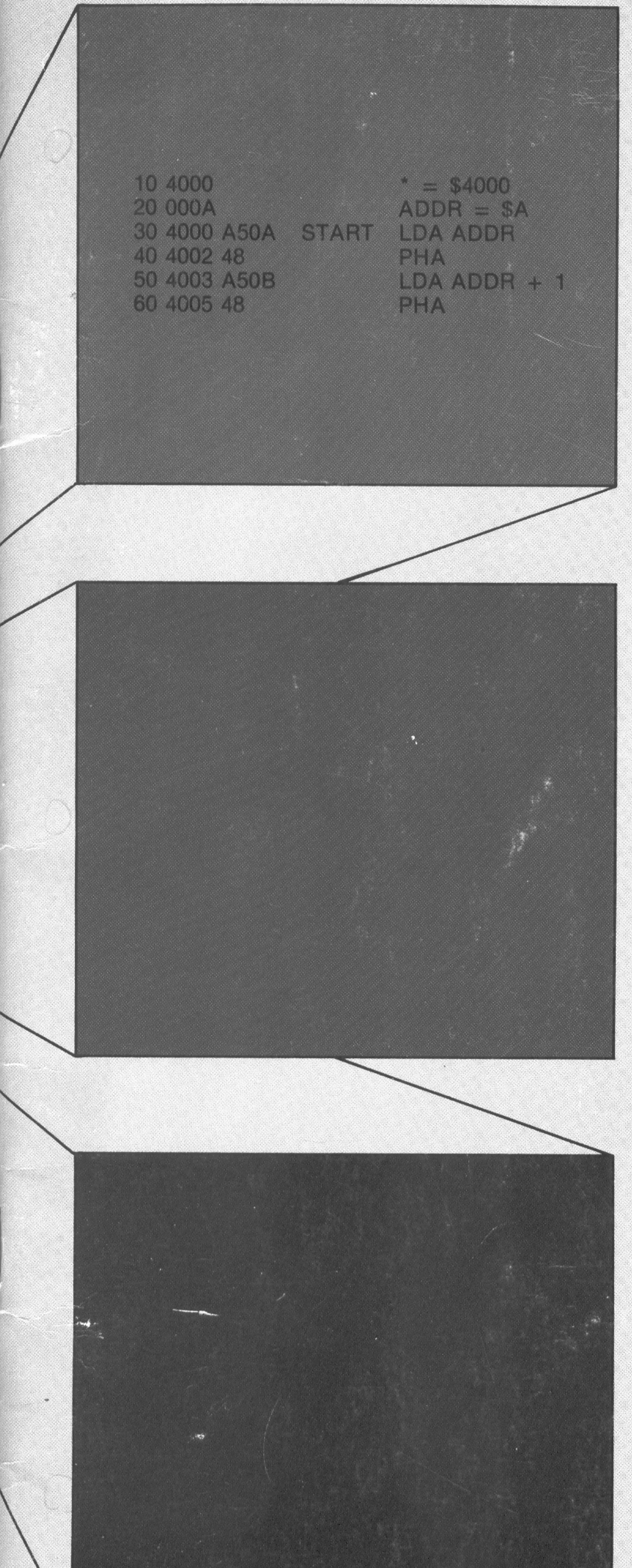

ASSEMBLER EDITOR AND EXTENDED MONITOR REFERENCE MANUAL

(C1P, C4P and C8P)

OHIO SCIENTIFIC SOFTWARE

$6.95

OHIO SCIENTIFIC

ASSEMBLER EDITOR and EXTENDED MONITOR REFERENCE MANUAL

Although great care has been taken in the preparation of this manual to insure the technical correctness, no responsibility is assumed by Ohio Scientific for any consequences resulting from the use of its contents. Nor does Ohio Scientific assume any responsibility for any infringements of patents or other rights of third parties which may result from its use.

If you discover misprints or errors, please send a letter to the attention of: Documentation Department, Ohio Scientific Inc., 1333 S. Chillicothe Road, Aurora, Ohio, 44202

CONTENTS

CHAPTER PAGE

APPENDICES

CHAPTER 1

INTRODUCTION

This manual is intended to be an introductory and reference manual for entering, editing, debugging and running assembly language programs using the OSI Assembler/Editor and Extended Monitor. This manual is not intended to be an introduction to assembly language programming. See Appendix O for a list of introductory texts. We shall assume that the reader is familiar with binary and hexadecimal numbers, the two's complement system for storing negative numbers, the architecture of the 6502 microprocessor (registers, flags, the stack, memory organization) and the rudiments of 6502 assembly language. In this manual we will use the following conventions.

<CR>	The carriage return key on the keyboard
<LF>	The line feed key on the keyboard
<BREAK>	The break key on the keyboard
↑	An up-arrow. May be ↑, ∧, or shift-N on some keyboards
@	The commercial-at. May be shift-P on some keyboards

We will use the following terms:

BYTE—The standard eight bit unit of storage.

MACHINE LANGUAGE—The language the microprocessor understands. For the 6502, each machine language instruction occupies one to three bytes of memory. Hence, a machine language instructor is one 8 bit number, two 8 bit numbers or three 8 bit numbers. When the microprocessor is running, it is always a machine language program that is running.

ASSEMBLY LANGUAGE—A symbolic language in which every line of a program translates into one machine language instruction. This is in contrast to high level languages like BASIC or PASCAL in which each line corresponds to many machine language instructions. While a machine language program consists entirely of groups of 8 bit numbers, in an assembly language program, the programmer may use symbolic names (mnemonics) for the machine instructions.

ASSEMBLER—The program which translates an assembly language program into machine language. The OSI Assembler/Editor also contains features which allow the editing of assembly language programs. While programmers sometimes say they are writing programs in "Assembler," it is technically more correct to say assembly language. An assembler is a large program, which functions somewhat like a compiler.

ADDRESS—The memory of a 6502 computer is organized into bytes. Each of these bytes has a unique number associated with it called its address. Addresses are usually written as 4-digit hexadecimal numbers. The first address is $0000. ($ denotes a hexadecimal number.)

PAGE—Memory is organized into large units called pages. Each page is 256 bytes. Page 0 consists of those bytes with addresses $0000 through $00FF, page 1 is those bytes with addresses $0100 through $01FF, and so on. Page 0 is important on the 6502, in that instructions which refer to page 0 are shorter and execute faster than instructions that refer to other parts of memory.

EFFECTIVE ADDRESS—Most 6502 instructions make reference to some byte of memory. The address of that byte is called the effective address for that instruction.

OBJECT CODE—Machine language. Generally this refers to the result of assembly of an assembly language program.

SOURCE CODE—The assembly language program to be assembled. The assembler translates source into object.

FILE—A program or group of data. Thus a file may be an assembly language file, a machine language file or a source file.

LOCATION COUNTER—When the assembler is assembling a source file, it keeps track of where in memory (the location) the object code is being put. The location counter is where that location is kept. Hence, the location counter is a place in memory that contains an address.

CHAPTER 2

BUILDING BLOCKS OF ASSEMBLY LANGUAGE

Each 65Ø2 assembly language statement is composed of one or more parts. Each part is built from the following:

A. OPERATION SYMBOLS

These are three-character codes which the assembler translates directly into a machine language operation code. For example, TXA (*T*ransfer register *X* to register *A*) is the operation symbol for the 65Ø2 operation code (opcode) $8A.

B. CONSTANTS

A constant is a number. On the 65Ø2, a constant is usually an 8 bit (one byte) or 16 bit (two bytes) number. The programmer may use numbers in decimal, binary, octal or hex (hexadecimal) and may also use character string constants, which are stored in memory in the 8 bit ASCII code. These codes are listed in Appendix J.

1) A decimal constant has no prefix.
2) A binary constant is prefixed by a per cent sign (%).
3) An octal constant is prefixed by a "commercial at" (@) sign.
4) A hex constant is prefixed by a dollar sign ($).

Thus, for example:

26 = %11Ø1Ø = @32 = $1A

5) A character string constant is a string of characters enclosed in single quotes (apostrophes). (If an apostrophe is to be included in the string then two consecutive apostrophes must be used.) Each character in the string is stored in one byte in memory in the ASCII code. For example, 'A/3' is stored as $41 2F 33 and 'P''R' is stored as $5Ø 22 32.

C. LABELS AND EXPRESSIONS

A label or symbol is a group of characters which looks like a variable name in BASIC or FORTRAN. A label consists of one to six characters from the set

A-Z
Ø-9
.
:
$

The first character in a label must be a letter. The characters must be contiguous, that is, there must be no blanks between the characters. The single letters A, X, Y, S and P cannot be used as labels, since they refer to the 65Ø2 registers, and no operation symbols (like JMP, SEC, BNE) can be used as labels. Examples:

LEGAL	ILLEGAL
LOOP	POINTER
END.2	$PR
TABL$	A/B
START:	LOO P
XY	A
LDAX	LDA

The programmer may also use arithmetic expressions in a program. An expression may be like an arithmetic expression in BASIC with no parentheses and no exponentiation allowed. Evaluation of an expression is always done left to right without regard to precedence of operators. Expressions are always evaluated to an answer of 16 bits or less, with any overflow ignored. Division is integer division with any remainder ignored. An expression cannot begin with a minus sign, however, an expression like Ø-1 is allowed, with the answer appearing in two's complement form if it is negative. For example, if Q=$5ØAF and D=64 then the

EXPRESSION	EVALUATES TO
Q/$1ØØ	$ØØ5Ø
Q*256	$AFØØ
Q/256*256	$5ØØØ
D+$ØA/%1Ø1Ø	$ØØØ7

The assembler also recognizes one predefined label, an asterisk (*), which denotes the current contents of the location assignment counter. The assembler can tell from context if an asterisk means this or multiplication.

CHAPTER 3

ASSEMBLY LANGUAGE STATEMENTS

There are three kinds of statements that can be entered into an assembly language source program: (a) remarks, (b) instructive statements and (c) directive statements. Each line must begin with a line number.

A. REMARKS

A line that begins with a semicolon (;) is a comment. Remarks are printed in any listing, but they do not affect the object code produced during assembly. In addition, any line in a source program can contain comments, as described below.

B. INSTRUCTIVES

These are the actual assembly language instructions that translate directly into machine language code. An assembly language statement has up to four parts called fields. The general form is

label   operation symbol   operand   remarks

The label and remarks fields are always optional. Some statements require that the operand field be blank. The fields may begin in any column and they are separated by blanks. It is, however, good practice to tabulate the four fields in columns. See Appendix D for tab characters.

For the statements that require an operand, the operand is either (i) the data for the instruction, or (ii) the Effective Address, or (iii) the information needed to calculate the Effective Address. To facilitate some examples we will next describe two directive statements. These will be described in greater detail in the following section.

The form of the .BYTE statement is

label   .BYTE   operand   remarks

The label and remarks fields are optional. The *operand* may be any expression. This directive causes the assembler to generate a one byte constant at the current location in the program.

The equals (=) directive causes a label to take a value which is used throughout assembly. The form is

label = expression   remarks

or

** = expression   remarks*

The second form sets the location counter to the value of the expression, and thus tells the assembler where to put the machine language code when the program is assembled.

Now back to assembly language instructive statements. The 6502 assembly language has five different addressing forms:

1. DIRECT ADDRESSING

The form is

label   op symbol   operand   remarks

The label and remarks fields are optional. The value of the operand is the Effective Address. The instructions that can be used in the direct addressing mode are

ARITHMETIC	DATA MOVE	BOOLEAN	COMPARING	SHIFTING	BRANCHING
ADC	LDA	AND	CMP	ASL	BCC
DEC	LDX	BIT	CPX	LSR	BCS
INC	LDY	EOR	CPY	ROL	BEQ
SBC	STA	ORA		ROR	BMI
	STX				BNE
	STY				BPL
					BVC
					BVS
					JMP
					JSR

The shifting instructions, in addition to allowing an expression as operator, may also have an A as operand, indicating that the accumulator is to be shifted. For the arithmetic, data movement, boolean, comparing and shifting instructions, two bytes of machine code are generated if the operand is less than $100 (i.e. on page zero). If the operand is greater than or equal to $100, then the machine code is three bytes. Execution is faster if the operand is on page 0. If the operand is an A (for a shifting instruction) then only one byte is generated. Three bytes are always generated for a JMP or JSR instruction. Two bytes are always produced for the other branching instructions, but the difference between the address of the instruction and the Effective Address must be between −126 and 129.
Example:

```
10      *=$200
20NMBR1 .BYTE $1
30NMBR2 .BYTE $4
40NMBR3 .BYTE $8
50START LDA   *-2
60NEXT  LDA   NMBR1+1
70      LDA   NEXT-5
80      LDA   $201
90      LDA   START+$400/3
```

The directive at line 10 causes the location counter to be initialized to $0200, so the first byte of data is placed at address $0200, that is, NMBR1=$0200. Hence, when this source code is assembled, the assembler will place a 1 in the byte with address $0200. Similarly, NMBR2=$0201 and a 4 is placed at that byte and NMBR3=$0202 and contains an 8. The instruction at line 50 is assembled into three bytes beginning at address $0203, hence START=$0203, NEXT=$0206 and the * in line 50 refers to the address of that instruction, so, at that point, *=0203. Therefore, the instructions at lines 60-90 are all the same, that is, each loads the accumulator with the contents of the byte whose address is $0201. After execution of any one of these, then, the contents of register A would be 4.

2. IMMEDIATE ADDRESSING

The form is

label *op symbol* *#operand* *remarks*

The operand can be any expression. The pound sign (#) is the indicator that this instruction is in the immediate addressing mode. In the direct mode, the value of the operand is interpreted to be a memory address. In the immediate mode there is no Effective Address. If the value of the operand is larger than 8 bits then any extra bits (on the left) are ignored. The instructions which can be used in the immediate mode are ADC, AND, CMP, CPX, CPY, EOR, LDA, LDX, LDY, ORA and SBC.
Example:

```
10      *=$400
20      NUM=$F
30      BIG=$285
40BEGIN         ADC #10
50              ADC #NUM-5
60              ADC #BIG/$100+@10
70              ADC #BIG*2
80              ADC #BEGIN/32-22
```

In hex,

```
BEGIN = $400
BIG/$100 = $2
BIG*2 = $50A
```

and

```
BEGIN/32 = $20
```

Each instruction (lines 40-80) adds, with carry, a decimal 10 to the accumulator.
Example:

```
10      *=$20
20DATA  .BYTE 255
30      LDX   #DATA
40      LDX   DATA
```

In this example, DATA=$20. In line 30, DATA itself is the data, so this instruction puts the decimal number 32 into register X. In line 40, DATA is the Effective Address, so this instruction puts a decimal 255 into register X.

3. INDEXED ADDRESSING

The form is

label *op symbol* *operand*,X *remarks*
or
label *op symbol* *operand*,Y *remarks*

In the first form, the effective address is the value of the operand plus the contents of register X. That is,

Eff. Ad. = *operand* + C(X)

In the second form

Eff. Ad. = *operand* + C(Y)

The value of the register is always taken to be non-negative, so $0 \le C(X) \le 255$ and $0 \le C(Y) \le 255$. Some indexable instructions can be indexed by either the X register or the Y register and some can be indexed by only one. See Appendix B.

Example

```
100      *=$100E
120UNO   .BYTE 5
130DUO   .BYTE 7
140TRES  .BYTE 9
150START LDX    #1
160      LDA    DUO,X
170      LDX    #2
180      LDA    UNO,X
190      LDY    #1
200      LDA    START-2,Y
```

For each LDA instruction (lines 160, 180 and 200) the effective address is $1010, so each puts a 9 into the accumulator.

4. INDIRECT ADDRESSING

The form is

label *op symbol* *(operand)* *remarks*
or
label *op symbol* *(operand,*X) *remarks*
or
label *op symbol* *(operand),*Y *remarks*

Only the JMP instruction can be used in the first form. The second form is called indexed indirect and the third is indirect indexed.

In the first form Eff. Ad. = C(operand)

In the second form Eff. Ad. = C(operand + C(X))

In the third form Eff. Ad. = C(operand) + C(Y)

In each the operand must be less than $100, i.e., the operand must be on page 0. For these instructions, the operand is taken to be the address of a .WORD, that is, a two byte number with the first byte containing the eight low order bits and the second byte containing the eight high order bits of the Effective Address, below. Hence the two bytes of the 16-bit number are in reverse order. For example, if C($001B)=$FF and C($001C)=$2A, then the Effective Address for the instruction

```
JMP ($001B)
```

is $2AFF.

Example:

```
5; Page 0 constants
10         * = $80
15ADDR1      .BYTE $C1
20           .BYTE $12
2 5ADDR2     .BYTE $C0
30           .BYTE $12
35;
40         * = $12C0
45; more constants
50K1         .BYTE $FA
55K2         .BYTE $FB
60K3         .BYTE $FC
65K4         .BYTE $FD
70           LDX  #2
75           LDY  #2
80           LDA  ADDR1
85           LDA  ADDR1,X
90           LDA  ADDR1,Y
95           LDA  (ADDR1,X)
100          LDA  (ADDR1),Y
```

The effect of each LDA instruction is as follows:

LINE	EFF. AD.	ACCUM (After Execution)
80	$80	$C1
85	$82	$C0
90	$82	$C0
95	$12C0	$FA
100	$12C3	$FD

5. IMPLIED ADDRESSING

The form is

label *op symbol* *remarks*

These instructions have no operand. They generally refer to an operation on a flag, a register or a register pair. Some instructions of this type are SEC (*SE*t the *C*arry flag), INX (*IN*crement the *X* register) or TXS (*T*ransfer register *X* to the *S*tack pointer). Each instruction in this addressing mode produces one byte of machine language.

C. DIRECTIVES

These assembly language statements do not translate into 6502 machine code. Directives are used to tell the assembler where in memory to put the object code, define labels and set up data locations in memory.

1. THE LOCATION COUNTER

The form is

*=*expression*

where the expression may contain an *.

For example:

```
10  * = $440B
20      LDA #%101
30DATA1      .BYTE $1A
40  * = * + 2
50DATA2      .BYTE $F0
```

The op code for an LDA instruction in the immediate mode is $A9, hence the LDA instruction is assembled to $A905. When the program is assembled, the machine code produced is:

	ADDRESS (Hex)	CONTENTS (Hex)
	$440B	$A9
	$440C	$05
DATA1 =	$440D	$1A
	$440E	?
	$440F	?
DATA2 =	$4410	$F0

The directive on line 40 causes the assembler to skip the two bytes with addresses $440E and $440F, so the contents of these bytes are not changed at assembly time. The statement

```
* = * + 2
```

allows the programmer to refer to DATA1 as an array of three elements with an index register (X or Y) acting as subscript.

2. DEFINING LABELS

The form is

label = *expression*

Example:

```
10  *=$1BF8
20  W=$12
30  E=2*W+3
40  START   LDA #E-W/3
50  J=*-1
```

The labels in this example have the following values:

LABEL	VALUE (Hex)
W	$12
E	$27
START	$1BF8
J	$1BF9

When this example is assembled, the location counter has the value $1BF8 before line 40 is assembled and the value $1BFA after line 40 is assembled. Hence, when line 50 is processed, *=$1BFA. Since E is defined in terms of W, an assembly error would result if lines 20 and 30 were interchanged.

3. DATA LOCATIONS

The assembler recognizes three directives which may be used to set up memory locations with data. The .BYTE directive is used to define one byte of data and .DBYTE and .WORD set up two bytes, with .WORD producing data with the bytes in reverse order, as required for indirect addressing.

a. .BYTE

The form is

label .BYTE *operand* *remarks*

The operand may be one part or several parts separated by commas. There must be no blanks (except in quotes) anywhere in the operand because a blank is used to separate the operand and remarks fields. Each part of the operand is either an expression or a string of characters enclosed in single quotes. Each expression or character in quotes produces one byte of data in memory. If the value of an expression is more than 8 bits then only the rightmost 8 bits are used.

Example:

```
10  *=$0F0E
20  C=15
30 Q1   .BYTE 10,$10,@10,%10
40 Q2   .BYTE C-3,Q1/$10
50 Q3   .BYTE 'OSI',0
60 Q4   .BYTE C/2-8,Q1/$4
```

The result, in memory, when this code is assembled, would be

ADDRESS	CONTENTS
Q1=$0F0E	$0A
$0F0F	$10
$0F10	$08
$0F11	$02
Q2=$0F12	$0C
$0F13	$F0
Q3=$0F14	$4F
$0F15	$53
$0F16	$49
$0F17	$00
Q4=$0F18	$FF
$0F19	$C3

b. .DBYTE

The form is

label .DBYTE *operand* *remarks*

This directive causes the assembler to place a two byte constant into memory. The operand may be a single expression or several expressions separated by commas. Character strings in quotes are not allowed.
Example:

```
10  *=$1E00
20  T=$011D
30 K1  .DBYTE T,T-$122
40 K2  .DBYTE K1,*-1
```

Assembly of this code would produce the following:

ADDRESS	CONTENTS
K1=$1E00	$01
$1E01	$1D
$1E02	$FF
$1E03	$FB
K2=$1E04	$1E
$1E05	$00
$1E06	$1E
$1E07	$05

c. .WORD
The form is

label .WORD *operand* *remarks*

The syntax is the same as for .DBYTE. This directive also produces a two byte constant, but the bytes are stored in reverse order.
Example:

```
10  *=$1E00
20  T=$11D
30 K1 .WORD T,T-122
40 K2 .WORD K1,*-1
```

Notice the operands are the same as the last example. Assembly would produce

ADDRESS	CONTENTS
K1=$1E00	$1D
$1E01	$01
$1E02	$FB
$1E03	$FF
K2=$1E04	$00
$1E05	$1E
$1E06	$05
$1E07	$1E

In 6502 machine language, addresses must be stored in this "backwards" fashion. For example the three byte instruction

```
CMP $17FA
```

is stored in memory as

$CD	operation code
$FA }	address
$17 }	

CHAPTER 4

BUILDING AND EDITING AN ASSEMBLY LANGUAGE SOURCE FILE

The Assembler is loaded by typing ASSEMBLER (or ASM) in response to the A* prompt in the OS-65D DOS mode. (This mode is reached by typing EXIT when in BASIC.) There are several commands that are accepted by the Assembler/Editor. The Assembler/Editor is waiting for a command when a period (.) is printed.

The user may enter a line into the current source file by typing a line beginning with a line number. The line is automatically inserted into the source file at the place specified by the line number. If a line is entered with the same number as a line which is already in the source file, then the new line replaces the existing line. Line numbers must be no larger than 65535. Besides entering a line into the current source file, the user may also use one of the following editing commands. Each command may be abbreviated to its initial letter.

RESEQ — Resequences all line numbers in the source file. The first line is assigned line number 10 and the line numbers increment by 10. After the resequence is finished, the next sequential line number is printed.

PRINT — Lists all or part of the current source file. PRINT may be used in the following forms:

PRINT	lists the entire file
PRINT *line*	lists one line
PRINT *first line - last line*	lists the specified lines
PRINT *first line -*	lists from the specified line to the end of the file
PRINT *- last line*	lists from the beginning of the file to the specified line.

Any number of the above line specifications, separated by commas, can be used with one PRINT command.

To direct the output to a printer refer to the DOS IO command the the IO flag bit settings (Appendix L).

DELETE — Deletes a line or lines from the file using the same line specifications as PRINT.

INIZ — Deletes all lines from the source file. To prevent inadvertently clearing the workspace, the question "INIZ? (Y/N)" is printed after a line beginning with an "I" is entered. The user must enter "YES" (or "Y") to complete initialization.

When entering commands or source text lines, corrections can be made to a line anytime before the carriage return. A back-arrow (or Shift O) can be used to delete single characters. An up-arrow (or Shift N) can be used to delete the entire line from the file.

NOTE: CTRL-P toggles device #4 on and off

CHAPTER 5

ASSEMBLY COMMANDS

The Assembler recognizes four assembly commands. Three of the commands give object code listings, and the fourth assembles the source to object code in memory, ready (hopefully) for execution. The commands are:

A0 (or A) Gives a full assembly listing. Each line printed contains:

(1) the line number
(2) the address in memory of the object code
(3) the object code (1-3 bytes)
(4) the source code

If errors are detected in the source, a pointer to the error and the appropriate error number are printed below the line with the error. The machine code generated in case of an error depends on the type of error, but, generally, is either the appropriate op code byte with a zero operand or is three NOP bytes. In many cases, this will result in correct addresses for the rest of the listing. The next section contains an example of an A0 listing.

A1 Gives an errors-only listing. This command produces the same output as the full assembly listing but for only those lines that contain errors.

A2 Gives an object tape listing in the standard checksum format. See Appendix K for a description of checksum format. The user may save this output on tape by typing

Save <CR> and then
A2 <CR>

(Note: "SAVE" is used only on cassette versions of assembler. See Appendix L for cassette I/O for disk machines.)

A3 Puts the object code into memory ready for execution. This command produces no listing, unless there are errors.

On disk systems, the M command may be used to change the place in memory where the object code is placed by the A3 command. This command does not affect the object code itself, only where it is put. For example, suppose the programmer wants to write a program which will be assembled to memory starting at address $3290. Thus the source program would have a line declaring *=$3290. However, an A3 command could not be executed, because the machine code produced would overwrite the source code and assembly would not be completed. This can be remedied by use of the M command to offset the address for the machine code. For example, if the programmer types

M1000<CR>

and then

A3<CR>

then the object code produced will be the same as without the M command, but it will be placed in memory starting at address $4290. The programmer can then use the disk command

!SAVE TT,S=4290/N

to save the machine language code, where TT is the track, S is the sector and N is the number of pages to be copied to the disk. The code may then (or later) be recalled to memory at the correct place for execution by the command

!CALL 3290=TT,S

from the Assembler/Editor or the Extended Monitor or by

DISK!"CA 3290=TT,S"

from BASIC.

CHAPTER 6

AN EXAMPLE

Suppose the programmer enters the following program through the keyboard. The program is a screen clearing program using indirect addressing.

```
P
 10          *=$4000
 20          ADDR=$A
 30START     LDA ADDR        ; save the page 0 locations
 40          PHA             ; in case this routine is
 50          LDA ADDR+1      ; called from BASIC
 60          PHA
 70          LDA #$D0        ; set up page 0 locations
 80          STA ADDR+1      ; for indirect addressing
 90          LDA #0
100          STA ADDR
110          LDX #7          ; counter
120          LDY #0          ; register for ind. addressing
130          LDA #32         ; blank character in ASCII code
140LOOP      STA (ADDR),Y
150          INY
160          BNE LOOP
170          INC ADDR+1      ; after 256 locations incr. page
180          DEX
190          BPL LOOP
200          PLA             ; recover the page 0 info
210          STA ADDR+1      ; & put it back
220          PLA
230          STA ADDR
240          RTS
250          .END
```

Note: On a C1P computer change:

```
line 90 to LDA #83
line 110 to LDX #3
```

If the user then enters the A command, the ouput will be:

```
A
 10 4000                   *=$4000
 20 000A                  ADDR=$A
 30 4000 A50A  START      LDA ADDR        ; save the page 0 locations
 40 4002 48               PHA             ; in case this routine is
 50 4003 A50B             LDA ADDR+1      ; called from BASIC
 60 4005 48               PHA
 70 4006 A9D0             LDA #$D0        ; set up page 0 locations
 80 4008 850B             STA ADDR+1      ; for indirect addressing
 90 400A A900             LDA #0
100 400C 850A             STA ADDR
110 400E A207             LDX #7          ; counter
```

```
120 4010 A000              LDY #0          ; register for ind. addressing
130 4012 A920              LDA #32         ; blank character in ASCII code
140 4014 910A   LOOP       STA (ADDR),Y
150 4016 C8                INY
160 4017 D0FB              BNE LOOP
170 4019 E60B              INC ADDR+1      ; after 256 locations incr. page
180 401B CA                DEX
190 401C 10F6              BPL LOOP
200 401E 68                PLA             ; recover the page 0 info
210 401F 850B              STA ADDR+1      ; & put it back
220 4021 68                PLA
230 4022 850A              STA ADDR
240 4024 60                RTS
250                        .END
```

For example, the third line is

```
30      4000      A50A      START      LDA      ADDR      ; save the page 0 locations
|       |         |         |          |        |         |remarks
|       |         |         |          |        |operand
|       |         |         |          |operation symbol
|       |         |         |label
|       |         |machine language for this line
|       |address of the first byte occupied by this instruction
|line number
```

There are no errors in the above assembly.
If, at this point, the A3 command is entered, no output will result. The assembler will, however, put the machine code into memory at addresses $4000 through $4024. If the user (on a disk) system enters

M0800

and then

A3

the resulting machine code will be exactly the same but will be put at addresses $4800 through $4824.

If the user enters the A2 command, the output will be the following, in checksum format for tape storage. See Appendix K for a description of checksum format.

```
; 184000A50A48A50B48A9D0850BA900850AA207
  A000A920910AC8D009D4
; 0D4018FBE60BCA10F668850B68850A600670
```

Assume next that the program is entered as below. Lines 70, 80, 140 and 190 have been changed so that they contain errors.

```
P
  10              *=$4000
  20             ADDR=$A
  30   START     LDA ADDR        ; save the page 0 locations
  40             PHA             ; in case this routine is
  50             LDA ADDR+1      ; called from BASIC
  60             PHA
  70             LDA #D0         ; set up page 0 locations
  80             STA ADR+1       ; for indirect addressing
  90             LDA #0
 100             STA ADDR
 110             LDX #7          ; counter
 120             LDY #0          ; register for ind. addressing
 130             LDA #32         ; blank character in ASCII code
 140   LOOP      STA (ADDR,Y)
 150             INY
 160             BNE LOOP
 170             INC ADDR+1      ; after 256 locations incr. page
 180             DEX
```

```
190         BPK LOOP
200         PLA              ; recover the page 0 infor
210         STA ADDR+ 1      ; & put it back
220         PLA
230         STA ADDR
240         RTS
250         .END
```

This time the result of an A command will be:

```
A

  10 4000                        *=$4000
  20 000A=                       ADDR=$A
  30 4000 A50A START             LDA ADDR        ; save the page 0 locations
  40 4002 48                     PHA             ; in case this routine is
  50 4003 A50B                   LDA ADDR+1      ; called from BASIC
  60 4005 48                     PHA
  70 4006 A900                   LDA #D0         ; set up page 0 locations
-------------------------------------------------------------^
E#  18
  80 4008 8D0100                 STA ADR+1       ; for indirect addressing
-------------------------------------------------------------^
E#  19
-------------------------------------------------------------^
E#  18
  90 400B A900                   LDA #0
 100 400D 850A                   STA ADDR
 110 400F A207                   LDX #7          ; counter
 120 4011 A000                   LDY #0          ; register for ind. addressing
 130 4013 A920                   LDA #32         ; blank character in ASCII code
 140 4015 EAEAEA LOOP            STA (ADDR,Y)
-------------------------------------------------------------^
E#  7
 150 4018 C8                     INY
 160 4019 D0Fa                   BNE LOOP
 170 401B E60B                   INC ADDR+1      ; after 256 locations incr. page
 180 401D CA                     DEX
 190 401E EAEAEA                 BPK LOOP
-------------------------------------------------------------^
E#  6
 200 4021 68                     PLA             ; recover the page 0 info
 210 4022 850B                   STA ADDR+1      ; & put it back
 220 4024 68                     PLA
 230 4025 850A                   STA ADDR
 240 4027 60                     RTS
 250                             .END
```

An A1 command will give the following:

```
A1
    70 4006 A900                LDA D0          ; set up page 0 locations
--------------------------------------------------------^
E#  18
    80 4008 8D0100              STA ADR+1       ; for indirect addressing
--------------------------------------------------------^
E#  19
--------------------------------------------------------^
E#  18
   140 4015 EAEAEA LOOP             STA (ADDR,Y)
........................................................^
E#  7
   190 401E EAEAEA              BPK LOOP
........................................................^
E#  6
```

CHAPTER 7

RUNNING A MACHINE LANGUAGE PROGRAM

After an assembly language source program has been assembled to memory by the A3 command or a machine language program has been called into memory from disk or tape, there are several options for running and testing. The most powerful debugging tool is the Extended Monitor, which is described in the next section. The procedure for interfacing a machine language program with a BASIC program is also discussed in Chapter 9.

A. PROM MONITOR-CASSETTE BASED SYSTEMS

On a cassette based system, the user may exit from the Assembler/Editor and enter the machine language Monitor in ROM by typing <BREAK> and then

```
M
```

A machine language program in memory may then be run by typing the entry address and then

```
G
```

The user may return from the Monitor in ROM by typing

```
.1300 G
```

provided memory from addresses $0240 through $1390 has not been altered. The Monitor in ROM commands are discussed more completely in Appendix I.

B. DOS KERNEL-DISK BASED SYSTEMS

On a disk based system, the user may type

```
EXIT (or E)
```

to enter the DOS kernel and then type

```
GO XXXX
```

where XXXX is the entry address of the machine language program in hex. If the user's program ends with an RTS then control will revert to the DOS kernel. (When in the DOS, the A* prompt appears.)

C. PROM MONITOR-DISK BASED SYSTEMS

Also a disk system, the user may exit to the Monitor in ROM by typing

```
EXIT
```
and then
```
RE M
```

The user may return from the Monitor in ROM to the DOS kernel by typing .2547 G.

CHAPTER 8

THE EXTENDED MONITOR

The 6502 Extended Monitor is an extensive machine code debugging aid. It includes the following commands for

- memory display and modification
  - memory display and change
  - memory dump
  - memory fill
  - memory move
  - memory relocate
- program debugging
  - disassembly
  - search for a byte string
  - search for a character string
  - breakpoint installation and control
  - processor register display and change
  - program execution
- audio cassette input/output
  - load
  - save
  - view
- hexadecimal arithmetic
  - calculate
  - display overflow/remainder

LOADING THE EXTENDED MONITOR

The method for loading the Extended Monitor depends upon which version you are using. Refer to the appendix appropriate to your system (Appendix G or H).

After the Extended Monitor has been loaded, its prompter, a colon (:), is displayed. This is the Extended Monitor's command mode.

A. THE EM COMMANDS

Each of the Extended Monitor commands is listed below. Any of these commands may be entered whenever you are in the command mode as indicated by the colon prompter. Many of the commands also have subcommands which can be entered only after the primary command has been entered. If an invalid command is entered, a "?" will be printed.

In the command descriptions below, all addresses and data values are hexadecimal and the following abbreviations or special characters are used:

MEANING

<LF>	the line feed key on the keyboard
<CR>	the carriage return (or return) key on the keyboard
↑	an up arrow character. May be a ↑, ∧ or a shift/N on some keyboards
@	a commercial-at character. May be a shift/P on some keyboards

MEMORY DISPLAY AND MODIFICATION COMMANDS

@aaaa — displays the address and contents of the location aaaa. New contents may or may not then be entered (two hex digits) followed by one of the following:

<LF>	displays the next location
↑	displays the previous location
/	displays the same location
"	prints the contents of the location as an ASCII or graphic character
<CR>	exits to the Extended Monitor command mode

Dffff,tttt — dumps the contents of memory locations ffff through tttt-1.

Fffff,tttt=dd — fills memory locations ffff through tttt-1 with the value dd.

Maaaa=ffff,tttt — moves the contents of memory from locations ffff through tttt-1 to the memory starting at location aaaa.
NOTE: The distance of an upward

move must be greater than the length of the move or data in the original locations will be overwritten (aaaa> =tttt or aaaa< ffff).

Raaaa=ffff,tttt relocates (moves the contents of memory from locations ffff through tttt-1 to the memory starting at location aaaa and appropriately adjusts all three-byte 65Ø2 instruction operand addresses that refer to locations within the range of the move. (Adds (aaaa-ffff) to each operand address that is >=ffff and <=tttt-1).
Note: The Distance of an upward move must be greater than the length of the move or data in the original locations will be overwritten (aaaa>=tttt or aaaa<ffff).

PROGRAM DEBUGGING COMMANDS

Qffff disassembles 65Ø2 machine code into 65Ø2 mnemonic code from memory location ffff up. Disassembly continues for a total of 24 lines—a maximum of 72 bytes. At completion, it awaits,
<LF> disassembles the next 24 lines, or
<CR> exits to the Extended Monitor command mode

Ndd. . .dd>ffff,tttt searches the contents of memory locations ffff through tttt-1 for the string of 1 to 3 data bytes dd. . .dd. If the string is found then the address of the first byte of the first occurrence of the string is displayed and the @ mode is entered.

Wc. . .c>ffff,tttt searches the contents of memory locations ffff through tttt-1 for the string of 1 to 6 ASCII characters c. . .c. If the string is found then the address of the first byte of the first occurrence of the string is displayed and the @ mode is entered.

Bn,aaaa installs breakpoint n (n = 1-8) at address aaaa. The contents of location aaaa is saved and may be restored with the En command. If breakpoint n had previously been assigned it is first restored. When a breakpoint is "hit" during program execution it is also automatically restored. (See Using Breakpoints for Program Debugging)

En eliminates breakpoint n (n = 1-8) and restores the original contents of the location where it was located.

Gaaaa goes (transfers program control) to address aaaa.

T prints a table of breakpoint addresses for each breakpoint 1 through 8. An address of FFFF indicates an unassigned breakpoint.

C continues program execution from the location of the last breakpoint. This command must only be used after a breakpoint has been "hit." The byte that was replaced by the breakpoint (and restored when the breakpoint was hit) is executed first.

I prints the address of the last breakpoint "hit" and the contents of the A, X, Y, processor status (P) and stackpointer (K) registers as they existed at that breakpoint.

A, X, Y, P, K these five commands print the contents the associated register. New contents may or may not then be entered (two hex digits) followed by one of the following:
" prints the contents of the register as ASCII or graphic character
<CR> exits to the Extended Monitor command mode

AUDIO CASSETTE COMMANDS

Sffff,tttt saves the contents of memory locations ffff through tttt-1 by writing them to the cassette port (as well as the terminal) in checksum format. This function may be terminated by typing "L" and a

	space. See Appendix K for a description of checksum format.
L	loads into memory the data read from the cassette port in checksum format. If a checksum error is detected, "ERR" is printed. To recover, stop the cassette machine, rewind the tape a short distance and restart playing it. Type an "L" to restart the loading. The LOAD command can be exited at any time by typing a space.
V	view the data read from the cassette port in checksum format. Same as Load, above, but displays the data without modifying memory.

CALCULATOR COMMANDS

Hxxxx,yyyy+	calculates the sum of the hexadecimal values xxxx and yyyy and prints the result.
Hxxxx,yyyy−	same as above for difference.
Hxxxx,yyyy*	same as above for product.
Hxxxx,yyyy/	same as above for quotient.
O	prints the overflow or remainder from the last multiplication or division performed with the H command. NOTE: at most 17 characters per command line are allowed.

B. THE R AND M COMMANDS

The M command moves the contents of one area of memory to another area, without change. The R command moves memory and changes the contents of those locations which can be interpreted to be the address portion of a three byte machine language instruction. This address portion is changed only if the address lies within the range of the move. For example, consider the following sequence of instructions residing at address $0800 through $0810:

ADDRESS	INSTRUCTION
$800	LDA $2000
$803	JSR $809
$806	JMP $1000
$809	LDX $810
$80C	STA $D740,X
$80F	RTS
$810	.BYTE $A

If the command

M0A00=0800,0811

is executed, then the machine code for these instructions is moved unchanged to memory address $0A00 through $0A10. If the command

R0A00=0800,0811

is executed, then the code is moved to locations 0A00 through 0A10 and becomes

ADDRESS	INSTRUCTION
0A00	LDA $2000
0A03	JSR $A09
0A06	JMP $1000
0A09	LDX $A10
0A0C	STA $D740,X
0A0F	RTS
0A10	.BYTE $A

For the LDX and JSR instructions, the address part of the instruction is changed, because the two addresses involved ($809 and $810) are in the range of the move (in this case between 0800 and 0811). For the remaining three byte instructions, the address is not changed. If an operand is changed, then it is changed by the amount of the move, that is, if

Raaaa=ffff,tttt

is executed then

New operand = old operand + (aaaa − ffff)

The use of the R command may cause problems if some of the locations that are relocated do not contain machine language instructions, but contain data. For example, if the following three bytes appear as data in a program at addresses $810 through $812:

.BYTE $AD
.BYTE $7
.BYTE $8

and the command

R0A08=0800,0820

is executed, then the contents of these three bytes may be interpreted to be the machine language for the instruction LDA $807. Then the R command would change these to

.BYTE $AD
.BYTE $F
.BYTE $A

One way to prevent this is to use the R command to relocate the entire program and then use the M command on the bytes that contain data, to correct any mistakes like the above.

C. BREAKPOINTS AND DEBUGGING

As the name implies, a breakpoint is a point where the execution of a running program may be "broken" or interrupted. Using the Extended Monitor, up to eight breakpoints may be placed into a program. When the program is run (executed) and a breakpoint is encountered, the Extended Monitor is re-entered and prints the following to document the breakpoint:

```
Bn@aaaa
A/aa X/xx Y/yy P/pp K/kk
```

where:
- **n** is the breakpoint number 1-8
- **aaaa** is the location where the breakpoint was encountered
- **aa** is the contents of the accumulator
- **xx** is the contents of the X index register
- **yy** is the contents of the Y index register
- **pp** is the contents of the processor status word
- **kk** is the contents of the stackpointer

To illustrate the use of a breakpoint, consider the following program:

```
100              *=$4000
120 START        LDA #101
140              LDX #2
160              STA $D290,X
180              DEX
200              BNE *-4
220              STA $D29C
240              RTS
```

When this program is executed, it will print two lower case e's at the left margin of the screen and another near the center. An assembly listing (assembler A command) yields:

```
.A
    100 4000        $=$4000
    120 4000 A965   START  LDA #101
    140 4002 A202          LDX #2
    160 4004 9D90D2        STA $D290,X
    180 4007 CA            DEX
    200 4008 D0FA          BNE *-4
    220 400A 8D9CD2        STA $D29C
    240 400D 60            RTS
```

Assuming the user is working with the Assembler/Editor, the program may now be assembled to memory by the A3 command. The Extended Monitor may now be entered (on disk systems) by the command

```
!RETURN EM  (or !RE EM)
```

The computer will respond

```
EM V2.0
:
```

If the user now enters

```
B1, 4007
B2, 4008
B3, 400D
```

then three breakpoints will be installed in the program. The T command will produce the following listing:

```
B1,4007
B2,4008
B3,400D
B4,FFFF
B5,FFFF
B6,FFFF
B7,FFFF
B8,FFFF
```

Note: When you exit and re-enter EM, all breakpoints are initialized.

If the command

```
G4000
```

is entered, one "e" will be printed on the screen and the Extended Monitor will print

```
B1@4007
A/65 X/02 Y/FF P/7D K/FF
```

indicating that breakpoint #1 has been hit and also the status of the five registers when the breakpoint was encountered. The breakpoint B1 has now been removed and the DEX instruction has been put back into the program. If the C command is now entered, the program will continue execution of just one instruction, the DEX, the next breakpoint will be hit and the Extended Monitor will print

```
B2@4008
A/65 X/01 Y/FF P/7D K/FF
```

If the C command is entered again, then two more e's will appear and the Extended Monitor will print

```
B3@400D
A/65 X/00 Y/FF P/7F K/FF
```

All breakpoints have now been eliminated. If the user now enters

```
B1,400D
```

and then

```
X
```

the Extended Monitor will respond with

```
/00
```

which is the contents of register X at the time the last breakpoint was hit. If the user now types

```
0A
```

then that will be the contents of the X register when execution is resumed. If the user now types

```
G4004
```

then eleven "e's" will appear on the screen and the Extended Monitor will print:

```
B1@400D
A/65 X/00 Y/FF P/7F K/FF
```

The programmer can also change the flow of execution of the program. For example, if the user now enters

```
B1,4008
B2,400D
G4000
```

the Extended Monitor will respond

```
B1@4008
A/65 X/01 Y/FF P/7D K/FF
```

If the user now enters the C command, execution of the program will resume and the branch back to

```
STA D290,X
```

will be executed. If instead the programmer types

```
P
```

then the Extended Monitor will respond

```
/7D
```

which is the contents of the Processor Status Word at the time the breakpoint was hit. If the user now types

```
7F
```

this will be the contents of the Processor Status Word when execution resumes. Specifically, the Z flag will be set so that no branch takes place. Hence, if the C command is entered, one more e will appear on the screen, and the Extended Monitor will print

```
B2@400D
A/65 X/01 Y/FF P/7F K/FF
```

USING THE EM AND THE ASSEMBLER/EDITOR SIMULTANEOUSLY

On disk based systems, the Extended Monitor and the Assembler/Editor are always loaded into memory simultaneously. The user may go from one to the other by typing

```
!RE AS   or   !RE EM
```

The Extended Monitor and Assembler/Editor (on disk systems) occupy memory from $0200 through $16FF. The Extended Monitor uses page 0 locations $C0 through $FF.

CHAPTER 9

INTERFACING WITH BASIC

There are several methods that can be used to call a machine language routine from a BASIC program. If a routine is stored on disk at track TT and sector S, then a BASIC program may contain the statement

DISK!"CA XXXX=TT,S"

to bring the machine code into memory to hexadecimal addresses XXXX. The user should take precautions to avoid having a running BASIC program change memory locations occupied by his machine language subroutine, and not to bring in machine code onto your BASIC program. Beginning at $327E, in the workspace, the BASIC program and numeric variables are stored, however, string variables are stored at the end of memory so that the end of memory may not be a "safe" place for a machine language subroutine. The user can create a safe place by running the BASIC utility CHANGE.

A. THE USR FUNCTION

The user can cause a BASIC program to branch to any location in memory in exactly the same fashion that BASIC's built-in functions (like ABS, RND, SIN) are called. The appropriate form is

Y=USR(X)

where Y can be any arithmetic variable and X can be replaced by any arithmetic expression. The address of the entry point into the user's routine must be POKEd into memory locations 574 (=23E hex) and 575 (=23F hex). The low order byte of the address goes to 574 and the high order byte to 575. (This is the standard 6502 method of storing addresses backwards.)

When Y=USR(X) is executed, control passes to the POKEd address via a JSR and the value of X (or whatever the argument) is loaded into the Floating Accumulator, which is on page 0 at addresses $AE through $B3. See appendix M for the format of numbers in the Floating Accumulator. This is all that is done by BASIC and nothing is stored at Y unless the user's routine does it. The value in the Floating Accumulator, in floating point format, can be converted to a 16 bit integer (in two's complement if negative) by calling the routine whose address is stored at addresses $0006 and $0007. This can be done, for example, by

```
     LDA 6
     STA CALL+1
     LDA 7
     STA CALL+2
CALL JSR $FFFF
```

This routine will put its answer at $AE and $AF with the high order byte of the answer at $AE. If the user wants to store an answer at Y (assuming Y=USR(X) is in the BASIC program) then this 16 bit value should be put in the Y register (low byte) and the A register (high byte) and then the routine whose address is stored at $0008 and $0009 can be called.

B. DISK!"GO XXXX"

On disk based systems, a BASIC program may call a machine language subroutine by this statement, where XXXX is the entry address, in hex, of the machine language routine. The routine must end with an RTS. Parameters can be passed to such a routine (or a routine accessed by the USR function) using POKEs.

C. DISK!"XQT NNNNNN"

This command loads the disk file named NNNNNN to address $3279 up and then executes a JMP to $327E. Thus the program should be assembled to start at $327E. Header and track length information are stored at $3279-$327D. NNNNNN can be the name of a disk file or a track number. Since $327E is the beginning of workspace for assembly language programs, the programmer must offset the assembly to avoid destroying the source code during assembly. In addition, to allow the program to be stored on disk, the user must put, at address $327D, the number of tracks required to hold the machine language program. (One track holds 2040 bytes.) For purposes of example, let us assume the assembled program will use $200 (=512 decimal) bytes of memory and that the Assembler/Editor command

M1000 will cause the assembler to assemble the code without running into the source program in the workspace. The following sequence of commands will set up the disk file ready for a DISK!"XQT NNNNNN" command in a BASIC program. The user's input is underlined. We assume the program is in the workspace.

```
.M1000
.A3
.!RE EM
EM V2.0
:M 327E=427E,447E
:@327D
327D/dd 01
:!PUT NNNNNN
```

*Note: This discussion assumes that the workspace starts at $327E, which is correct for minifloppies. For eight inch floppies substitute $317E and subtract $100 from the above locations.

APPENDIX A

ASSEMBLY ERRORS

The following descriptions of assembly errors and their possible causes are provided to facilitate elimination of these errors in an assembly.

1) **A, X, Y, S and P are Reserved Names**
   One of these reserved names was found in the label field. No code is generated for a statement with a reserved name as a label. Use of a reserved name in an expression will give an "undefined label" error, error 18.
2) There isn't any.
3) **Address Not Valid**
   An address greater than 65535 (hex FFFF) was encountered.
4) **Forward Reference In Equate, Origin or Reserve Directive**
   An expression used in one of these directives includes a label that hasn't been previously defined in the assembly source file.
5) **Illegal Operand Type For This Instruction**
   An operand was found which is not defined for the specified instruction opcode. Refer to Appendix B for the defined instruction addressing modes.
6) **Illegal or Missing Opcode**
   A defined opcode was not found. Refer to Appendix B for the defined opcodes.
7) **Invalid Expression**
   An expression was found that is not a valid sequence of numerical constants and/or labels separated by valid operators or is not a valid instruction operand form.
8) **Invalid Index—Must Be X Or Y**
   An indexable instruction was found with an invalid index. Refer to Appendix B.
9) **Label Doesn't Begin With Alphabetic Character**
   A non-alphabetic character was encountered where a label was expected.
10) **Label Greater Than Six Characters**
    A string of more than six valid label characters (A-Z, Ø-9, $, ., :) was found before a non-valid label character. This is a warning message. Assembly continues using the first six characters of the label.
11) There isn't any.
12) **Label Previously Defined**
    The identified label has previously occurred in the assembler source file or this occurrence of the label had a different value on pass one than on pass two. The latter error may be caused by previous errors in the assembly.
13) **Out Of Bounds On Indirect Addressing**
    An indirect-indexed or indexed-indirect address does not fall into page zero as required.
14) There isn't any.
15) **Ran Off End Of Line**
    An operand is required and wasn't found before the end of the line.
16) **Relative Branch Out Of Range**
    The target address of a branch instruction is farther away than the minus 128 to plus 127 byte range of the instruction permits.
17) There isn't any.
18) **Undefined Label**
    The identified label is not defined anywhere within the assembler source file.
19) **Forward Reference To Page Zero Memory**
    This warning message is generated when an instruction that has both page zero and absolute addressing modes has an operand that is defined later in the assembly source file to be a page zero address. During pass one of the assembly, two bytes are allocated for the operand since its value is not yet known. Then during pass two, the operand is found to require only a single byte so one byte is wasted. This is usually not a serious error because the generated code will generally execute as expected.
20) **Immediate Operand Greater Than 255**
    An immediate operand expression evaluated to greater than 255, the maximum value that can be represented in a single byte immediate operand.
21) There isn't any.
22) There isn't any.
23) There isn't any.
24) There isn't any.
25) **Label (Symbol) Table Overflow**
    The size of the workspace is insufficient to hold the current source file and a table for all of the labels encountered in the program. To assemble will require a reduction in either the size of the program source file or the number of symbols or an increase in the size of the workspace.

APPENDIX B

6502 INSTRUCTION ADDRESSING MODES

	ASSEMBLER ADDRESSING MODES	AC	IM	DIRECT			INDEXED				INDIRECT		
	MACHINE LANGUAGE ADDRESSING MODES	AC	IM	ZP	Abs	Rel	ZP X	Abs X	ZP Y	Abs Y	In X	In Y	In
GENERAL	ADC AND CMP EOR LDA ORA SBC		X	X	X		X	X		X	X	X	
SHIFT	ASL LSR ROL ROR	X		X	X		X	X					
BIT TEST	BIT			X	X								
COMPARE INDEX	CPX CPY		X	X	X								
DECREMENT/ INCREMENT	DEC INC			X	X		X	X					
JUMP	JMP				X								X
	JSR				X								
LOAD INDEX	LDX		X	X	X				X	X			
	LDY		X	X	X		X	X					
STORE INDEX	STX			X	X				X				
	STY			X	X		X						
STORE	STA			X	X		X	X		X	X	X	
BRANCH	BCC BCS BEQ BMI BNE BPL BVC BVS					X							
IMPLIED	BRK NOP RTI RTS CLC CLD CLI CLV DEX DEY INX INY PHA PHP PLA PLP SEC SED SEI TAX TAY TSX TXA TYA TXS	Implied (No Operand)											

AC—Accumulator
IM—Immediate
ZP—Zero Page

Abs—Absolute
Rel—Relative
In—Indirect

APPENDIX C

ASSEMBLER/EDITOR STATISTICS

Source File Storage Requirements (per line):

Two bytes for the line number plus,
one byte for each text character plus,
one byte for the line terminator character (ØD).

All repeated characters such as a sequence of spaces occupy only two bytes; one for the character and one for a repeat count.

Symbol Table Storage Requirements:

Six bytes/symbol.
65Ø2 opcodes and reserved names occupy no symbol table space.

Assembly Speed:

Approximately 600 lines per minute.

APPENDIX D

OS-65D V3.N VERSION OF THE 6502 ASSEMBLER/EDITOR

In OS-65D V3.N, the Assembler/Editor is loaded from disk and initiated by typing ASM after the A* prompter in the DOS kernel command mode. Whenever exiting to the DOS, you can return to the Assembler/Editor as long as it is loaded by typing RETURN ASM (or RE ASM).

This version of the 65Ø2 Assembler/Editor contains the following commands in addition to those described elsewhere in this manual.

Exit	exits the Assembler/Editor and transfers control to the OS-65D kernel which then displays the A* prompter.
Hnnnn	sets the high memory limit to hexadecimal address nnnn.
Mnnnn	sets the memory offset for A3 assemblies to hexadecimal nnnn.
Control-I	tabs 8 spaces from the current print position. Also: CONTROL-U 7 spaces CONTROL-Y 6 spaces CONTROL-T 5 spaces CONTROL-R 4 spaces CONTROL-E 3 spaces
Control-C	aborts the current operation.
!Command Line	sends the command line to OS-65D to be executed, then returns to the Assembler.

This version of the Assembler/Editor occupies memory from Ø2ØØ through 16FF. Its workspace starts at 3179 (3279 in mini-floppy versions) and is utilized as shown below:

3179,317A	address of start of source (low, high)—normally 317E
317B,317C	address of end of source +1 (low, high)
317D	number of tracks required for source
317E	normal start of source

Note: It is possible to carry the Assembler's symbol table forward from one assembly to another. To do so, exit the Assembler after the first assembly and enter the machine language monitor by typing "RE M". Change location Ø855 from ØA to 18 and read out the contents of locations 2F83 and 2F84. Enter the values from these locations into locations 12FA and 12FB, respectively. Then re-enter the Assembler by re-entering the DOS kernel at 2547 and typing "RE AS." Now the symbols from the first assembly will remain in the symbol table for reference during the next assembly. Likewise, the symbols from the first and second assemblies will remain for the third assembly, etc. If you want to eliminate all but the symbols from the first assembly, exit the Assembler and immediately re-enter it by tying "RE AS." To restore normal operation of the Assembler, change location Ø855 back to ØA. This will cause the symbol table to be cleared at the beginning of each assembly.

APPENDIX E

CASSETTE VERSION OF ASSEMBLER/EDITOR

This version of the Assembler/Editor is supplied on an auto-load cassette tape. The following procedure may be used to load the Assembler from tape:

LOADING THE ASSEMBLER/EDITOR

1) Apply power to your personal computer then reset it by depressing the <BREAK> key. Load the cassette, label up, into the cassette machine and turn the cassette machine on with the volume at about mid-range.
2) Type "ML".
   The M initiates the 65VP monitor and the L starts the auto-load. In a few seconds the four zeros in the upper left portion of the video monitor should change to an incrementing address value with a rapidly changing data field. The value of the address is dependent on which auto-load cassette is being read. At this time, a checksum loader is being read into memory in 65VP format. Upon completion (no more than 30 seconds), the checksum loader will load the rest of the cassette. The Assembler comes up with the message INIZ? (Y/N).
   Should a checksum error occur, the following message is printed:
   OBJECT LOAD CHECKSUM ERR
   REWIND PAST ERR—TYPE G TO RESTART
   If a checksum error consistently happens at the same location, the cassette is probably bad. Contact your OSI dealer concerning replacement. However, should checksum errors occur randomly, at various locations, it is most likely that there is a problem with the cassette machine or the connection to the computer. Check for broken or frayed connections. Make sure the playback head and pressure roller/capstan assembly is clean. With a minimal amount of care, no problems with auto-load cassettes should be encountered.

The cassette version of the Assembler/Editor permits loading and saving source codes in a manner similar to ROM BASIC.

TO SAVE SOURCE CODE

Type "SAVE" <CR> (*carriage return*), type "PRINT" <line specification>, turn on the cassette machine in record mode and hit <CR>. As in ROM BASIC, the SAVE mode is disabled by typing "LOAD" <CR> followed by a space.

TO LOAD PREVIOUSLY RECORDED SOURCE CODE

Turn on cassette machine in play, type "LOAD", wait for leader to pass, then hit <CR>. The LOAD mode is disabled by hitting a space.

This version of the Assembler/Editor also provides the following commands:

EXIT—causes the computer to execute its reset vector and display "C/W/M?". Great care must be taken never to type "C", as this will destroy the Assembler/Editor. The Assembler/Editor may then be re-entered by typing "M 1300 G".

CONTROL-I—tabs 8 spaces from the current cursor location.
The above commands, as all other Assembler/Editor commands, may be executed by typing the first letter only.

This version of the Assembler/Editor occupies memory from 0240 through 1390 (hexadecimal) and requires a minimal total of 8K of memory to operate. Its source file workspace starts at 1391 and ends at 1FFF, as supplied. The entry location is hex 1300. While running, all of page zero is used. However, you can exit the Assembler/Editor-use page zero and re-enter it by typing "M 1300 G".

The following locations may be changed in the cassette version of the Assembler/Editor to suit your requirements:

12C9,12CA—the low, high memory address of the start of the source file workspace. 1391 hex, as supplied.

12CB,12CC—the low, high memory address of the end of the source file workspace. 1FFF, as supplied.

12FC,12FD—the low, high memory offset used to bias placement of object code during an A3 assembly. 0, as supplied.

12FE,12FF—the low, high address of the next available source file storage location. These locations are initialized to the address of the start of the workspace by the INIZ command and, thereafter, are automatically updated by the Editor.

It is possible to carry the Assembler's symbol table forward from one assembly to another. To do so, exit the Assembler after the first assembly and enter the machine language monitor by "M". Change location 0855 from 0A to 18 and read out the contents of locations 000A and 000B. Enter the values from those locations into locations 12FA and 12FB, respectively. Then re-enter the Assembler by typing ".1300G". Now the symbols from the first assembly will remain in the symbol table for reference during the next assembly. Likewise, the symbols from the first and second assemblies will remain for the third assembly, etc. If you want to eliminate all but the symbols from the first assembly, exit the Assembler and immediately re-enter it by typing "M1300G". To restore normal operation of the Assembler, change location 0855 back to 0A. This will cause the symbol table to be cleared at the beginning of each assembly.

APPENDIX F

EXTENDED MONITOR COMMAND SUMMARY

COMMAND	FUNCTION	SUBCOMMANDS (<CR> ALWAYS RETURNS TO ":")
@aaaa	display contents of aaaa	dd—change aaaa (dd = two hex digits) "—display as character <LF>—display next location ↑—display previous location /—display same location
A, X, Y, P or K	display A, X, Y, P or K from last break A, X, Y—processor register P—processor status K—stackpointer	dd—change register "—display as character
Bn,aaaa	enter breakpoint n at aaaa	(n = 1-8)
C	continue from last breakpoint	
Dffff,tttt	dump ffff through tttt-1	
En	eliminate breakpoint n	
Fffff,tttt = dd	fill ffff through tttt-1 with dd	
Gaaaa	go to aaaa	
Hxxxx,yyyy +	display xxxx + yyyy	(also −, *, /)
I	display location of last breakpoint	
L	load memory from cassette	SPACE key returns to ":"
Maaaa = ffff,tttt	move ffff through tttt-1 to aaaa	
Ndd...dd>ffff,tttt	search ffff through tttt-1 for dd...dd	(dd...dd is 1-8 bytes)
O	display overflow/remainder from last H command	
Qaaaa	disassemble from aaaa	<LF> continue disassembly
Raaaa = ffff,tttt	relocate ffff through tttt-1 to aaaa	
Sffff,tttt	save ffff through tttt-1 to cassette	
T	display breakpoint table	
V	view from cassette	SPACE key returns to ":"
WC...c>ffff,tttt	search ffff through tttt-1 for c...c	(c...c is 1-8 characters)

APPENDIX G

OS-65D V3.N VERSION OF THE EXTENDED MONITOR

In OS-65D V3.N, the Extended Monitor is loaded from disk and initiated by typing EM after the A* prompter in the DOS kernel command mode. Whenever exiting to the DOS, you can return to the Extended Monitor as long as it is loaded by typing RETURN EM.

This version of the Extended Monitor occupies memory from 17ØØ through 1FFF and uses page zero locations CØ through FF.

APPENDIX H

CASSETTE VERSION OF EM

This version of the Extended Monitor is supplied on an auto-load cassette tape. The following procedure may be used to load the Extended Monitor from tape:

LOADING THE EXTENDED MONITOR

1) Apply power to your personal computer then reset it by depressing the <BREAK> key. Load the cassette, label up, into the cassette machine and turn the cassette machine on with the volume at about mid-range.

2) Type "ML.

3) The M initiates the 65VP monitor and the L starts the auto-load. In a few seconds the four zeros in the upper left portion of the video monitor should change to an incrementing address value with a rapidly changing data field. The value of the address is dependent on which auto-load cassette is being reared. At this time, a checksum loader is being read into memory in 65VP format. Upon completion (no more than 30 seconds), the checksum loader will load the rest of the cassette. The Extended Monitor comes up with the prompter":". Should a checksum error occur, the following message is printed:

    OBJECT LOAD CHECKSUM ERR
    REWIND PAST ERR—TYPE G TO
    RESTART

   If a checksum error consistently happens at the same location, the cassette is probably bad. Contact your OSI dealer concerning replacement. However, should checksum errors occur randomly, at various locations, it is most likely that there is a problem with the cassette machine or the connection to the computer. Check for broken or frayed connections. Make sure the playback head and pressure roller/capstan assembly is clean. With a minimal amount of care, no problems with auto-load cassettes should be encountered.

This version of the Extended Monitor contains one additional instruction for dumping the contents of memory on the 24 character 1P video screen:

COMMAND	FUNCTION
Zaaaa	dumps 8 bytes from aaaa

SUBCOMMAND

<LF> dumps next 8 bytes

This version occupies memory from 0800 through 0FFF and uses page zero locations D0 through FF. The checksum loader used to load the Extended Monitor occupies locations 0700 through 07EF.

This version of the Extended Monitor is normally entered at 0800. This causes the stackpointer to be set to 01FF and the breakpoint registers to be initialized. Under certain circumstances, it may be desirable to re-enter the Extended Monitor without this initialization. This may be done by entering it at 081F.

There are two free command letters—J and U, that can be utilized by inserting the address of a command routine at 0974 for J or 098A for U. The machine language command routine must end with an RTS instruction.

APPENDIX I

ROM MONITOR COMMANDS

In the cassette version, the ROM Monitor is entered by typing <BREAK> and then M. If BASIC, the Assembler/Editor, or the Extended Monitor is in memory when <BREAK> is hit, then the user may return to it by typing <BREAK> and then W.

On disk systems, the user can also enter the ROM Monitor by typing <BREAK> and then M, but, if this is done, then re-entry to BASIC, the Assembler/Editor, or the Extended Monitor is usually impossible. However, the disk based user may also enter the ROM Monitor by typing "EXIT" and then "RE M". The DOS kernel may then be re-entered by typing .2547G <CR>. The ROM Monitor begins at address $FE00.

The ROM Monitor has four command modes:

1) Addressing Mode
When an address is typed, the address and the contents of that address are displayed. If the Monitor is not in the Addressing Mode then it may be entered by typing a period (.).

2) Data Entry Mode
Hexadecimal data may be put into the memory location whose address is displayed. This mode is entered by typing /. When in this mode, a <CR> will increase the displayed address by one.

3) Go Mode
If the Monitor is in the Addressing Mode, then typing a G will cause the Monitor to execute a JMP to the address currently displayed.

4) Cassette Loader Mode
If in the Addressing Mode, typing L permits the loading of programs from cassette. Upon typing L, all ASCII commands are accepted from the audio cassette input port rather than the keyboard.

Some addresses associated with the monitor ROM are

FE00—Start of Monitor (restart location)
FE0C—Restart with clear screen and no other initialization
FE43—Entry to Addressing mode
FE77—Entry to Data mode

A complete listing of the monitor ROM may be found in the Appendix of the OSI 65V Primer.

APPENDIX J

ASCII CHARACTER CODES

CODE	CHAR	CODE	CHAR	CODE	CHAR
ØØ	NUL	2B	+	56	V
Ø1	SOH	2C	,	57	W
Ø2	STX	2D	—	58	X
Ø3	ETX	2E	.	59	Y
Ø4	EOT	2F	/	5A	Z
Ø5	ENQ	3Ø	Ø	5B	[
Ø6	ACK	31	1	5C	/
Ø7	BEL	32	2	5D	]
Ø8	BS	33	3	5E	^
Ø9	HT	34	4	5F	—
ØA	LF	35	5	6Ø	(Blank)
ØB	VT	36	6	61	a
ØC	FF	37	7	62	b
ØD	CR	38	8	63	c
ØE	SO	39	9	64	d
ØF	SI	3A	:	65	e
1Ø	DLE	3B	;	66	f
11	DC1	3C	<	67	g
12	DC2	3D	=	68	h
13	DC3	3E	>	69	i
14	DC4	3F	?	6A	j
15	NAK	40	@	6B	k
16	SYN	41	A	6C	l
17	ETB	42	B	6D	m
18	CAN	43	C	6E	n
19	EM	44	D	6F	o
1A	SUB	45	E	7Ø	p
1B	ESC	46	F	71	q
1C	FS	47	G	72	r
1D	GS	48	H	73	s
1E	RS	49	I	74	t
1F	US	4A	J	75	u
2Ø	SP	4B	K	76	v
21	!	4C	L	77	w
22	″	4D	M	78	x
23	#	4E	N	79	y
24	$	4F	Ø	7A	z
25	%	5Ø	P	7B	{
26	&	51	Q	7C	}
27	′	52	R	7D	¦
28	(	53	S	7E	÷
29	)	54	T	7F	DEL
2A	*	55	U		

APPENDIX K

CHECKSUM FORMAT

The checksum format is as follows for each "record" of data:
;18aaaadd...ddcccc

where:

;	is the start of record character
18	is the hexadecimal number of data bytes in the record (24 decimal)
aaaa	is the address of the first data byte in the record
dd...dd	are the 24 data bytes
cccc	is the checksum—a sum modulo 65536 of all bytes in the record after 10 the start of record character

APPENDIX L

OS-65D DISK OPERATING SYSTEM

COMMANDS	
ASM	Load the assembler and extended monitor. Transfer control to the assembler.
BASIC	Load BASIC and transfer control to it.
CALL NNNN=TT,S	Load contents of track "TT", sector "S," to memory location "NNNN".
D9	Disable error 9. This is required to read some earlier version files (V1.5, V2.0).
DIR NN	Print sector map directory of track "NN".
EM	Load the assembler and extended monitor. Transfer control to the extended monitor.
EXAM NNNN=TT	Examine track TT. Load entire track contents, including formatting formation, into location "NNNN".
GO NNNN	Transfer control <GO> to location "NNNN"
HOME	Reset track count to zero and home the current drive's head to track zero.
INIT	Initialize the entire disk, i.e., erase the entire diskette (except track Ø) and write new formatting information on each track.
INIT TT	Same as "INIT", but only operates on track "TT".
IO NN, MM	Changes the input I/O distributor flag to "NN", and the output flag to "MM". See the table at the end of this appendix for I/O flag settings.
IO, MM	Changes only the output flag.
IO NN	Changes only the input flag.
LOAD FILNAM	Loads named source file, "FILNAM" into memory.
LOAD TT	Loads source file into memory given starting track number "TT".
MEM NNNN,MMMM	Sets the memory I/O device input pointer to "NNNN", and the output pointer to "MMMM".
PUT FILNAM	Saves source file in memory on the named disk file "FILNAM".
PUT TT	Saves source file in memory on track "TT", and following tracks.
RET ASM	Restart the assembler.
RET BAS	Restart BASIC.
RET EM	Restart the Extended Monitor.
RET MON	Restart the Prom Monitor (via RST vector).
SAVE TT,S=NNNN/P	Save memory from location "NNNN" on track "TT" sector "S" for "P" pages.
SELECT X	Select disk drive, "X" where "X" can be, A, B, C, or D. Select enables the requested drive and homes the head to track Ø.
XQT FILNAM	Load the file, "FILNAM" as if it was a source file, and transfer control to location $327E.

NOTE:

—Only the first 2 characters are used in recognizing a command. The rest up to the blank are ignored.
—The line input buffer can only hold 18 characters including the return.
—The DOS can be reentered at 9543 ($2547).
—File names must start with an "A" to "Z" and can be only 6 characters long.
—The dictionary is always maintained on disk. This permits the interchange of diskettes.
—The following control keys are valid:
CONTROL—Q continue output from a CONTROL-S
CONTROL—S stop output to the console

CONTROL—U	delete entire line as input
BACKARROW	delete the last character typed.
SHIFT—O	delete the last character (polled keyboards)

ERROR NUMBERS

1—Can't read sector (parity error).
2—Can't write sector (reread error).
3—Track zero is write protected against that operation.
4—Diskette is write protected.
5—Seek error (track header doesn't match track).
6—Drive not ready.
7—Syntax error in command line.
8—Bad track number.
9—Can't find track header within one rev of diskette.
A—Can't find sector before one requested.
B—Bad sector length value.
C—Can't find that name in directory.
D—Read/Write attempted past end of named file!

MEMORY ALLOCATION

5″ Floppies		8″ Floppies
0000–22FF	BASIC or Assembler/Extended Monitor	0000–22FF
2200–22FE	Cold start initialization on boot	2200–22FE
2300–265B	Input/Output handlers	2300–265B
265C–2A4A	Floppy disk drivers	265C–2A4A
2A4B–2E78	OS-65D V3.2 operating system kernel	2A4B–2E78
2E79–2F78	Directory buffer	2E79–2F78
2F79–3178	Page 0/1 swap buffer	2F79–3178
3179–3278	DOS extensions	– – –
3279–327D	Source file header	3179–317D
327E–	Source file	317E

DISKETTE ALLOCATION

5″ Floppies		8″ Floppies
0–1	OS-65D V3.2 (boot-strap-loads to $2200 for 8 pages).	0–1
2–6	9½ Digit Microsoft BASIC	2–4
7–9	Assembler-Editor (if present)	5–6
10–11	Extended Monitor (if present)	7–7
12	Sector 1—Directory, page 1.	8
	Sector 2—Directory, page 2.	
	Sector 3—BASIC overlays.	
	Sector 4—GET/PUT overlays	
13	Track0/Copier utility (loads to $0200 for 5 pages).	1 Sector 2
14–38	User programs and OS-65D utility BASIC programs.	9–75
39	Compare routine, on some disks only.	76

I/O FLAG BIT SETTINGS

INPUT:

Bit 0—ACIA on CPU board (terminal).
BIT 1—Keyboard on 540 board.
BIT 2—UART on 430/550 board.
BIT 3—NULL.
BIT 4—Memory input (auto incrementing).
BIT 5—Memory buffered disk input.
BIT 6—Memory buffered disk input.

BIT 7—55Ø board ACIA input. As selected by index at location $2323 (8995 decimal).

OUTPUT:

BIT Ø—ACIA on CPU board (terminal).
BIT 1—Video output on 54Ø board.
BIT 2—UART on 43Ø/55Ø board.
BIT 3—Line printer interface.
BIT 4—Memory output (auto incrementing).
BIT 5—Memory buffered disk output.
BIT 6—Memory buffered disk output.
BIT 7—55Ø board ACIA output. As selected by index.

NOTE: In the ASM $12EØ contains the number of lines per page and is set to top of page after each RE ASM.

APPENDIX M

THE FLOATING POINT ACCUMULATOR

The floating accumulator (FAC) on disk based systems is located in six bytes on page zero at addresses $AE-$B3. See Note 2 for BASIC-in-ROM. The FAC is used during all operations involving numeric variables. Of interest to end users is the fact that when a BASIC statement like

Y=USR(formula)

is executed, the value of the formula is loaded into the FAC before BASIC branches to the user's routine. The floating point format is

$\pm .m \times 2^e$

1) e is the exponent. The byte with address $AE contains e+$80.

2) is the mantissa. The binary point is assumed to be on the left of m. m is always normalized, that is, m is left shifted and e is decremented until the leftmost bit of m is 1. Thus, for example, .125 is stored as $.1 \times 2^{-10}$ (binary) instead of $.001 \times 2^{0}$. The mantissa is a 32 bit number and is put into the FAC at $AF, %B0, $B1, $B2.

3) The sign of the floating point number is put into the sign bit (leftmost bit) of the byte with address $B3. This bit is 0 for a positive number and 1 for a negative number. The remaining bits are indeterminate and have no meaning.

Examples:

Number (decimal)	floating point (binary)	In FAC (hex)
26.5	$.110101 \times 2^{101}$	85D400000000
−26.5	$-.110101 \times 2^{101}$	85D400000080
.25	$.1 \times 2^{-1}$	7F8000000000
.2	$.1100 \times 2^{-10}$	7ECCCCCCCD00

Note 1: When .2 is converted from decimal to binary, it becomes an infinite repeating number. The bar over the mantissa indicates that those four bits repeat forever. Thus, the mantissa is
.11001100110011001100110011001100---
when this is rounded to 32 bits it becomes
.11001100110011001100110011001101

Note 2: For BASIC-in-ROM, the FAC is five bytes at addresses $AC-$B0. the exponent (+$80) is in the first byte, the sign is the sign bit of the last byte and the mantissa is the middle three bytes.

APPENDIX N

OPCODE TABLE

MSD \ LSD	Ø	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Ø	BRK	ORA-IND,X				ORA-Z,Page	ASL-Z,Page		PHP	ORA-IMM	ASL-A			ORA-ABS	ASL-ABS	
1	BPL	ORA-IND,Y				ORA-Z,Page,X	ASL-Z,Page,X		CLC	ORA-ABS,Y				ORA-ABS,X	ASL-ABS,X	
2	JSR	AND-IND,X			BIT-Z,Page	AND-Z,Page	ROL-Z,Page		PLP	AND-IMM	ROL-A		BIT-ABS	AND-ABS	ROL-ABS	
3	BMI	AND-IND,Y				AND-Z,Page,X	ROL-Z,Page,X		SEC	AND-ABS,Y				AND-ABS,X	ROL-ABS,X	
4	RTI	EOR-IND,X				EOR-Z,Page	LSR-Z,Page		PHA	EOR-IMM	LSR-A		JMP-ABS	EOR-ABS	LSR-ABS	
5	BVC	EOR-IND,Y				EOR-Z,Page,X	LSR-Z,Page,X		CLI	EOR-ABS,Y				EOR-ABS,X	LSR-ABS,X	
6	RTS	ADC-IND,X				ADC-Z,Page	ROR-Z,Page		PLA	ADC-IMM	ROR-A		JMP-IND	ADC-ABS	ROR-ABS	
7	BVS	ADC-IND,Y				ADC-Z,Page,X	ROR-Z,Page,X		SEI	ADC-ABS,Y				ADC-ABS,X	ROR-ABS,X	
8		STA-IND,X			STY-Z,Page	STA-Z,Page	STX-Z,Page		DEY		TXA		STY-ABS	STA-ABS	STX-ABS	
9	BCC	STA-IND,Y			STY-Z,Page,X	STA-Z,Page,X	STX-Z,Page,Y		TYA	STA-ABS,Y	TXS			STA-ABS,X		
A	LDY-IMM	LDA-IND,X	LDX-IMM		LDY-Z,Page	LDA-Z,Page	LDX-Z,Page		TAY	LDA-IMM	TAX		LDY-ABS	LDA-ABS	LDX-ABS	
B	BCS	LDA-IND,Y			LDY-Z,Page,X	LDA-Z,Page,X	LDX-Z,Page,Y		CLV	LDA-ABS,Y	TSX		LDY-ABS,X	LDA-ABS,X	LDX-ABS,Y	
C	CPY-IMM	CMP-IND,X			CPY-Z,Page	CMP-Z,Page	DEC-Z,Page		INY	CMP-IMM	DEX		CPY-ABS	CMP-ABS	DEC-ABS	
D	BNE	CMP-IND,Y				CMP-Z,Page,X	DEC-Z,Page,X		CLD	CMP-ABS,Y				CMP-ABS,X	DEC-ABS,X	
E	CPX-IMM	SBC-IND,X			CPX-Z,Page	SBC-Z,Page	INC-Z,Page		INX	SBC-IMM	NOP		CPX-ABS	SBC-ABS	INC-ABS	
F	BEQ	SBC-IND,Y				SBC-Z,Page,X	INC-Z,Page,X		SED	SBC-ABS,Y				SBC-ABS,X	INC-ABS,X	

LSD—Least Significant Digit
MSD—Most Significant Digit

APPENDIX O

6502 REFERENCE LIST

1.* How to Program Microcomputers
by William Barden
Howard W. Sams & Company, Inc.
Indianapolis, IN 46268

2. 6502 Software Gourmet Guide and Cookbook
by Robert Findley
SCELBI Publications
20 Hurlbut Street
Elmwood, CT 06110

3. The First Book of KIM

4. Programming a Microcomputer: 65Ø2
by Caxton C. Foster
Addison Wesley Publishing Company, Inc.
Reading, MA 01867

5. 65Ø2 Assembly Language Programming
by Lance Leventhal
Osborne/McGraw-Hill

6. MCS65ØØ Microcomputer Family Programming Manual
MOS Technology, Inc.
950 Rittenhouse Road
Norristown, PA 19401

7. MICRO: The 65Ø2 Journal
P.O. Box 65Ø2
Chelmsford, MA 01824

8. SY65ØØ/MCS65ØØ Microcomputer Family Hardware Manual
Synertek
3Ø5Ø Coronado Drive
Santa Clara, CA 95Ø51

9. Programming the 65Ø2 (Second Edition)
by Rodney Zaks
Sybex
2344 Sixth Street
Berkeley, CA 94710

10. 65Ø2 Applications Book
by Rodney Zaks
Sybex
2344 Sixth Street
Berkeley, CA 9471Ø

11. 65Ø2 Games
by Rodney Zaks
Sybex
2344 Sixth Street
Berkeley, CA 94710

12. Programming & Interfacing The 65Ø2, With Experiments
by Marvin L. De Jong
Howard W. Sams & Co., Inc.
4300 West 62nd Street
Indianapolis, IN 46268

13.* 65V Primer: The Workbook of Programming exercises in machine code, using your machine's built-in 65V monitor in ROM.
Ohio Scientific
1333 S. Chillicothe Rd.
Aurora, OH 44202

* Available from OSI

INDEX

A

B

C

D

E

F

I

L

M

O

P

R

S

T

U

W