ASM65 USERS MANUAL

Larry Fish 1980

ASM65 is a two pass resident assembler for 650X machine code. It is a professional quality assembler designed specifically to develop large machine language programs on small machines. Some of the features include:

High Speed (over 1900 lines per minute)
Forward References
Ten Psuedo-Ops
Alphabetized Symbol Table
Cross Reference Listing
Offset Assembly
Separate Source Modules
Opcode Expansion

INTRODUCTION

An assembler is a program that is used to simplify the development of machine language programs. Generally, the development consists of four basic steps:

1) Editing
2) Assembling
3) Loading
4) Saving and Executing

In the first step, the program text is generated using an editor. During the second step, the assembler converts the program into the actual operation codes for the computer. In the third step, a loader is used to move the opcodes and data into the computer´s memory. At this point, the program can be tested, executed, or saved on disk for later use. These four steps are repeated until you are satisfied that the program is operating correctly.

GETTING STARTED

Assuming that you have already written a source file, assembling a program is quite simple. From the operating system type the following:

```
ASM filename.ext
```

This will cause the operating system to set up the specified file for assembly. If the file has the extension "P65" it is not necessary to type the extension, since the assembler automatically expects the "P65" extension. The output file will automatically have the extension "BIN".

When the assembler starts running it will print the following message:

```
1) DEFAULT MODE
2) CHANGE DEFAULTS
```

Most of the time you will run the assembler in default mode. This will assemble your file and put the binary in a file on the disk. The file can then be loaded and executed or saved on disk. To load the program type the following:

```
        LOAD filename.Ext
```

Again the extension is usually omitted, since Loader expects the "BIN" extension. The Loader loads the file into memory and returns to the operating system. From here you may either execute or save the file. This is accomplished using the SAVE, SET and START commands. Refer to the Loader section of this manual and the APEX manual for complete information.

WRITING PROGRAMS

The assembler takes specially formatted text called "Source" and and converts it into machine loadable binary text. It also creates a combined text containing hex addresses, opcodes, source and symbol table. This text is called a "Listing".

The purpose of an assembler is to simplify the task of generating machine language programs. If you´ve ever hand assembled a program you can apprecate how tedious it is to look up opcodes, keep track of jumps, and calculate branches. The assembler does all of this for you. Better yet, it allows you to make changes in a program without recalculating all of the jumps and branches.

The assembly process takes place in two stages, called passes. Two passes are necessary to allow forward references. A forward reference occurs when code near the beginning of a program calls code near the end of the program. During assembly both passes take place automatically and the whole process is transparent to the user.

During pass one, the assembler counts through all of the code in the source and retains the address of each symbol. This process is called building a symbol table. The complete symbol table must be built before pass two is started. During the second pass, the assembler generates machine loadable binary and a listing.

During the assembly, the assembler keeps a count of the current address. This counter is called the program counter or the location counter. As each byte of code is assembled the program counter is advanced by one. As you will see, many operations reference the counter. The user may use its value or reset it to begin a new program segment.

CREATING A SOURCE FILE

A source file consists of series of formatted commands to the assembler. Generally, you will generate the source text by typing it into one of the editors available under APEX. Here is a sample source file for a short program:

```
;SHORT PROGRAM
        .DEF    WRT=$FBFD        ;ROM TV OUTPUT ROUTINE
        .LOC    $800
START:  LDA#    ´A+$80
        JSR     WRT              ;OUTPUT A CHARACTER
        JMP     START            ;START OVER
        .END
```

The source text is divided into four fields. Each field is separated by a TAB character (Control-I). Starting from left to right the fields are: Label Field, Opcode Field, Operand Field, and Comment Field. Each line of the source text may have items in one or more fields. For example, the fourth line of the sample program has a label, an opcode and an operand, but nothing in the comment field.

LABELS

Labels are used to designate a certain line of code symbolically. For example, the jump instruction in our sample must jump somewhere. Instead of jumping to an exact address that may change if we alter the program, the jump instruction jumps to a symbolic location called "START". The exact address of "START" may be unknown until final assembly.

Labels can also be used to hold some numeric constant. For example, the value of a Carriage Return could be placed in a label called "CR" for reference by other part of the program.

Label names can contain as many as 10 characters including numbers; except that the first character cannot be a number. The label must be followed by a colon and a tab (Control I). Label names are usually choosen so that they describe the function of the line designated. For example, the label "START" indicates that this line is the start of the program.

OPCODES

The second field contains either an opcode or a pseudo-opcode. An opcode is mnemonic representation of a machine language operation. For example, LDA represents the machine language operation of loading the accumulator. Each opcode and its mnemonic is described in detail in the MOS Technology Programming Manual. Since there are thirteen addressing modes available in the 6502 processor, the mnemonic must also indicate addressing modes. Addressing modes are represented as follows:

LDA#	IMMEDIATE MODE
LDA	ABSOLUTE MODE
LDA	ZERO PAGE MODE
ROLA	ACCUMULATOR MODE
LDAX@	INDEXED INDIRECT X MODE
LDA@Y	INDIRECT INDEXED Y MODE
LDAX	ZERO PAGE INDEXED X MODE
LDAX	ABSOLUTE INDEXED X MODE
LDAY	ABSOLUTE INDEXED Y MODE
BCC	RELATIVE MODE
JMP@	INDIRECT MODE
LDXY	ZERO PAGE INDEXED Y MODE

Notice that the format is slightly different from the MOS Technology assembler. In this format the opcode carries the addressing mode instead of the operand. This makes source text cleaner and simpler. The assembler automatically chooses zero page addressing when the operand is on zero page. No special symbol is required for zero page addressing.

EXPANDED OPCODES

ASM65 has a set of special opcodes which execute special operations. These opcodes perform common operations that require more than one regular instruction to perform. When the assembler encounters one of these special opcodes it substitutes all of the regular instructions that are needed to accomplish the operation. For example, it is frequently necessary to move two bytes (e.g. a 16 bit address) from one memory location to another. Of course, you would usually do this by writing two loads and two stores. With the special opcodes the same operation can be accomplished by writing a single line.

When the listing is printed, only the first three bytes of the expansion is printed.

Each special code is designed to generate optimal code for its defined function, but since they can never be entirely optimal and improvements are always possible they should never be assumed to generate a specific expansion. Assumptions that are not a part of the definition are discouraged. In general,

i) No expansion will affect the X or Y register.

ii) No expansion may be assumed to leave the AC with a particular value unless so defined. In general the double precision operations leave the AC undefined while single precision operations leave it defined.

iii) No double precision expansion can be assumed to leave the status flags with a particular value.

In the following A and B are addresses and n is a constant.

DOUBLE PRECISION OPERATIONS:

				min: bytes/cycles
DINC	A	A+1 => A		6/8
DDEC	A	A-1 => A		6/9
DMOV	A,B	A => B		8/12
DMOV#	n,A	n => A		6/8
DADD	A,B	A+B => B		13/20
DADD#	n,A	n+A => A		11/13
DSUB	A,B	B-A => B		13/20
DSUB#	n,A	A-n => A		11/13
DADM	A	(AC)+A => A		9/11
DPSH	A	A => STK	low byte first.	6/12
DPOP	A	STK => A	high byte first.	6/14

SINGLE PRECISION OPERATIONS

ADD	A	(AC)+A => AC	3/5
ADD#	n	(AC)+n => AC	3/4
SUB	A	(AC)-n => AC	3/5
SUB#	n	(AC)-n => AC	3/4
MOV	A,B	A => B & AC	4/6
MOV#	n,A	n => A & AC	4/5

EXTENDED ACCUMULATOR OPERATIONS

INCA	(AC)+1 => AC	3/4
DECA	(AC)-1 => AC	3/4
ASRA	(AC)/2 => AC	4/11

SUBSTITUTE MNEMONICS

The opcodes "Branch Carry Set" and "Branch Carry Clear" can also be thought of as "Branch If Greater or Equal" and "Branch Less Than" because of the way they work after a compare. The assembler allows you to substitute these opcodes for the normal BCC and BCS:

BLT == BCC   BGE == BCS

PSEUDO-OPS

Psuedo-ops are direct commands to the assembler. They are placed in the opcode field. All psuedo-ops begin with a dot ".". The following pseudo-ops are available in ASM65:

.BYTE

Instructs the assembler to set aside one byte at the current assembly address. The value of the byte can be set by an argument in the operand field. If no argument is used the byte is set to a default value of zero.

```
        .BYTE    $45
```

If the assembly address had been $1000, this pseudo-op would have set aside one byte at address $1000, with the value of the byte set to $45. Here are more examples:

```
        .BYTE    %012
        .BYTE    START
```

.HBYTE

Instructs the assembler to set the value to the most significant byte of a two byte argument. This may also be accomplished by placing a ">" character in front of the operand:

```
        .HBYTE   START
        .HBYTE   $4567
        .BYTE    >START
```

.WORD

Instructs the assembler to set aside two bytes at this location. The value maybe set by an argument in the operand field. The bytes are set up so that the least significant byte of the argument is placed into the first byte and the most significant byte into the second byte in memory. Thus, the bytes are placed in memory in the reverse order of the argument, in keeping with the 6502 addressing scheme.

```
        .WORD    $7066
        .WORD   START
```

.PAGE

Instructs the assembler to print a form feed and move the listing to the top of the next page. No argument is required.

.ASCII

Instructs the assembler to convert the characters in the argument to ASCII. The characters in the argument are enclosed in single quotes:

```
        .ASCII  ´NOW IS THE TIME´
```

.LOC

Instructs the assembler to reset the location counter to a new value as specified by the argument. If no location is at the beginning of assembly, the assembler starts the address counter at zero.

This operation allows the assembler to place the program anywhere in memory. Also, it can be used to generate a single program that has separate segments in different parts of memory.

```
        .LOC    $1000
        .LOC    BEGIN
```

OFFSET ASSEMBLY

One of the useful features of the assembler allows you to assemble a program at one set of addresses, and yet load the program into a different segment of memory. This is usually done because the program will be relocated after it is loaded. To accomplish an offset assembly, an extra argument is added to to the .LOC psuedo-op. The first argument is the normal assembly address. The second address is load address. Here are some examples:

```
        .LOC    $1000,$2000
        .LOC    START,LOAD
```

The program is assembled in the normal way using the first argument as the starting location for all internal assembly operations. The only real difference is that the program is loaded starting at the second address. The offset assembly continues until a new .LOC is encountered. If the new .LOC has only one argument, the offset assembly is discontinued.

.DEF

This operation is used to define or redefine the value of a symbol. Generally, it is used to define a constant, such as the ASCII value of a carriage return character. It is often used to define the starting address of external subroutines.

```
        .DEF    WRT=$72C6
        .DEF    LAND=WRT
```

.END

Instructs the assembler to finish the assembly process. It must be placed at the end of the source or an error will result.

.LIST .NOLIST

These two pseudo-ops allow you to list parts of the program instead of the entire listing. This is very useful when you have a slow printer or you want to save paper during the program development process. In this way you can make listings of only those parts of the program that you are currently working on.

To use the feature, place .NOLIST at the beginning of the source code. Place .LIST before any segment that you want to list. Place .NOLIST at the end of the segment.

If you wish to list the symbol table, put .LIST at the end of the source code. It is possible to print only the symbol table by this means.

.LINK

The .LINK pseudo-op allows you to split a source code up into separate modules which are linked together by the assembler at assembly time. This is a useful feature since many programs are so large that they cannot be edited conveniently in a limited amount of disk space. With .LINK you edit each segment of the program separately, even on a separate disk, and then copy them to a single disk where they will be linked during assembly.

The .LINK pseudo-op is placed at the end of each module except for the last one. Each LINK will be followed by the name of the next module in the program. When the assembler encounters the LINK it will open the file specified as it´s new input file. All of the modules must be on the same disk. Since the assembler automatically assumes that the file will have the extension .P65, only the file name is necessary. Here is an example:

```
        .LINK   STAMP
```

OPERANDS

Generally, an operand is the argument to an opcode or a pseudo-op. Operands can consist of labels, addresses, numbers or expressions. Numerical values can be represented in hex, octal or decimal form. Several special characters can be used within the operand field.

DOT "."

When ever a DOT is used as an argument in the assembler it represents the value of the program counter at that particular point in the assembly. Thus if the assembler has assembled five bytes of program starting at $1000, the value of DOT will be $1005. It can be used to advance the location counter:

```
        .LOC    .+8
```

It can also be used to set a symbol equal to the location counter:

```
        .DEF    FRON=.
```

DECIMAL OPERANDS

The assembler assumes that all numbers are decimal unless otherwise specified. Of course, when the program is loaded, the number will be the binary equivalent of the value.

PERCENT "%"

Percent is used to indicate that a number must be taken as an octal value:

```
        .DEF    GOL=%021
```

DOLLAR SIGN "$"

The dollar sign indicates that the number following is in hexadecimal representation:

```
        .LOC    $4500
```

SINGLE QUOTES "´"

A Single quote causes the assembler to convert the following alpha-numeric character into it´s 7 bit ASCII equivalent:

```
        LDA#    ´B
```

EXPRESSIONS

The assembler has the ability to evaluate a mathematical expression as a part of the assembly process. In this way, certain constants can be calculated as the program is assembled.

At this time, the assembler will do addition and subtraction but not multiplication or division. Here is a sample expression that calculates the number of pages in a program:

```
        .DEF    LENGTH=>FINISH->START+1
```

LEFT AND RIGHT ANGLE BRACKETS "<" ">"

Angle Brackets are used to specify the high or low part of a two byte value. Left Angle Bracket (<) is used to indicate the least significant part of a two byte value and Right Angle Bracket (>) indicates the most significant part.

```
        LDA#    <WRT
        LDA#    >WRT
```

COMMENTS

Comments allow the user to leave notes throughout the source text. All comments must be preceded by a semicolon. Comments may appear in any field and are ignored by the assembler.

Good programmers will put many comments in the source code of their programs. Not only does this aid another person when he has to use your code, but it also saves you the embarrassment of not being able to understand your own code a year later.

SYMBOL TABLE

The symbol table is a list of all the symbols that the assembler encounters during assembly. The table is printed at the end of the assembly listing and is in alphabetical order. Each entry is followed by the defined value of the symbol. If there is any error associated with the symbol, a single letter error code will follow the symbol´s value.

CROSS REFERENCE TABLE

One of the special features of the assembler is the ability to generate a cross reference table (CREF). This table contains a list of each symbol that was encountered in the program, and the location or locations where the symbol was referenced in the program. Here is a sample:

```
HANDY  0000 FDF0 C013
HERE   1234 ADAD
```

In this example, the symbol HERE is referenced in some way by two separate parts of the program. Once at $1234, and once at $ADAD.

The assembler will print the table at the end of the listing. If wish, you may disable the generation of Cross Reference by changing the default settings. This will make the assembler run slightly faster since it has less work to do. If the assembler runs out of memory while generating the table, a message will be printed and no CREF will be generated.

DEFAULT SETTINGS

ASM65 uses a default structure that allows maximum flexibility, while minimizing typing. The assembler runs under a series of defaults that control such things as where to send the listing, whether to generate a cross reference table, etc. When the assembler is run, you can do one of two things: either run the assembler under the defaults or change the defaults.

Under the APEX operating system, all input and output to a program runs through standard device channels. Each channel is numbered and is associated with a specific input or output device. Here is a list of the devices most commonly used by the assembler:

0	Console (Keyboard and TV Screen)
2	Printer
3	Disk files
7	Null device

For a complete list and description of I-O devices, refer to the APEX manual. The Console, Printer and Disk should be self-explanatory. The Null device deserves some explanation. This is a dummy device that is used when you want to throw away the output of a specific operation. For example, most of the time you will not want to generate a listing with every assembly. In this case the listing output will be sent to the Null device.

If you choose to change the default settings, the assembler will begin the following dialogue:

```
ENTER DEFAULT SETTINGS
DEVICE # FOR BINARY OUTPUT:
DEVICE # FOR LISTING OUTPUT:
WIDTH (127 MAX) OF LISTING DEVICE:
DEVICE # FOR SOURCE INPUT:
DO CREF (Y-N)?
```

Several of the questions ask where you want the the assembler to get or send some part of the assembly information. Simply answer these questions with the appropriate device number. For example, you will usually want to get the source code from a disk file, so you would answer the question with number "3".

Another feature allows the user to set the width of the listing device to any value between 0-127. If a line length exceeds the preset value, the remainder of the line is discarded. If, for example, you are using 80 column paper you will probably want to set the width to about 75 to give the listing neat margins. Also, as a part of the default settings, you may tell the assembler whether or not it should generate a cross reference table.

Most of the time when you make changes to the defaults, they only apply to the assembly that you are about to make, since the default changes aren´t made to the permanent copy of the assembler on your system disk. If you want to make default changes that will be there every time you execute this copy of the assembler, do the following:

Run the assembler:

```
        ASM
```

Now change the defaults so that they do what you want them to do. Then strike Control-P. This will enter APEX through the SAVER entry point and will allow you save this new copy of the assembler in place of the old. To save it type:

```
        SAVE ASM
```

For more information on the SAVE operation refer to the APEX manual.

LOADING

During the assembly process, the assembler generates a file that contains all of the opcodes and data generated during the assembly. These opcodes and data are in hexadecimal form and must be converted to binary before they can be loaded into memory. The file also contains addresses which indicate where the binary should be loaded into memory. This file is called a "BINARY" file and has the extension ".BIN".

The loader is a program that is used to convert the file to pure binary and place the binary at the appropriate place in memory.

The loader´s operation is completely automatic. To run it from APEX, simply type:

```
        LOAD filename.Ext
```

Usually, the extension will be ".BIN". The Loader will load the file into memory and reenter the operating system through the "SAVER" entry (see APEX manual).

Unlike the loaders for high level languages, the assembly language loader doesn´t know certain key pieces of information about the program it loads. For example, it has no way of knowing what the starting address of the program will be or how long, in total, the program is. The operating system must have this information before it can save or execute the program. There are two ways to deal with this. The first way is to use the SAVE and SET utilities to set the correct program parameters. The SAVE command takes arguments that allow you to specify the program´s size. The SET command allows you to set all of the remaining program parameters (see the APEX manual for more information).

The second method of setting up the program parameters is to put a some code into the assembly source that will load over the system parameter page and set it to the proper values. Here is a sample that could be used to set up some of the parameters. It would be placed at the end of the program.

```
.DEF    MARK=.   ;SET MARK TO END OF PROGRAM
.LOC    $BF00    ;SET PC TO PROGRAM PAGE
JMP     RSTART   ;RESTART ADDRESS OF THE PROGRAM
JMP     START    ;START ADDRESS OF THE PROGRAM
.LOC    $BF15
.WORD   START    ;START OF SEGMENT TO BE SAVED
.BYTE   >MARK->START+1  ;CALCULATE PROGRAM SIZE
.END
```

For a complete listing of all program parameters and more on the loader, refer to the APEX manual.

ERROR CODES

There are a number of possible error conditions that can be detected by the assembler. When the assembler detects an error, a message is printed on the console, along with a copy of the line in which the error occured. The error message is also printed into the listing. Certain types of errors are also indicated in the symbol table. The following is a list of error codes:

L	LABEL ERROR
M	MULTIPLY DEFINED ERROR
U	UNDEFINED ERROR
P	PHASE ERROR
I	INTERNAL ERROR
O	OPCODE ERROR
A	ADDRESSING ERROR
S	SYNTAX ERROR

LABEL ERROR.
This error generally indicates that there is something wrong with a label. Usually the label is terminated with an improper character. All labels must be terminated with a TAB (Control-I).

MULTIPLY DEFINED.
Here there is an attempt to place the same symbol as a label at two different places in a program.

UNDEFINED ERROR.
In this error, there is a symbol that is being referenced, but that has never been defined, either by setting it as the marker to a line or through the .DEF pseudo-op.

PHASE ERROR.
Phase errors occur where a symbol is defined in different locations from pass one to pass two. It usually occurs when a symbol is referenced first and then later defined as a zero page location. Since the label is referenced before it is defined, there is no way of knowing at that time whether it is zero page or not. Thus there is no way of knowing whether the operand will be one or two bytes.

INTERNAL ERROR.
This error picks up some of the more unusual conditions.

OPCODE ERROR.
This indicates that the assembler found an illegal or nonexistant opcode.

ADDRESSING ERROR.
This error can indicate several illegal addressing conditions. Most commonly, it is a relative branch that is out of range. Another error occurs when the address pointed to by the jump indirect operand falls across a page boundary. Due to a hardware deficiency in the 6502 chip the processor will not fetch the second byte of the address if it is on a different page. The assembler checks for this problem. It can be corrected by moving the address one byte forward, for example:

```
0200 6C FF20    JMP@    $20FF           !!!ADDRESS ERROR!!!

0200 6C 0021    JMP@    $2100           !!!FIXED!!!
```

SYNTAX ERROR.
This error appears when a line of source has a syntax problem. These are usually things like .DEF pseudo-ops without the correct argument.

OTHER ERRORS:

CREF OVERFLOW
This message indicates that the assembler has run out of memory while building the cross reference. When this error occurs the assembler simply stops biulding the CREF. Assembly continues. Its only consequence is that no CREF will be printed.

SYMBOL TABLE OVERFLOW
This message indicates that the assembler has run out of memory while building the symbol table. Since assembly cannot be continued without a complete symbol table the program is aborted.

The usual way to deal with this difficulty is to assign more memory to the assembler. See the Apex manual for details.

I-O ERROR
This occurs when the assembler encounters a difficulty while accessing some I/O device. The possible causes depend upon the device at fault. For example device 3, the disks files, will

give this error if there is not enough space on a unit, or if the unit is disabled for some external reason.

LINK FILE NOT PRESENT

This indicates that the program appears to be incomplete and no valid link has been provided. Perhaps the file is missing or perhaps you simply forgot to use the .END psuedo-op. Note that the last line of your file must contain a carriage return to be valid.