A/65

Pegasus Software
P.O. Box 10014
Honolulu, Hawaii 96816
Phone (808) 735-5013

Version 1.0

CHAPTER I

A/65 INTRODUCTION

1.1 GENERAL

A/65 is a powerful, two pass assembler for OHIO SCIENTIFIC COMPUTERS using the 8" disk system. It is designed to provide the OSI user with a professional quality assembler to allow the OSI system to be used as a "serious" development system. A/65 is designed to assemble files from disk and produce formatted output listings. Object code may be assembled directly to memory (Be careful to avoid A/65 and the symbol table and Never assemble code to page zero or page one) or directed to the disk.

1.2 USE OF A/65

A/65 is approximately 6k long and is loaded into RAM memory from the disk. After bringing up the disk, exit to DOS and type "A/65". The assembler will load and begin execution. The user must then answer several questions before assembly may begin. The first is the location of the symbol table. The assember will print:

```
PEGASUS SOFTWARE A/65 Version 1.0
Copyright 1980 by Compas
Symbol Table Access

From=          To=
```

"FROM" is the beginning of the symbol table and "TO" is the end of the symbol table. Since each symbol requires 8 bytes, the ending address must be modulo 8. Enter the numbers in the normal hexadecimal fashion. Carriage return will default to all available memory. The next question asks for the source file name:

```
File=
```

Next the system asks if you want a full output listing (Y) or an errors only listing (N):

```
Listing?
```

The next question asks if you want object code output:

```
Object Code?
```

If yes then whether to disk or memory:

```
Memory or Disk (M/D)?
```

If "M", it asks for offset (default is Ø):

```
Offset?
```

If "D", it asks for disk file:

```
File=
```

The next question asks if you want a symbol table printed:

```
Print Symbol Table?
```

If you said yes to either listing or symbol table, you're asked how many lines per page (in hexadecimal). Default is 3B which is for 11" paper:

```
# Lines?
```

It will then ask for output device # (default is console). Refer to OS65-D3 Operating System Manual. These are not always equivalent to devices as used with Basic.

```
Output Device No.?
```

At this point, you're reminded to set the top of form, so that the listing will be positioned properly on paper.

Several examples:

(1)
```
Pegasus Software  A/65 Version 1.0
Copyright (c) 1980 By COMPAS

Symbol Table Address
From=4580                   To=BFFF
File=XAMPLE
Listing? Y
Object Code? Y   To: Memory or Disk (M/D)? D   File=XOBJCT
Print Symbol Table? Y      # Lines? 3B
Output device no.  8
Set top of form
```

(2)
```
Pegasus Software A/65 Version 1.0
Copyright (c) 1980 by COMPAS

Symbol Table Address
From=4580                   To=BFFF
File=TEST
Listing? Y
Object Code? Y   To: Memory or Disk (M/D)? M   Offset: 0000
Print Symbol Table? Y      # Lines? 3B
Output device no.  8
Set top of form
```

1.3 OBJECT CODE LOADER

Included with the A/65 system are two object code file loaders. These facilitate the loading of the disk based object files produced by A/65 into memory.

The loader is invoked by using the DOS (disk operating system) XQ command followed by the name of the desired loader. Two different loaders are provided so that the user can load a file into memory either above or below the DOS without conflicting with the loader itself. The "LOADER" program uses memory from $317E to $3E00, and should not be used to load object code into this area. Similarly the "LOADR2" program uses memory from $1500 to $2200, and will not load object code into this area properly.

The loader reports the starting address of the load sequence for verification. This address is not necessarily the "go" address or the lowest byte of memory used by the loaded program. It is simply the first address in the object file. Subsequent addresses may be either higher or lower than the one reported by the loader, dependent upon the original assembler source file.

CHAPTER 2

USE OF THE A/65 ASSEMBLER

2.1 OVERVIEW

The process of translating a mnemonic or symbolic form of a computer program (source code) to actual processor instructions (object code) is called an assembly, and a program which performs the translation is called an assembler. The symbols used and rules of association for these symbols are the assembly language. One assembly language statement will translate into one processor instruction.

A/65 features many powerful capabilities. The user has complete control over where the symbol table is located in memory, how much of the listing is generated, how many lines per page are generated, where the listing is printed, and whether the complete text of ASCII strings are printed.

2.2 SETTING UP THE SYMBOL TABLE

As a guideline for allocating space for the symbol table, allow eight bytes of memory space for each uniquely defined symbol in the source code. Since each symbol in the symbol table requires eight bytes of memory, the end of the symbol table must always be a multiple of eight bytes from the start of the symbol table. If the allocated symbol table area is not large enough, the assembly will be terminated and the message:

SYMBOL TABLE OVERFLOW

will be printed.

2.3 INVOKING A/65

A/65 is invoked by exiting to operating system and typing A/65. The assembler automatically loads and executes.

2.4 A/65 EXPRESSIONS

Assembler expressions are very useful tools to facilitate programming and to generate both readable and easily changeable code.

There are two components of Assembler expressions: elements and operators.

2.4.1 ELEMENTS

Elements may be classified into three distinct types: constants, symbols, and the location counter.

CONSTANTS

Numeric constants may be written in several bases. The base is determined by the type of prefix character preceding the digits. The relationship between prefix character and base is defined in the following table:

PREFIX CHARACTER	BASE
(none)	10 (Decimal)
$	16 (Hexadecimal)
@	8 (Octal)
%	2 (Binary)

Some example are:

```
0005  1000  10                          .BYT  $10,10,@10,%10
0005  1001  0A
0005  1002  08
0005  1003  02
0006  1004  AD  10  F7                   LDA  $F710
0007  1007  A5  1D                       LDA  29
0008  1009  A5  7E                       LDA  @176
0009  100B  A5  6D                       LDA  %01101101
```

ASCII literal constants are enclosed in quotes, and are used to insert the ASCII representation of character strings into memory.

For example:

```
0011  100D  25                          .BYT  '%','I''M',''''
0011  100E  49  27  4D
0011  1011  27
0012  1012  A9  50                       LDA  #'P
0013  1014  A9  27                       LDA  #''
0014  1016  A9  35                       LDA  #'5
```

Note that two quotes are needed to represent the insertion of a quote in memory. Thus, in the last field of the .BYT directive, the first represents a single quote, and the last closes off the string.

SYMBOLS

Symbols are names used to represent numerical values. They may be from one to six alphanumeric characters long, but the first character must be alphabetic. In addition, the 56 valid opcodes (listed in Table 2-1) and the reserved symbols A,X,Y,X, and P have special meaning to the Assembler, and may not be used as symbols.

For example:

```
0016  1018                                VARBLE =$2C
0017  1018  AB                            DATA1  .BYT  $AB,VARBLE
0017  1019  2C
0018  101A  AD  18  10                    LAB190 LDA  DATA1
0019  101D  A5  2C                               LDA  VARBLE
```

LOCATION COUNTER

The location counter, referenced by the character "*", is a sequential counter used by A/65 to keep track of its current position in memory, and may be freely used in expressions by the programmer.

For example:

```
0021  101F  10  1F                        .DBY *
0022  1021  AD  21  10                    LDA *
```

2.4.2 OPERATORS

Four arithmetic operators are provided in A/65:

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division

Evaluation of expressions proceeds strictly from left [illegible], with no parenthetical grouping allowed; all operators have equal precedence.

In addition, there are two special operators:

Character	Operation
>	High-Byte-Selection
<	Low-Byte Selection

Operators > and < truncate a two-byte value to its high or low byte, respectively.

For example:

```
0024  1024  AB                HIGH    .BYT  >$ABCD,<*,51<HIGH-%10
0024  1025  25
0024  1026  27
0025  1027  A5  41                    LDA  <*+>$1AC2
0026  1029  A5  1A                    LDA  %101+7+$7+@7
0027  102B  A5  11                    LDA  >HIGH-$401<65
```

Expressions which evaluate to negative values are illegal. The twos complement representation of a negative number must be expressed as an unsigned (preferably hexadecimal) constant (e.g., write "-1" as "$FF").

Note especially that expressions are evaluated at assembly time, not at execution time.

2.5 A/65 SOURCE STATEMENTS

Assembler source statements are comprised of four possible fields:

```
[label]    [opcode   [operand] ]    [;comment]
```

Brackets surrounding a field indicate that it is optional. Thus, although none of the fields is mandatory, an opcode field must precede an operand field. Input to A/65 is free form; any field may start in any column.

In particular, note that due to the reserved opcodes, the user is able to precede labels with spaces. If no label is present, an opcode may be placed in the first column.

Each field in the program need only be separated by a single space. If the fields are separated in this manner, A/65 will columnize the fields and produce a readable listing. The user's program may then be stored on the disk in a highly compressed form.

Also note that the comment field should be preceded by a semicolon. If the semicolon is omitted, the comment field will not be placed in its proper column in the listing.

2.5.1 LABELS

A label is a symbol which, to be defined, must appear as the first field on a line, although it may begin in any column. Using a symbol as a label is a way to assign the current value of the location counter to the symbol before the rest of the line is processed by the Assembler. Labels are used with instructions as branch targets and with memory data cells for reference in operands.

A line containing only a label is valid, so several labels may be assigned to the same memory location by putting each on a separate line:

```
0029  102D                              SAME1
0030  102D                              SAME2
0031  102D  AD  2D  10                  SAME3  LDA  SAME1
```

2.5.2 OPCODES AND OPERANDS

There are two distinct classes of assembler instructions available to the programmer: machine instructions and assembler directives.

MACHINE INSTRUCTIONS

The 56 valid machine instruction mnemonics (listed and defined in Table 2-1) represent the operations implemented on the 6500 family of microprocessors. When assembled, each mnemonic generates one byte of machine code, the actual bit pattern depending upon both the operation specified in the opcode field and the addressing mode determined from the operand field. The operand field may generate one or two bytes of address.

Operand Addressing Modes

ABSOLUTE ADDRESSING. The absolute addressing mode is the most common in concept; the data following the machine code is treated as the address of a memory location containing the actual data to be processed during the instruction step. This address is stored in reverse order-as low-byte, then high-byte-to increase processing efficiency during execution time.

For example:

```
0066  1050
0067  1050
0068  1050
0069  1050
0070  1050
0071  1050  2C  1F  4C
0072  1053  CD  D7  C2
0073  1056  CE  BA  54
0074  1059  4D  B0  54
0075  105C  4C  00  A0
0076  105F  20  00  C8
0077  1062  AD  5F  D8
0078  1065  6E  CF  11
0079  1068  ED  CF  54
0080  106B  8D  E2  4D
```

```
PIA      =$4C1F
LATCH    =$4DE2
BUFF1    =$54B0
START    =$A000
EXTRTN   =$C800
         BIT  PIA
         CMP  $C2D7
         DEC  BUFF1+10
         EOR  BUFF1
         JMP  START
         JSR  EXTRTN
         LDA  %1101100001011111
         ROR  $11CF
         SBC  BUFF1+$1F
         STA  LATCH
```

TABLE 2.1 6500 MICROPROCESSOR INSTRUCTION SET-ALPHABETIC SEQUENCE

* Instructions legal only in the implied addressing mode.

ADC	Add Memory to Accumulator with Carry
AND	AND Memory with Accumulator
ASL	Shift Left One Bit (Memory or Accumulator)
BCC	Branch on Carry Clear
BCS	Branch on Carry Set
BEQ	Branch on Result Zero
BIT	Test Bits in Memory with Accumulator
BMI	Branch on Result Minus
BNE	Branch on Result Not Zero
BPL	Branch on Result Plus
* BRK	Force 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 Falg
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
EOR	Exclusive-OR Memory with Accumulator
INC	Increment Memory by One
* INX	Increment Index X by One
* INY	Increment Index Y by One
JMP	Jump to New Location
JSR	Jump to New Location Saving Return Address
LDA	Load Accumulator with Memory
LDX	Load Index X with Memory
LDY	Load Index Y with Memory
LSR	Shift Right One Bit (Memory or Accumulator)
* NOP	No Operation
ORA	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
ROL	Rotate One Bit Left (Memory or Accumulator)
ROR	Rotate One Bit Right (Memory or Accumulator)
* RTI	Return from Interrupt
* RTS	Return from Subroutine
SBC	Subtract Memory from Accumulator with Borrow
* 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 Pointer to Index X
* TXA	Transfer Index X to Accumulator
* TXS	Transfer Index X to Stack Pointer
* TYA	Transfer Index Y to Accumulator

PAGE ZERO ADDRESSING. In practice, the zero page addressing mode (identical in concept to absolute addressing) is the most frequently used. This allows the expression of the instruction to be two bytes instead of three; the low byte of the data address is taken from memory, and the high byte is assumed to be zero. All instructions legal in absolute mode are also legal in zero page mode with the exception of the JMP and JSR instructions (see Table 2-1); A/65 automatically generates the shortest possible code. It is good programming practice to reserve page zero (memory locations 0-255) for declaration of variables.

For example:

```
0083  106E                        MODE    =$06
0084  106E                        KEY     =$0C
0085  106E                        COUNTR  =$37
0086  106E                        TTYBUF  =$6B
0087  106E  65  37                 ADC  COUNTR
0088  1070  24  7A                 BIT  $7A
0089  1072  C4  06                 CPY  MODE
0090  1074  E6  38                 INC  COUNTR+1
0091  1076  A6  6B                 LDX  TTYBUF
0092  1078  46  9A                 LSR  $21+@171
0093  107A  05  0C                 ORA  KEY
0094  107C  86  AA                 STX  TTYBUF+$3F
```

IMMEDIATE ADDRESSING. The immediate mode of addressing is coded by the character "#" followed by a byte expression; the code inserted into memory is treated as the data to be operated upon according to the machine code.

For example:

```
0059  1046                        DOLLAR  =$24
0060  1046  69  03                LAB1    ADC  #3
0061  1048  29  B5                        AND  #$10110101
0062  104A  E0  24                        CPX  #DOLLAR
0063  104C  A9  45                        LDA  #'E
0064  104E  A0  46                        LDY  #<LAB1
```

IMPLIED ADDRESSING. Twenty-five of the fifty-six instructions, legal only in the implied addressing mode, require no operand-their execution may be completed with no other information than that contained in the opcode. These instructions are preceded by a * in Table 2-1.

For example:

```
0041  1033  00                    BRK
0042  1034  D8                    CLD
0043  1035  C8                    INY
0044  1036  EA                    NOP
0045  1037  68                    PLA
0046  1038  60                    RTS
0047  1039  8A                    TXA
```

ACCOMULATOR ADDRESSING. Instructions implementing the four shift operations have, in addition to addressing modes referencing memory, a special mode which allows manipulation of the accumulator. Usage of this mode similar to implied addressing, causes generation of a single byte of machine code.

For example:

```
0049  103A  0A              ASL  A
0050  103B  4A              LSR  A
0051  103C  2A              ROL  A
0052  103D  6A              ROR  A
```

RELATIVE ADDRESSING. There are eight conditional branch instructions available to the programmer; normally these immediately follow load, compare, arithmetic, and shift instructions. Branch instructions uniquely use the relative addressing mode. The branch address is a one-byte positive or negative offset, expressed in twos complement notation, from the run-time program counter. At the time the branch address calculation is made, the program counter points to the first memory location beyond the branch instruction code. Hence, the one-byte offset limits access to branch addresses with 129 bytes forward and 126 bytes backward from the beginning of the branch code. (A one-byte twos complement number is limited to the range -128 to 127 inclusive.) An error will be flagged at assembly time if the branch target lies out-of-bounds for relative addressing.

For example:

```
0054  103E  90  80              BCC *-126
0055  1050  F0  FE        HERE  BEQ HERE
0056  1042  30  FC              BMI *-2
0057  1044  70  7F              BVS *+129
```

INDEXED ADDRESSING. Indexed addressing (possible with index registers X or Y) facilitates certain types of table processing. The address given as the operand is treated as the base address, to which the contents of either the X or the Y register is added to arrive at the actual address of the memory location contining the data to be operated upon. All instructions implementing absolute indexed addressing with the X register also allow the same addressing in the page zero mode; several instructions (LDX, LDY, STX, and STY) allow zero page indexed addressing with the Y register.

For example:

```
0096  107E                    ARRAY  =$0B
0097  107E                    NUMBUF =$50
0098  107E                    TABLE  =$22C0
0099  107E  79  4F  00              ADC  NUMBUF-1,Y
0100  1081  DD  C0  22              CMP  TABLE,X
0101  1084  D6  0B                  DEC  ARRAY,X
0102  1086  5D  CC  22              EOR  TABLE+$C,X
0103  1089  B6  0B                  LDX  ARRAY,Y
0104  108B  36  22                  ROL  >TABLE,X
0105  108D  96  8F                  STX  NUMBUF+$3F,Y
```

INDIRECT ADDRESSING. The concept of indirect addressing constitutes a level of complexity beyond that of absolute addressing. The operand address references not one memory location containing data, but a sequence of two memory locations containing the address-stored in low-byte, high-byte order- of the location containing the actual data to be processed. True indirect addressing is offered only with the JMP instruction; otherwise, indexed indirect addressing with the X register and indirect indexed addressing with the Y register are implemented. For indexed indirect addressing, the indexed address is computed before the indirect is taken; the order of evaluation is reversed for indirect indexed addressing. Note that normal indirect addressing takes place when the index register contains zero. The JMP indirect uses an absolute-length (two-byte) operand; others require the operand address to lie in page zero between 0 and 254 inclusive.

For example:

```
0107  108F                        INDADR =$02
0108  108F                        CURSOR =$57
0109  108F                        OLDPTR =$7E
0110  108F                        NEXT   =$D9
0111  108F  21  02                      AND  (INDADR,X)
0112  1091  D1  7E                      CMP  (OLDPTR),Y
0113  1093  6C  D9  00                   JMP  (NEXT)
0114  1096  B1  57                      LDA  (CURSOR),Y
0115  1098  E1  02                      SBC  (INDADR,X)
0116  109A  81  57                      STA  (CURSOR,X)
```

A/65 DIRECTIVES

There are nine assembler directives; these are used to set symbol and location counter values (=), reserve and initialize memory locations (.BYTE, .WORD, .DBYTE), and control both assembler input/output (.OPT,.FILE, .END) and assembler listing format (.PAGE, .SKIP). All may be considered as assembly-time instructions, rather than as execution-time instructions.

Equate Directive

The equat ("=") directive assigns the value of an expression containing no forward references (symbols defined in a following section of code) to either a symbol or the location counter:

```
0118  109C                        *=$1800          ;EQUATE DIRECTIVE
0119  1800              TABLE2 =$C800              ; ASSIGN SYMBOLS
0120  1800              WRDPTR =$2A
0121  1800              NUMPTR =WRDPTR+2
```

A label used with an equate directive which increments the location counter will reserve work area memory locations; this is especially useful when consecutively allocating uninitialized memory at the beginning of a program:

```
0124  1800                        *=0              ;EQUATE DIRECTIVE
0125  0000              EOT     *=*+1              ; RESERVE MEMORY
0126  0001              EOTADR  *=*+2
0127  0003              BUFFER  *=*+72
0128  004B              ENDFLG  *=*+1
```

Symbols assigned one-byte values may be programmed as assembler constants- assembly-time values, used consistently throughout a program, which may be changed at a later time when the program is reassembled. Source code is designed so that alteration simply requires reassignment of the corresponding assembler constants. This is considered good programming practice and is a much better alternative to changing each constant as it occurs throughout a program.

```
0130  004C                    *=0          ;EQUATE DIRECTIVE
0131  0000              STRTCH =$28        ; ASSEMBLER CONSTANTS
0132  0000              ENDCH  =$29
0133  0000              DELIM  =$2C
0134  0000              LOWCH  =$41
0135  0000              HIGHCH =$5A
0136  0000              KEYLEN =4
0137  0000              BUFLEN =72
```

.BYTE Directive

The .BYTE directive initializes byte memory locations. Multiple arguments, separated by commas, may be specified in a single .BYTE command to load consecutive memory locations; either ASCII strings or expressions evaluating to an eight-bit value are legal. ASCII strings in .BYTE directives must not generate more than 20 characters.

```
0141  1800  41  42            ASCII  .BYT  'ABCD','EFGH','JOE''S'
0141  1802  43  44
0141  1804  45  46
0141  1806  47  48
0141  1808  4A  4F
0141  180A  45  27  53
0142  180D  00                .BYT <ASCII,>ASCII+2-%1,<*+>*,2
0142  180E  19
0142  180F  27
0142  1810  02
```

Note the use of two quotes within an ASCII string to insert a single quote into memory.

.WORD Directive

The .WORD directive is very useful in constructing jump tables and initializing pointers. An operand expression is evaluated as a two-byte address and is stored in low-byte, high-byte sequence, the order in which the microprocessor fetches addresses from memory. As with .BYTE, multiple operand fields, separated by commas, are allowed:

```
0144  1811  04  C8            JMPTBL  .WOR  $C804,$E4B9,$F77A
0144  1813  B9  E4
0144  1815  7A  F7
0145  1817  02  00                    .WOR $2,<JMPTBL,>JMPTBL,*,@6371
0145  1819  11  00
0145  181B  18  00
0145  181D  1D  18
0145  181F  F9  0C
```

.FILE Directive

This directive is used to "link" source files together. Often source files are longer than may be edited in available memory. If this happens, a .FILE command may be placed at the end of each block of code to indicate the next file name. The general form is:

.FILE XXXXXX

where "XXXXXX" is any valid file name. The last statement in the file must be:

.END XXXXXX

where "XXXXXX" is the name of the first file in the chain.

.END Directive

This must be the last statement in the program and indicates the end of source code input.

.DBYTE Directive

It if is desired to generate a sixteen-bit expression value in normal high-byte, low-byte order, the .DBYTE assembler directive must be used. Its syntax rules are the same as those for .WORD:

```
0148  1821  C8  04            DATA    .DBY  $CB04,$E4B9,$F77A
0148  1823  E4  B9
0148  1825  F7  7A
0149  1827  00  0E                    .DBY  $E,<DATA,>DATA,*,%010110111101
0149  1829  00  21
0149  182B  00  18
0129  182D  18  2D
0149  182F  05  BD
```

.PAGE Directive

The .PAGE directive (applicable to the assembler source listing) is used both to cause a page eject (top-of-form) and to generate or alter the title printed at the top of the new page. A title may be specified as an ASCII string in the operand field, and it may be cleared with a string of one or more blanks. Absence of an operand will cause the title printed on the previous page, if any, to be repeated at the top of the new page. This command is not printed as entered in the source code-only the results appear. For example, entry of:

```
.PAGE  'ORIGINAL TITLE'
.PAGE
.PAGE  'NEW TITLE'
.PAGE  ' '
```

would cause the fllowing to appear at the top of each page:

```
ORIGINAL TITLE. . . . . PAGE 0001
LINE  #  LOC                CODE LINE

ORIGINAL TITLE. . . . . .PAGE 0002
LINE  #  LOC                CODE LINE

NEW TITLE. . . . . PAGE 0003
LINE  #  LOC                CODE LINE

. . . . . .PAGE 0004
LINE  #  LOC                CODE LINE
```

.SKIP Directive

Blank lines, up to the end of the current page, may be inserted in the program listing with the .SKIP directive. If no operand is given, one line will be skipped; otherwise, the number of lines on the current page corresponding to the value of the operand expression will be left blank. Like .PAGE, this command does not appear as input:

```
                    *=0
                    .SKIP
CURSOR              *=*+2
EOT                 *=*+2
                    .SKIP 2
TWO                 =EOT
```

causes the following listings to be printed:

```
0002  0000                              *=0

0004  0000                    CURSOR    *=*+2
0005  0002                    EOT       *=*+2

0007  0004                    TWO       =EOT
```

.OPT Directive

The four options of the .OPT directive control generation of output files and expansion of ASCII strings in .BYTE directives. These options are selected by specifying:

```
.OPT   LIST, GENERATE, ERRORS, SYMBOL
```

and are eliminated by coding:

```
.OPT   NOLIST, NOGENERATE, NOERRORS, NOSYMBOL
```

Since only the first three characters of each option are scanned, the following may be written:

```
.OPT   LIS,GEN,ERR,SYM
.OPT   NOL,NOG,NOE,NOS
```

The four options control aspects of the listings as clarified below:

1. LIST [NOLIST] controls generation of the program listing, which contains assembled source input, generated object code, errors, and warnings.

2. GENERATE [NOGENERATE] controls the printing of object code for ASCII strings in the .BYTE directive. Only code for the first two characters is listed if NOG is specified; otherwise, the whole literal will be expanded.

3. ERRORS [NOERRORS] controls the listing of only erroneous program source lines together with the respective messages generated. Fatal assembler table overflows are also messaged in this file.

4. SYMBOL [NOSYMBOL] controls the listing of the symbol table at the end of the assembly. Also, it controls retaining the symbol table memory for subsequent use in disassembling instructions during debugging traces.

2.5.3 COMMENTS

Comments may be freely inserted into source code following the last field on a line. If preceded by an opcode (and possibly operand) field, the comment may optionally begin with a semicolon(;). Otherwise, the semicolon is necessary. A comment may be the only field on a line.

For example:

```
0033  1030                    ;COMMENTS MAY EXIST ALONE
0034  1030                                ;ANYWHERE ON A LINE
0035  1030                    L1            ;THEY MAY FOLLOW LABELS
0036  1030                    L2  =$3AB7    AFTER EQUATES AND
0037  1030  A9  1A                LDA #$1A  OPERANDS A SEMICOLON
0038  1032  0F                  .BYT $F     IS OPTIONAL
```

2.6 OUTPUT

This section describes the two output files generated by A/65-the listing and memory files. The listing file contains the program listing and symbol table. The memory file contains the object code produced by the assembly. The existence of each file is controlled by the IFLAGS variable or by the .OPT directive.

The program listing contains, for each source statement line, the corresponding line number (under the heading LINE #), the hexadecimal location counter (LOC), the one, two, or three bytes of generated code (CODE), and the image of the source code input to A/65 (LINE). In the upper left corner of each page is the page heading specified as the ASCII literal operand of a .PAGE assembler directive, followed by the page number. Error and warning messages appear after erroneous statements. (For an explanation of error codes, see Appendix A). At the end of the porgram is a count of the errors and warnings found during the assembly.

The symbol table contains an alphabetically sorted list of all symbols used in the program, and the value of each symbol.

APPENDIX A - Summary of Error Codes

1. UNDEFINED SYMBOL
2. LABEL PREVIOUSLY DEFINED
3. ILLEGAL OR MISSING OPCODE
4. ADDRESS NOT VALID
5. ACCUMULATOR MODE NOT ALLOWED
6. NOT USED
7. RAN OFF END OF CARD
8. LABEL DOESN'T BEGIN WITH ALPHABETIC CHAR.
9. LABEL GREATER THAN SIX CHARACTERS
10. LABEL OR OPCODE CONTAINS NON-ALPHANUMERIC
11. FORWARD REFERENCE IN EQUATE OR ORG
12. INVALID INDEX - MUST BE X OR Y
13. INVALID EXPRESSION
14. UNDEFINED ASSEMBLER DIRECTIVE
15. NOT USED
16. NOT USED
17. RELATIVE BRANCH OUT OF RANGE
18. ILLEGAL OPERAND TYPE FOR THIS INSTRUCTION
19. OUT OF BOUNDS ON INDIRECT ADDRESSING
20. A,X,Y,X, AND P ARE RESERVED LABELS
21. PROGRAM COUNTER NEGATIVE - RESET TO 0
22. SYMBOL TABLE OVERFLOW

SYNTAX ERROR MESSAGES

Descriptive error messages accompanying the statement in error are given in the listing file. The following is an annotated, numerically-ordered list of all syntax error messages. The following error messages generate only partial code; leaving undefined results where code cannot be generated.

** UNDEFINED SYMBOL (Error #1)

The assembler has found a symbol in an operand expression which is nowhere defined (given a value by appearing either as a label or as the destination field of an equate directive) in the source code. This error will also occur if a reserved name (A, X, Y, S, or P) is referenced as a symbol in an expression; these names are not defined in the symbol table when the programmer attempts to assign values to them.

Check for use of reserved symbols, misspelled labels, or missing labels.

** LABEL PREVIOUSLY DEFINED (Error #2)

The first field on the line, interpreted as a symbol, has been found already defined with a value in the symbol table. This symbol has been associated with a value by appearing as a label or as the destination field of an equate expression somewhere previously in the source code. Redefinition of symbols is not allowed. This error may also occur if a page zero variable is referenced before it is defined.

Check for a misspelled label or an attempt to use a label for two different purposes.

** ILLEGAL OR MISSING OPCODE (Error #3)

The assembler has found a line containing a label followed by an expression which it tried to interpret as an instruction. If the field following a label begins with a semicolon (";") it is treated as a comment; otherwise, either an assembler directive or one of the 56 instruction mnemonics is expected.

Check for: two or more labels defined on the same line; a label followed by a misspelled instruction, a comment without a leading semicolon, or an operand field; a comment line not preceded by a semicolon.

** ADDRESS NOT VALID (Error #4)

An address referenced in an instruction or in one of the assembler directives (.BYTE, .WORD, or .DBYTE) is invalid. An instruction operand value generated by the assembler must be greater than or equal to zero and less than or equal to hexadecimal FFFF (1 bytes long). (This excludes relative branches, which are computed and messaged separately.) If the operand expression generates more than 2 bytes of code or less than zero, this error message will be printed. For a .BYTE directive each operand is limited to one byte, and for a .WORD or .DBYTE each operand is limited to two bytes. All values must be greater than or equal to zero.

Check the values of symbols used in the operand field (see the symbol table for this information).

** ACCUMULATOR MODE NOT ALLOWED (Error #5)

Following a legal instruction mnemonic and one or more spaces is the letter "A" followed by 1 or more spaces (denoting the accumulator addressing mode). The assembler tried to use the accumulator as the operand; however, the instruction in the statement is one which does not allow reference to the accumulator.

Either check for a statement (illegally) labelled A to which this statement is referencing or, if trying to reference the accumulator, look up the valid addressing modes for the mnemonic used.

** RAN OFF END OF CARD (Error #7)

This error message occurs when the assembler is looking for a needed field and runs off the end of the card (line image) before the field is found.

The following should be checked for: a valid opcode field not followed by a necessary operand field; an instruction mnemonic that was thought to be legal for implied addressing, but which in fact needs an operand; an ASCII string that is missing the closing quote (make sure any embedded quotes are doubled; to insert a quote in a 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 an operand field, indicating either more operands or an index to follow.

** LABEL DOESN'T BEGIN WITH ALPHABETIC CHARACTER (Error #8)

The first non-blank field, being neither a comment nor a valid instruction, is assumed to be a label. However, the first character of the field begins with a numeric character (0-9), violating the rules of symbol construction.

Check for an unlabelled statement with only an operand field that starts with a number. Also check for an illegally labeled instruction.

** LABEL GREATER THAN SIX CHARACTERS (Error #9)

The first field on the line is a string containing more than six characters. Not being preceded by a semicolon, denoting a comment, it is assumed to be a symbol whose length limit has been exceeded.

Check for a label that is too long, either lack of spacing or an invalid separator between a label and an opcode, or a comment line with a long first word that doesn't begin with a semicolon.

** LABEL OR OPCODE CONTAINS NON-ALPHANUMERIC (Error #10)

The label or opcode field on a line is at most six characters long, but it (illegally) contains a character which is not alphanumeric. Note that symbols contain one to six alphanumeric characters, the 56 machine instruction mnemonics contain three alphabetic characters each, and assembler directives, following the initial period, are scanned for three alphabetic characters.

Check for the following: one or two labels on a line, one of which contains an invalid character; an instruction which either contains an illegal character or is not separated by a blank from its operand field; a comment, either following a label or standing alone, which is not preceded by semi-colon.

** FORWARD REFERENCE IN EQUATE OR ORG (Error #11)

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 and assign addresses (values) to both symbols and the location counter. Processing of input values is done sequentially, which means that all symbols fall into two classes -- those already defined and those not previously encountered. As it parses source code, the assembler assigns defined symbols and builds a table of undefined but referenced symbols; then, when a previously referenced symbol is defined, its value is substituted into the table. The cross assembler processes all of the input statements a second time, inserting values for all expressions -- even those which are "referenced forward" to a subsequently defined symbol.

Due to the sequential processing of the assembler and the dependence of the value of the location counter on symbols, 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.

This error may also mean that the symbol referenced is not defined at all in the program, in which case the cure is the same as for undefined variables.

** INVALID INDEX - MUST BE X OR Y (Error #12)

A legal operand expressions follows an opcode; following this expression is a comma (denoting indexed addressing) and an invalid string where either X or Y was expected. This error will be given whether an indexed addressing mode is legal for the corresponding instruction mnemonic or not.

** INVALID EXPRESSION (Error #13)

While evaluating an expression, the assembler found a character it couldn't interpret. This can occur if the operand expression contains illegal or misplaced operators or invalid constants or symbols.

Check the operand field to make sure expression syntax is legal.

** UNDEFINED ASSEMBLER DIRECTIVE (Error #14)

If a period is the first character in a non-blank field, the assembler interprets the following three characters as an assembler directive. Either an invalid directive has been found or the first three characters of one of the options in the .OPT directive are uninterpretable.

Check for a misspelled directive, a period at the beginning of a field that is not a directive, or an illegal option as a .OPT operand.

** RELATIVE BRANCH OUT OF RANGE (Error #17)

All of the conditional branch instructions are assembled into 2 bytes of code. One byte is for the machine code, and the other is for the branch address offset. To allow both forward and backward branches, the offset is in two's-complement representation; it is added to the address of the beginning of the next instruction to compute the new run-time program counter value. If the value of the offset is from 0 to 127 the branch is forward; if the value is from 128 to 255 (interpreted as from -128 to -1) the branch is backward. Therefore, a branch instruction can only branch either forward 127 bytes or backward 128 bytes relative to the beginning of the next instruction. If an attempt is made to branch further than these limits, this error message is printed.

**ILLEGAL OPERAND TYPE FOR THIS INSTRUCTION (Error #18)

After finding an instruction mnemonic that does not allow implied addressing, the assembler passes to the operand field (the non-blank field following the mnemonic) and determines what type of operand it is (indexed, absolute, etc.). If the type of operand found is not valid for the instruction, this error message is printed.

Check to see which operand types are allowed for the mnemonic and make sure the form of the operand type is correct.

** OUT OF BOUNDS ON INDIRECT ADDRESSING (Error #19)

An indirect address is recognized as such by the parentheses that surround it in the operand field of an instruction menmonic. Since indirects require two bytes of page zero memory, the address referencing this area must be less than or equal to 254. (Note especially that an indirect address of 255 is illegal - this would cause the two-byte area to fall partially outside of page zero). This error will only occur if the operand field is in correct form (i.e., an index register follows the address).

To correct this, the address field must refer to page zero memory between 0 and 254 inclusive.

** A, X, Y, S, AND P ARE RESERVED LABELS (Error #20)

The programmer has attempted to define one of the five reserved names (A, S, Y, S, and P) as a symbol. These names have special meaning to the assembler and cannot be used in this manner. The symbol is not defined and, if referenced elsewhere in the program, will appear in the symbol table as an undefined variable. Consequently, error messages will be printed as if the symbol were never declared.

Appendix B. A/65 Memory Map

$0000 A/65 uses all of page zero memory.
$0100 Reserved for the hardware stack.
$0200 Start of A/65.
$1B00 End of A/65.
$2300 Start of DOS (disk operating system).
$2F78 End of DOS.
$2F79 Start of A/65 Source file disk buffer.
(designated as page Ø and 1 swap buffer by OSI)
$3A79 Start of A/65 object code disk buffer.
$457A Start of free memory - (symbol table).