MOS

MOS

MOS

INSTRUTEK

8700 Horsens · Tlf. 05 · 61 11 00

MICROCOMPUTERS

MICROCOMPUTERS

MICROCOMPUTERS

MICROCOMPUTERS

KIM ASSEMBLER

MANUAL PRELIMINARY

MICROCOMPUTER FAMILY
KIM ASSEMBLER MANUAL
Preliminary

JANUARY 1977

MOS TECHNOLOGY · Frankfurter Str. 171–175 · D–6378 Neu-Isenburg
Telefon (06102) 8003 · Telex 04 185663

TABLE OF CONTENTS

LIST OF TABLES

I. INTRODUCTION

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.

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 KIM assembler 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 KIM Assembler 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 KIM Assembler 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

KIM 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 Results 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)
RØR	Rotate One Bit Right (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 space must separate the fields of an instruction.

There are five assembler directives used to reserve storage and direct information to the assembler. Four have symbolic names with a period as the first character. The fifth, 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     .OPT     .END     =
```

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

A typical KIM 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.
 |    |      |       \         |    |                      |
 |    |      |      label      |  operand              comments
 |    |      |                 |
 |    |  object code         opcode
 |    |    address
 |  memory
 |  number
 |
line number
```

Example 1. Segment of a KIM 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. Both ALPHA and BETA must have previously been defined.

Memory associated with the KIM processor 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.

Declaring an Address in Immediate Mode

It is often useful to be able to reference the address of a label as immediate mode data. The assembler recognizes the characters > and < for this purpose. For instance:

```
LDA #<HERE
```

will load the accumulator with the low order eight bits of the *address* of the byte labeled HERE and:

```
LDA #>HERE
```

will load the accumulator with the high order byte of the address of HERE. This replaces the usual method of declaring the address as a label, saving four bytes:

Bytes		Bytes	
2	THERE .WORD HERE		
3	LDA THERE	2	LDA #>HERE
3	LDA THERE+1	2	LDA #>HERE
8		4	

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 instruction.

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.

Four instructions ASL, LSR, ROR, and ROL are special in that the accumulator, A, can be used as an operand. In this special case these four 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

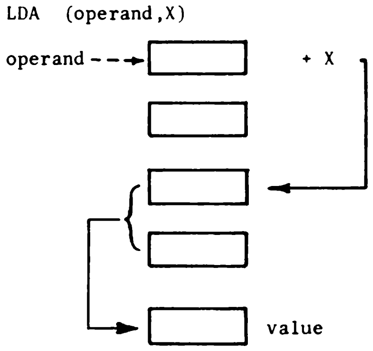
In this mode the operand address is a location in page zero memory which contains the address to be used as an operand. An example is:

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:

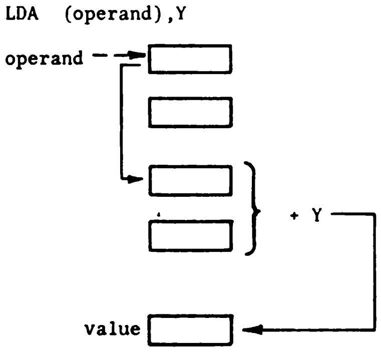
```
LDA   (GAMMA),Y
```

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		
ROR (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. <u>Instruction addressing modes</u>

III. ASSEMBLER DIRECTIVES

There are five 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, .OPT, .END and equates which are denoted by the equals sign =. All directives which are preceded 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.

= 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       reserves two bytes
*=$200              set program counter
NB=8                assign value
MB=NB+%101          assign value
```

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

For example:

```
Q = 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.

.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   ERRORS, LIST, SYMBOLS, GENERATE, TAB
.OPT   NOERRORS, NOSYMBOLS, NOLIST, NOGENERATE, NOTAB
```

The operand fields in this directive are only scanned for the first three characters.

Default settings are:

```
.OPT   SYM, LIST, ERR, TAB
```

The individual .OPT operands are:

(1) SYM is used to control the printing of the symbol table at the end of the listing. The symbol table is not sorted.

(2) ERRORS {NOERRORS} is used to control creation of error listing. To view only errors on the assembly use:

```
                    .OPT   ERRORS, NOLISTING
```

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

```
                    .OPT   NOERRORS, LISTING
```

(3) LIST {NOLIST} is used to control the generation of the listing which contains source input, errors and warnings & code generated.

(4) 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.

(5) TAB {NOTAB} is used to control automatic spacing of labels, op codes, and comments. Use of this option will save memory space in storing the source code, since only a single space is needed between labels, opcodes, and comments, yet the assembler output is neatly formatted. Users with narrow-carriage terminals (e.g., 32-character/line CRT terminals) may find the spacing objectionable and wish to turn it off.

.END must be the last statement in a program and is used to signal the physical end of the program.

ERROR MESSAGES

Error #1

NOT USED

Error #2

FATAL - LABEL PREVIOUSLY DEFINED

The first field on the line 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.

Error #3

FATAL - 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.

Error #3 (Continued)

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 line). 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.

Error #4

FATAL - 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 ± 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).

Error #5

FATAL - 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.

Error #5 (Continued)

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.

Error #6

FATAL - FORWARD REFERENCE IN .BYTE OR .WORD

Error #7

FATAL - RAN OFF END OF LINE

- This error message will occur if the assembler is looking for a needed field and runs off the end of the line before the field is found. The following should be checked for: a valid opcode field without an operand field on the same line; an opcode that was thought to take an implied operand, which in fact needed an operand; as 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 line looking for them.

Error #8

FATAL - LABEL DOESN'T BEGIN WITH ALPHABETIC CHAR.

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.

Error #9

FATAL - 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 line 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.

Error #10

FATAL - LABEL OR OPCODE CONTAINS NON-ALPHANUMERIC

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.

Error #10 (Continued)

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 line as a comment.

Error #11

FATAL - 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 assembler is to evaluate expressions or labels and assign addresses or values to them. The 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 assembler assigns defined values and builds a table of undefined values. When a previously used value is discovered, it is substituted into the table.

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 assembler allows for one level of forward reference so that the following code is allowed:

Line Number	Label	Opcode	Operand
100		BNE	New One
200	New One	LDA	#5

but the following is not allowed:

Line Number	Label	Opcode	Operand
100		BNE	New One
200	New One		Next + 5
300	Next	LDA	#5

Error #11 (Continued)

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

Line Number	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.

Error #12

FATAL - 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

Error #12 (Continued)

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).

Error #13

FATAL - INVALID EXPRESSION IN OPERAND

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).

Error #14

FATAL - 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.

Error #15

FATAL - INVALID OPERAND FOR PAGE ZERO MODE

Error #16

FATAL - INVALID OPERAND FOR ABSOLUTE MODE

Error #17

FATAL - 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.

Error #18

FATAL - 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).

Error #19

FATAL - 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.

Error #20

FATAL - A,X,Y,S, AND P ARE RESERVED LABELS

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

Error #20 (Continued)

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.

Error #21

FATAL - 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.

Error #22

FATAL - INVALID CHARACTER - EXCEPTING "=" FOR ORG

ASSEMBLER/EDITOR OPERATION

Assembler Location

The KIM Resident Assembler ROM packages must first be inserted in the KIM ROM board and the ROM Board address switches must be set for E000, the starting address of the assembler. See the ROM board documentation for details. The Assembler/Editor resides in memory in locations E000 through F7FF.

Reserving Space for the Symbol Table

The assembler will generate object code and store it at the locations specified in the assembly language program. RAM must exist in your KIM at these locations. If your source program is also memory-resident, your object program should not be written to the same RAM area where the source code resides.

In addition to space for source code and object code, you must reserve space in memory for the assembler to store its symbol table while it is constructing your object program. Each symbol you define in your source program will take 8 bytes of memory in the symbol table, thus if you expect that a particular source program will use about 50 symbols, you should reserve at least 400 bytes for the symbol table. If the assembler runs out of room in the symbol table, the program will terminate with an error message.

You define the RAM area you wish to reserve for the symbol table by entering the upper and lower addresses of the area in two pairs of locations in page zero. At location DF and E0 you enter the low order and high order portions of the beginning of your symbol table and at location E1 and E2 the address of the upper limit of the symbol table.

An example:

You have a 100-line source program with about 25 symbols which you wish to assemble. You have 4K of RAM beginning at location 2000 (hex). You can estimate that your source program will take about 2000 (decimal) bytes (20 characters per average source line) and your symbol table will take about 200 bytes (8 bytes / symbol). You have defined the origin of your object program at location 2000 (hex), and estimate that your source program will generate about 200 bytes of object code (roughly 2 bytes per source line). Thus your total memory allocation should be about:

Source Code	2000	bytes
Object Code	200	bytes
Symbol Table	200	bytes
	2400	bytes

Since you have 4096 bytes of RAM there is no problem.

A reasonable way to allocate memory might be:

Memory Use	Memory Address (hex)	
Source Program	2500-2FFF	(3072 location)
Object Program	2000-21FF	(512 location)
Symbol Table	2200-24FF	(512 location)

So your memory could hold about 150 lines of source code, 500 bytes of object code, and 60 symbols. To define this memory you must specify the origin of your object program at 2000 (hex) by using the statement: *=$2000 in your source program. To specify the origin of your source program, answer "2500" to the BASE = query when entering the source program via the editor.

To specify the location of your symbol table enter the following values:

Memory Location	Value	
0062	00	beginning of symbol table
0063	22	
0064	FF	end of symbol table
0065	24	

These locations should be specified before you enter the editor to type in your source program so that you can go directly from the editor to the assembler (using the A command) with symbol table space already defined.

The terminal session for such a run might look like (what you type is underlined):

```
        (hit RS on KIM and rubout on TTY)

            KIM
            XXXX XX DF

            00DF XX 00.
            00E0 XX 22.
            00E1 XX FF.                Define symbol table
            00E2 XX 24.
            00E3 XX F100               Starting address of Editor
            F100 XX G
            BASE = 2500                Starting address for source
            N OR O? N                  New File

            10 *=$2000                 First line of source code
                |
            570  .END                  Last line of source code
            A                          Go to Assembler
            KIM ASM                    Assembler listing begins
                |
                |
            KIM                        Back to KIM
            XXXX XX 2000               Go to object program
            2000 XX G                  Run your program
              |
```

Assembling From Paper Tape

If your source program is too large to fit in your available KIM system memory, you may assemble directly from paper tape. You create the paper tape by entering successive portions of your source program into the editor and dump each portion to paper tape using the T command. When all the paper tape is ready, set the boundaries for the symbol table and then load a value of 51 in location 00E3 and a value of F4 in 00E4. Go to the starting address of the assembler (E000) and put the first tape in the reader. Hit the G key on your TTY and the assembler will read the first line from tape and assemble it and then read the next line, etc. When the first paper tape ends, insert the next tape and restart the reader. The assembler will continue to read tapes until it encounters a .END directive.

Once the assembly is complete, you may run or store your object program as in a memory-to-memory assembly.

Reserved Memory Locations

The assembler and text editor use all page zero memory locations above location 0042. Object Code should not be assembled into those locations. You may use those locations for data storage for your program, but remember that locations above 0042 will be altered by the assembler and editor during their operation.

Assembling Large Source Programs from Cassette Tape

If you wish to assemble a source program which will not fit in your available memory, you may assemble it in pieces. Enter the first portion of your source program into the text editor and store it on audio cassette (see the editor manual for details). Then enter the next and succeeding portions in the same memory area, designating each as a new file to the editor. Save each portion on audio tape. <u>Only the last segment should contain a .END statement.</u>

Now play the first segment back into memory using normal audio tape loading procedures, enter the editor, giving the base address of the text file and declaring it to be an old file. Assemble the first segment using the A command. The assembler will return to the KIM monitor when it comes to the end of the file.

Now enter the second segment into the same memory area using the audio tape dump procedure. Enter the editor, give the base address of the text file and declare it to be an old file. Do <u>not</u> use the A command. Exit to KIM and enter the assembler through location E011. This will cause the assembler to continue assembly where it left off with the previous file without clearing the previously generated symbol table.

Continue to enter successive segments of text, enter and exit the editor for each segment and return to the assembler at location E011 for each segment. When the assembler detects a .END directive it will print the symbol table and terminate the assembly process.

Notizen

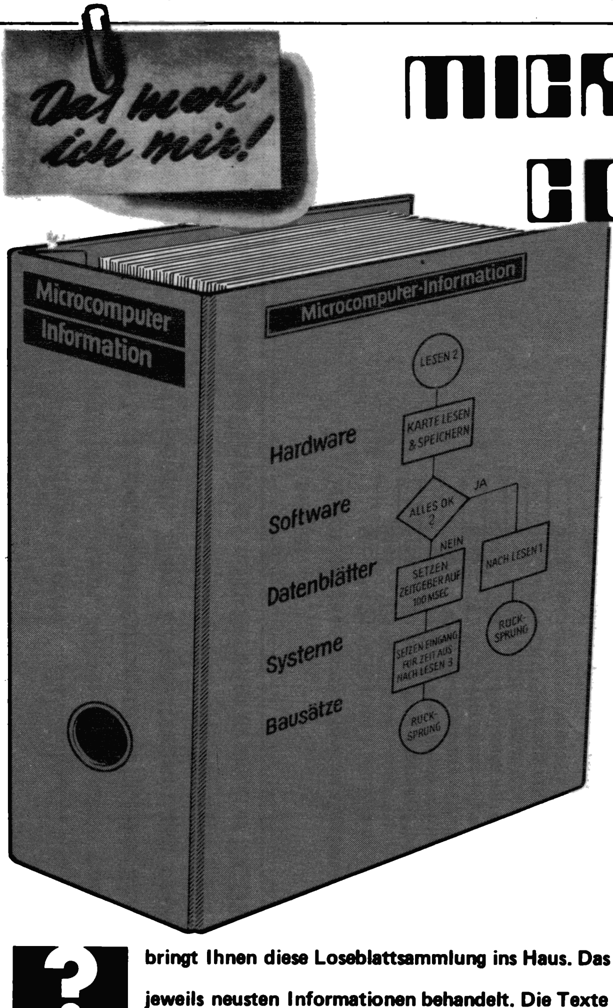
Notizen

Notizen

Notizen

MCDS MICROCOMPUTER-INFORMATION

···nen · Informationen · Info



MICRO-COMPUTER: ALLES ÜBER IHN

Herausgeber:

MCDS Microcomputer Datensysteme GmbH

Was Sie wissen sollten ...

bringt Ihnen diese Loseblattsammlung ins Haus. Das Sachgebiet wird mit Originaldaten und den jeweils neusten Informationen behandelt. Die Texte sind in deutsch und englisch abgefaßt. Der Ausgabenumfang bewegt sich zwischen 1 und 250 Seiten. Um immer up to date zu sein, können Sie als Abonnent kontinuierlich unsere Ergänzungen beziehen.

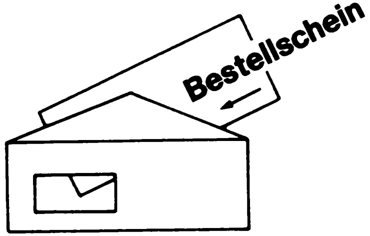
Der Seitenpreis beträgt 0,10 DM.

MCDS Microcomputer Datensysteme GmbH · Luisenplatz 4 · D-6100 Darmstadt

MCDS Microcomputer-Information

nen · Informationen · Info

Eine Information, auf die Sie nicht verzichten können. Allein das Suchen nach einem Begriff, einer Aussag oder einem Programm wird Sie das Vielfache des Abonnentenpreises kosten. Sie sind immer über den neueste Stand der Microcomputer-Technik unterrichtet. Diese Loseblattsammlung deckt das gesamte Spektrum de Sachgebietes „MOS-Microcomputer" ab. Vom Datenblatt, über die Hardware zur wichtigen Software, über Bü cher, Bausätze, Fachberichte, vollständige Systeme finden Sie alles in unserer MCDS Microcomputer-Information



Anforderung

Ich/Wir möchten per Nachnahme zugeschickt bekommen:

____ Stck.	121 MOS-KIM 1 Microcomputer-Handbuch	deutsch	DM 19,80
____ Stck.	101 MOS-KIM 1 Microcomputer-Handbuch	englisch	DM 19,80
____ Stck.	131 MOS-Programmier-Fibel	deutsch	DM 28,60
____ Stck.	111 MOS-Programmier-Fibel	englisch	DM 28,60
____ Stck.	132 MOS-Hardware-Handbuch	deutsch	DM 24,90
____ Stck.	112 MOS-Hardware-Handbuch	englisch	DM 24,90
____ Stck.	201 MOS-Produktspektrum Übersicht		kostenlos
____ Stck.	901 MCDS-Produktspektrum Übersicht		kostenlos
____ Stck.	911 MCDS-Datensichtgeräte Information		kostenlos
____ Stck.	Loseblattsammelordner (Kunststoff)		DM 4,95

☐ Möchte Microcomputer-Information im Abonnent beziehen.

Bitte Absender hier sorgfältig eintragen:

Name, Firma

Name, Firma, Abteilung

Straße, Hausnummer, Postfach

Postleitzahl Ort

MCDS Microcomputer Datensysteme GmbH
Luisenplatz 4

D-6100 Darmstadt

MOS TECHNOLOGY, INC.

microcomputer
datensysteme gmbh·

luisenplatz 4·
d 61 darmstadt·
☎ 06151/23959