction count

The instruction count table is optional and will be produced unless the NOCOUNT option is selected in the .OPT directive. This table is a listing, in alphabetical order, of all the op codes with a usage count for each one. At the end of the table is a count of the number of symbols, bytes, lines and cross-references generated along with the assembler limits for each of these. An example of the instruction count table is shown below.

```
ADC          2
AND          0
ASL          5
BCC          1
BCS          0
      :
TXS          5
TYA          3

# SYMBOLS =  20(LIMIT=500)      # BYTES = 243(LIMIT=1500)
# LINES = 280(LIMIT=1000)      # XREFS = 75(LIMIT=1500)
```

B. Error file

Error messages are given in the program listing accompanying the statement in error as shown in previous examples. The same information may be produced on a separate file unless the NOERRORS option is specified. This file can be used conveniently if the NOLIST option is taken. It would typically be used with a timesharing system where a long program listing during debug is time-consuming and unnecessary. The following is a list of all error messages which might be produced during assembly.

** A, X, Y, S, AND P ARE RESERVED NAMES

A label on a statement is one of the five reserved names (A, X, Y, S and P). They have special meaning to the assembler and therefore cannot be used as labels. Use of one of these names will cause the above error message to be printed and no code to be generated for the statement. The label does not get defined and will appear in the symbol table as an undefined variable. Reference to such a label elsewhere in the program will cause error messages to be printed as if the label were never declared.

How to avoid: don't use A, X, Y, S or P as a label to a statement.

** ACCUMULATOR MODE NOT ALLOWED

Following a legal opcode and one or more spaces is the letter A followed by 1 or more spaces. The assembler is trying to use the accumulator (A means accumulator mode) as the operand. However, the opcode in the statement is one which does not allow reference to the accumulator. Check for a statement labelled A (an illegal statement) to which this statement is referencing. If you were trying to reference the accumulator, look up the valid operands for the opcode used.

** ADDRESS NOT VALID

An address referred to in an instruction or in one of the assembler directives (.BYTE and .WORD) is invalid. In the case of an instruction, the operand that is generated by the assembler must be greater than or equal to zero and less than or equal to $FFFF_{16}$ (2 bytes long). (This excludes relative branches which are limited to $\pm$ 127 from the next instruction.) If the operand generates more than 2 bytes of code or is less than zero, this error message will be printed. For a .BYTE each operand is limited to one byte and for a .WORD each operand is limited to two bytes. All must be greater than or equal to zero.

This validity is checked after the operand is evaluated. Check for values of symbols used in the operand field (see the symbol table for this information).

** FORWARD REFERENCE IN EQUATE OR ORG

The expression on the right side of an equals sign contains a symbol that hasn't been defined previously. One of the operations of the cross assembler is to evaluate expressions or labels and assign addresses or values to them. The cross assembler processes the input values sequentially which means that all of the symbolic values that are encountered fall into two classes--already defined values and not previously encountered values. The cross assembler assigns defined values and builds a table of undefined values. When a previously used value is discovered, it is substituted into the table and the cross assembler processes all of the input statements a second time using currently defined values.

A label or expression which uses a yet undefined value is considered to be referenced forward to the to-be-defined value.

To allow for conformity of evaluating expressions, this cross assembler allows for one level of forward reference so that the following code is allowed:

Card Sequence	Label	Opcode	Operand
100		BNE	New One
200	New One	LDA	#5

but the following is not allowed:

Card Sequence	Label	Opcode	Operand
100		BNE	New One
200	New One		Next + 5
300	Next	LDA	#5

This feature should not disturb the normal use of labels as the cure for this error.

Card Sequence	Label	Opcode	Operand
100		BNE	New One
300	Next	LDA	#5
301	New One		Next + 5

is very simple and always solves the problem.

This error may also mean that the value on the right side of the = is not defined at all in the program in which case the cure is the same as for undefined values.

Due to the sequential processing of the assembler and the dependency on the value of the program counter on symbols, throughout the rest of the program, the assembler cannot

process a forward reference in this type of statement. All expressions with symbols that appear on the right side of any equals sign must refer only to previously defined symbols for the equate to be processed.

** ILLEGAL OPERAND TYPE FOR THIS INSTRUCTION

After finding an opcode that does not have an implied operand, the assembler passes the operand field (the next non-blank field following the opcode) and determines what type of operand it is (indexed, absolute, etc.). If the type of operand found is not valid for the opcode, this error message will be printed.

Check to see what types of operands are allowed for the opcode and make sure the form of the operand type is correct (see the section on addressing modes).

** ILLEGAL OR MISSING OPCODE

The assembler searches a line until it finds the first non-blank character string. If this string is not one of the 55 valid opcodes it assumes it is a label and places it in the symbol table. It then continues parsing for the next non-blank character string. If none is found, the next line will be read in and the assembly will continue. However, if a 2nd field is found it is assumed to be an opcode (since one label is allowed per line). If this character string is not a valid opcode, the error message is printed.

This error can occur if opcodes are misspelled in which case the assembler will interpret the opcode as a label (if no label appears on the card). It will then try to assemble the next field as the opcode. If there is another field, this error will be printed.

Check for a misspelled opcode or for more than one label on a line.

** INVALID EXPRESSION

In evaluating an expression, the assembler found a character it couldn't interpret as being part of a valid expression. This can happen if the field following an opcode contains special characters not valid within expressions (e.g. parentheses). Check the operand field and make sure only valid special characters are within a field (between commas).

** INVALID INDEX - MUST BE X OR Y

After finding a valid opcode, the assembler looks for the operand. In this case, the first character in the operand field is a left paren. The assembler interprets the next field as an indirect address which, with the

exception of the jump statement, must be indexed by one of the index registers, X or Y. In the erroneous case, the character the assembler was trying to interpret as an index register was not X or Y and the error was printed.

Check for the operand field starting with a left paren. If it is supposed to be an indirect operand, recheck the correct format for the two types available. If the format was wrong (missing right paren or index register), this error will be printed. Also check for missing or wrong index registers in an indexed operand (form: expression, index register)

** LABEL DOESN'T BEGIN WITH ALPHABETIC CHARACTER

The first non-blank field is not a valid opcode. Therefore, the assembler tried to interpret it as a label. However, the first character of the field does not begin with an alphabetic character and the error message is printed.

Check for an unlabelled statement with only an operand field that does start with a special character. Also check for illegal label instruction.

** LABEL GREATER THAN SIX CHARACTERS

All symbols are limited to six characters in length. When parsing, the assembler looks for one of the separating characters to find the end of a label or string. If other than one of these separators is used, the error message will be printed providing the illegal separator causes the symbol to extend beyond six characters in length. Check for no spacing between labels and opcodes. Also check for a comment card with a long first word that doesn't begin with a semicolon. In this case the assembler is trying to interpret part of the comment as a label.

** LABEL OR OPCODE CONTAINS NON-ALPHANUMERIC CHARACTER

Labels are made up of from one to six alphanumeric digits. The label field must be separated from the opcode field by one or more blanks. If a special character or other separator is between the label and the opcode, this error message might be printed.

The 55 valid opcodes are each three alphabetic characters. They must be separated from the operand field (if one is necessary) by one or more blanks. If the opcode ends with a special character (such as a comma), this error message will be printed.

In the case of a lone label or an opcode that needs no operand, they can be followed directly by a semicolon to denote the rest of the card as a comment.

** LABEL PREVIOUSLY DEFINED

The first field on the card is not an opcode so it is interpreted as a label. If the current line is the first line in which that symbol appears as a label (or on the left side of an equals sign) it is put in the symbol table and tagged as defined in that line. However, if the symbol has appeared as a label, or on the left of an equate, prior to the current line, the assembler finds the label already in the symbol table. The assembler does not allow redefinitions of symbols and will, in this case, print the error message.

** OUT OF BOUNDS ON INDIRECT ADDRESSING

An indirect address is recognized by the assembler by the parentheses that surround it. If the field following an opcode has parens around it, the assembler will try to assemble it as an indirect address. Since indirects work only in page zero memory, if the address in the operand field extends into absolute (is larger than 256 - one byte) this error message will be printed.

This error will only occur if the operand field is in correct form (i.e. an index register following the address), and the address field is out of page zero. To correct this, the address field must refer to page zero memory.

** PROGRAM COUNTER NEGATIVE - RESET TO 0

An assembled program is loaded into core from position 0 to 64K (65536). This is the extent of the machine. Instructions can only refer to up to 2 bytes of information. Because there is not such a thing as negative memory, an attempt to reference a negative position will cause this error and the program counter (or pointer to the current memory location) will be reset to 0.

When this error occurs, the assembler continues assembling the code with the new value of the program counter. This could cause multiple bytes to be assembled into the same locations. Therefore, care should be taken to keep the program counter within the proper limits.

** RAN OFF END OF CARD

This error message will occur if the assembler is looking for a needed field and runs off the end of the card (or line image) before the field is found. The following should be checked for: a valid opcode field without an operand field on the same card; an opcode that was thought to take an implied operand, which in fact needed an operand; an ASCII string that is missing the closing quote (make sure any embedded quotes are doubled - to have a quote in the string at the end, there must be 3 quotes - 2 for the embedded quote and one to close off the string); a comma at the end of the operand field indicates there are more operands to come; if there aren't other operands, the assembler will run off the card looking for them.

** RELATIVE BRANCH OUT OF RANGE

All of the branch instructions (excluding the two jumps) are assembled into 2 bytes of code. One byte is for the opcode and the other for the address to branch to. To allow a forward or backward branch, this branch is taken relative to the beginning of the next instruction, according to the address byte. If the value of the byte is 0-127 the branch is forward; if the value is 128-255 the branch is backward. (A negative branch is in 2's complement form). Therefore, a branch instruction can only branch forward or backward 127 bytes relative to the beginning of the next instruction. If an attempt is made to branch further than these limits, the error message will be printed.

** UNDEFINED ASSEMBLER DIRECTIVE

All assembler directives begin with a period. If a period is the first character in a non-blank field the assembler interprets the following character string as a directive. If the character string that follows is not a valid assembler directive, this error message will be printed.

Check for a misspelled directive, or a period at the beginning of a field that is not a directive.

** UNDEFINED SYMBOL

This error is generated by the 2nd pass. If in the first pass the assembler finds a symbol in the operand field (the field following the opcode or an equals sign) that has not been defined yet, that symbol is flagged for interpretation by pass 2. If the symbol is defined (shows up on the left of an equate or as the first non-blank field in a statement) pass 1 will define it and enter it in the symbol table. Then a symbol in an operand field before the definition will be defined with a value when pass 2 assembles it. In this case the assembly process can be completed. However,

if pass 1 doesn't find the symbol as a label or on the left of an equate, it never enters it in the symbol table as a defined symbol. When pass 2 tries to interpret the operand field this type of symbol is in, there is no value for the symbol and the field cannot be interpreted. Therefore, the error message is printed with no value for the operand.

This error will also occur if a saved symbol (A, X, Y, S or P) is used as a label and refered to elsewhere in the program. On the statement that references the saved symbol, the assembler sees it as a symbol that has not been defined.

Check for use of saved symbols, misspelled labels or missing labels to correct this error.

Along with the error messages listed, there is one warning message that might be printed. The difference between the errors and warnings is that unlike errors, warnings generate the full code for the statement. Errors generate partial code and leave NOPs where code cannot be generated. The following is the warning that might be produced during assembly.

** FORWARD REFERENCE TO PAGE ZERO MEMORY

When the assembler finds an expression (whether it is in an operand field or on the right of an equals sign) it tries to evaluate the expression. If there is a symbol within the expression that hasn't been defined yet, the assembler will flag it as a forward reference and wait to evaluate it in the second pass. If the expression is on the right side of an equals sign, the forward reference is a severe error and will be flagged as such. However, if the expression is in an operand field of a valid opcode, the first pass will set aside 2 bytes for the value of the expression and flag it as a forward reference. When the 2nd pass fills in the value of the expression, this warning will be printed if the expression is one byte long

(i.e. $\leq$ 256). The warning is printed because the forward reference to page zero memory wastes one byte of memory - the extra one that was saved because during the first pass the assembler didn't know how large the value was so had to save for the largest value - 2 bytes.

C. Interface file

The interface file will be produced on a separate file unless the NOMEMORY option is used in the .OPT directive. This file is the output from the assembler that is used as input to the simulator and other loader programs.

D. The following example lists some of the characteristics and capabilities of the MCS650X cross-assembler.

(1) The title is generated with the following card:

```
.PAGE  'MULTIPLE BYTE DECIMAL ADD'
```

The page directive can be used for listing control and title information. A directive with no title field will cause a skip to top of next page. If a title had previously been used, it will be printed again. To clear a title field, enter the following:

```
.PAGE  ' '
```

(2) Comment - first non-blank is ';'.

(3) The program counter is set to zero. In this example, not really necessary as the program counter is automatically started at zero.

(4) An equate. The variable NB is assigned the decimal value 8.

(5) These instructions provide the dual purpose of defining the start of data areas and reserving memory locations for the data. Expressions which do not contain forward references are permitted.

(6) This shows an address calculated from the current value of the program counter. The current PC (*) points to the beginning of the instruction (24), hence *-1=23. Note also that addresses on the MCS650X machines are stored low byte, high byte; thus the operand field of the jump instruction is 23,00 and not 00,23. The code printed on the assembly listing is exactly as it is loaded in memory and fetched by the processor during machine execution.

(7) Blank line generated by .SKIP 1. Note card was counted but not printed.

(8) Program counter is set to decimal 100 (Hex 64).

(9) Immediate mode used to load Y with byte count.

(10) Expressions. Note that expressions are evaluated in strict left to right order with no parenthetical nesting allowed. Thus the logical evaluation of the expression is:

(3*NB)-1 = (3*8)-1 = 24-1 = 23 = 17 (Hex)

(11) The .END directive signals end of assembly.

(12) Cross-reference listing requested by

.OPT XREF

Showing sorted symbol table (12A), value (12B), line defined (12C), and line number where symbol was referenced (12D). A symbol which is referenced but never defined will be clearly marked with:

**UNDEFINED

in the value field and symbols defined but never referenced are marked by

****

in the reference position. This is not an error but is included for programmer reference as an unreferenced symbol may sometimes indicate logical errors.

(13) Count of all instructions used requested by

.OPT COUNT
or .OPT CNT

(14) Error file requested by:

.OPT ERRORS

Contains card where error occurs and error message. This is normally of greatest use to time-sharing users who have limited access to high speed printers. The error file can be listed first on the terminal to see if there were any assembly errors. If there were, they can be corrected in the source input without listing the entire printout at the terminal.

(15) This is the interface file which is created by the assembler as input

to the simulator and various loader programs. All interface file records begin with a semicolon and a number. Records then contain a four character hexadecimal sequence number. The format after this point depends on the record type.

(15A) Type 1 records are basically a dump of the symbol table giving symbols and their value.

(15B) Type 2 records describe the value of the program counter at the beginning of each card.

(15C) Type 3 records contain the code generated by the program and are normally of greatest interest. Each record contains the standard information, a hexadecimal byte count, starting program counter value, 32 characters containing 16 bytes of hexadecimal code and a four character checksum. On records which contain fewer than 16 bytes of data, the unused byte positions are filled with zeroes. In this example, the first type 3 record contains 14 (decimal) bytes of data, the second a full 16 and the last contains 5 bytes.

(15D) Type 4 records are used to denote the end of the interface file.

SYMBOL TABLE

SYMBOL	VALUE	LINE DEFINED	CROSS-REFERENCES
ADDR	0000	8	21 30
BCD	0064	29	22
MAIN	0019	18	****
NB	0008	10	11 12 13 29 33 34 35
NEXT	006A	33	38
PP	0001	11	20
Q	0009	12	****
RES	0011	13	****

12A 12B 12C 12D

12

```
INSTRUCTION COUNT

ADC     1
AND     0
ASL     0
BCC     0
BCS     0
BEQ     0
BIT     0
BMI     0
BNE     1
BPL     0
BRK     0
BVC     0
BVS     0
CLC     1
CLD     1
CLI     0
CLV     0
CMP     0
CPX     0
CPY     0
DEC     0
DEX     1
DEY     1
EOR     0
INC     0
INX     0
INY     0
JMP     1
JSR     1
LDA     1
LDX     3
LDY     1
LSR     0
NOP     1
ORA     0
PHA     0
PHP     0
PLA     0
PLP     0
ROL     0
RTI     0
RTS     1
SBC     0
SEC     0
SED     1
SEI     0
STA     1
STX     1
STY     0
TAX     0
TAY     0
TSX     0
TXA     0
TXS     1
TYA     0

# SYMBOLS =     8 (LIMIT =   400)     # BYTES =    36 (LIMIT =  4096)
# LINES =    42 (LIMIT =  2100)     3# XREFS =  4  13 (LIMIT =   900) 6
```

(13)

FORM # 1412PC

Using the G.E. Timesharing Cross-assembler.

Before using the assembler, users should be familiar with the BLIST, MEDIA and editing commands. The BLIST command is used to list the output reports returned by the assembler and the MEDIA command is used to transfer them to foreground files.

Prior to running the assembler, a file must be created containing the source code to be assembled. This file must not contain line numbers (EDIT DESEQUENCE command) and all alphabetic characters must be in upper case.

Once the file is created and ready to be assembled, call the assembler by issuing the command:

RUN MOSASM

The assembler will respond by typing a header and requesting the input filename. Enter the filename containing the code to be assembled and hit the carriage return. Failure to enter a saved filename will cause the error

INPUT FILE NOT SAVED - PROGRAM TERMINATED

to be printed, followed by termination of the program.

If a saved filename is entered, the assembler will then ask for the control filename. This is the file that will be created by the interface program and background to control the assembly process. If the filename entered is an old file, the old file will be overwritten; if it is a new file, the file will be created and placed in the user's area. It is up to the user to delete this file. If blanks or a carriage return are entered rather than a filename, the question will be repeated.

Once the control file has been established, the assembler will ask for a priority. Valid entries are O(overnight), N or carriage return (normal), P (priority) and S (super). The program checks the first character only and an invalid entry will cause a repeat of the question. Refer to the GE manual 2000.01B for a description of the priorities.

When the priority is properly entered the job will be transfered to background for processing. The user's job ID will be printed followed by a summary of the reports generated by the assembler. Refer to the section on output files for further explanations of these reports. The reports can be listed using the BLIST command. If you wish to save the reports beyond 24 hours, use the MEDIA command to transfer them to foreground files.

VI. PROCEDURE FOR USING THE MCS650X CROSS-ASSEMBLER ON THE NCSS SYSTEM

1. Dial the appropriate number for the terminal speed you are using and sign on. A sample sign on for a 30 cps terminal is shown below.

```
CSS ONLINE - STMI

>L HSYS MOSTEC01
PASSWORD:
XXXXXXXXX
A/C INFO:
>DB
HSYS READY AT 15.43.20 ON 02SEPT75
CSS.211 05/07/75
```

2. Using the editor build and save a file containing the desired source code. In Example 1, the sample program was inputed, saved, and then listed using the PRINTF command.

```
14.46.30 >EDIT DICKI DATA
NEW FILE.
INPUT:
>;
>;  650X CROSS ASSEMBLER SAMPLE PROGRAM.
>;
> * = $C000     DEFINE ORIGIN.
> LDX #$FF      SET UP STACK.
> TXS           LOAD STACK POINTER.
> LDA #$F0      LOAD A WITH HEX F0.
> STA ASAVE     SAVE A IN ASAVE.
>;
>;  ALLOCAR?TE SAVE AREA[
>;  ALLOCATE SAVE AREA.
>;
> * = $0000
>ASAVE * = *
> .END
>
EDIT:
>FILE

14.52.22 >PRINTF DICKI DATA

;
;  650X CROSS ASSEMBLER SAMPLE PROGRAM.
;
 * = $C000     DEFINE ORIGIN.
 LDX #$FF      SET UP STACK.
 TXS           LOAD STACK POINTER.
 LDA #$F0      LOAD A WITH HEX F0.
 STA ASAVE     SAVE A IN ASAVE.
;
;  ALLOCATE SAVE AREA.
;
 * = $0000
ASAVE * = *
 .END
```

Example 1

3. Build a file containing the JCL required to run the job and save this file. In Example 2, the JCL file is saved as Sampl Exec.

   File 1 is a scratch file which will contain the intermediate object code between Pass 1 & Pass 2.

   File 2 is a scratch file which will contain all the error messages generated by the assembly just completed.

   File 3 is a permanent file which will contain the symbol table, the line number table, and the object code in loader format.

   File 5 is the source code as entered above.

   File 6 is defined as the terminal and the assembly will be listed as it runs. This could be defined as a disk file and listed after the assembly is completed.

```
14.52.34 >EDIT SAMP1 EXEC
NEW FILE.
INPUT:
>ATTACH TEMP5 AS 192
>FILEDEF 1 DSK SCRAT1 DATA T LRECL 120
>FILEDEF 2 DSK SCRAT2 DATA T LRECL 120
>FILEDEF 3 DSK FCCAI DATA
>FILEDEF 5 DSK DICKI DATA P
>FILEDEF 6 CONO
>RUN VV3S
>
EDIT:
>FILE

15.15.04 >PRINTF SAMP1 EXEC

ATTACH TEMP5 AS 192
FILEDEF 1 DSK SCRAT1 DATA T LRECL 120
FILEDEF 2 DSK SCRAT2 DATA T LRECL 120
FILEDEF 3 DSK FCCAI DATA
FILEDEF 5 DSK DICKI DATA P
FILEDEF 6 CONO
RUN VV3S
```

Example 2

4. List the error file and the source code file as desired using the PRINTF command. Examples 3 and 4 show the assembler listing and the listing of Files 2 and 3.

```
15.18.31 >SAMP1
15.18.37 ATTACH TEMP5 AS 192
SCRATCH  ATTACHED AS 192,(T)
15.18.38 FILEDEF 1 DSK SCRAT1 DATA T LRECL 120
15.18.38 FILEDEF 2 DSK SCRAT2 DATA T LRECL 120
15.18.38 FILEDEF 3 DSK FCCAI DATA
15.18.38 FILEDEF 5 DSK DICKI DATA P
15.18.38 FILEDEF 6 CONO
15.18.38 RUN VV3S
EXECUTION:

CARD #  LOC      CODE        CARD                              PAGE 1
    1                      ;
    2                      ;  650X CROSS ASSEMBLER SAMPLE PROGRAM.
    3                      ;
    4  0000                 * = $C000     DEFINE ORIGIN.
    5  C000   A2 FF         LDX  $FF      SET UP STACK.
    6  C002   9A            TXS           LOAD STACK POINTER.
    7  C003   A9 F0         LDA  $F0      LOAD A WITH HEX F0.
    8  C005   8D 00 00      STA ASAVE     SAVE A IN ASAVE.
***** WARNING ** FORWARD REFERENCE TO DIRECT MEMORY - NEAR
                                                      COLUMN 6
    9                      ;
   10                      ;  ALLOCATE SAVE AREA.
   11                      ;
   12  C008                 * = $0000
   13  0000                ASAVE * = *
   14                       .END

END OF MOS/TECHNOLOGY 6501 ASSEMBLY VERSION 3
NUMBER OF ERRORS =   0,  NUMBER OF WARNINGS =   1


          SYMBOL TABLE

SYMBOL      VALUE  LINE DEFINED        CROSS-REFERENCES

ASAVE       0000         13     8
```

Example 3

```
15.20.32 >PRINTF SCRAT2 DATA

ERROR FILE

CARD ✣  CARD
    8    STA ASAVE     SAVE A IN ASAVE.
***** WARNING ** FORWARD REFERENCE TO DIRECT MEMORY - NEAR COLUMN    6


15.21.03 >PRINTF FCCAI DATA

;12711 ASAVE  0000         0000        0000          0000
;24E21      1 0000      2 0000     3 0000       4 0000      5 C000
;24E22      6 C002      7 C003     8 C005       9 C008     10 C008
;24E23     11 C008     12 C008    13 0000      14 0000      0 0000
;37531 08 C000 A2FF9AA9F08D00000000000000000000 0461


15.21.40 >DETACH TEMP5
 !!E(00002)!!

15.22.07 >DR
INVALID CSS COMMAND

15.22.14 >DETACH TEMP5 AS 192
DEV 192 DETACHED

15.22.26 >LOG
23.730 VPU'S,    1.01 CONNECT HRS,  988 I/O
LOGGED OFF AT 15.22.32 ON 02SEPT75
xZω(
```

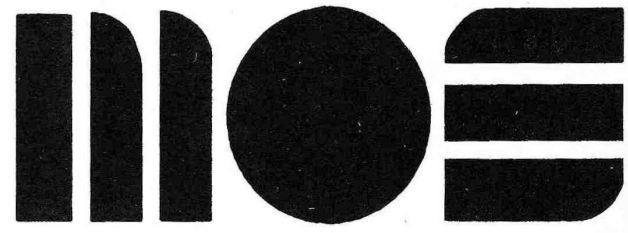
Example 4

INSTRUCTION COUNT

ADC	0	DEC	0	ROL	0
AND	0	DEX	0	RTI	0
ASL	0	DEY	0	RTS	0
BCC	0	EOR	0	SBC	0
BCS	0	INC	0	SEC	0
BED	0	INX	0	SED	0
BIT	0	INY	0	SEI	0
BMI	0	JMP	0	STA	1
BNE	0	JSR	0	STX	0
BPL	0	LDA	1	STY	0
BRK	0	LDX	1	TAX	0
BVC	0	LDY	0	TAY	0
BVS	0	LSR	0	TSX	0
CLC	0	NOP	0	TXA	0
CLD	0	ORA	0	TXS	0
CLI	0	PHA	0	TYA	0
CLV	0	PHP	0		
CMP	0	PLA	0		
CPX	0	PLP	0		
CPY	0				

HEADQUARTERS –
MOS TECHNOLOGY, INC. 950 Rittenhouse Road
Norristown, Pa. 19401, (215) 666-7950, TWX: 510/660/4033
EASTERN REGION –
Mr. William Whitehead
MOS TECHNOLOGY, INC., Suite 312,
410 Jericho Turnpike, Jericho, N.Y. 11753
(516) 822-4240
WESTERN REGION –
MOS TECHNOLOGY, INC. 2172 Dupont Drive,
Patio Bldg., Suite 221. Newport Beach, CA. 92660
(714) 833-1600

Mr. Petr Sehnal, Regional Applications Mgr.
MOS TECHNOLOGY, INC., 26921 Grasmere Place,
Hayward, CA. 94542
(415) 881-8080



MOS TECHNOLOGY, INC.