SYNASSEMBLER

SYNAPSE Software (c) 1982

An Adaptation by Steve Hales
of the S.C. Assembler II - Version 4.0

SYNASSEMBLER

INDEX

APPENDIX

INTRODUCTION

SYNASSEMBLER is a convenient and powerful tool for software development on the Atari computer system. The assembler uses standard 6502 mnemonics and syntax, and includes many useful features for creating, editing, assembling and testing your assembly language programs. Now assembly language programming is almost as easy as programming in BASIC.

Here is a summary of the most exciting features:

* Full use of the standard Atari Screen Editor
* Tab stops for opcode, operand, and comment fields
* Fast parameterized renumber and delete command
* Uses BASIC like commands for files (eg. LOAD, SAVE, BLOAD etc.)
* Labels up to 32 characters long (lowercase letters and . accepted)
* English ERROR messages
* Ability to append source programs from tape or disk
* Display of memory usage
* Multiple source files, using .IN directive
* Store object code directly to disk file, using the .TF directive
* Listing to screen option using .LI ON and .LI OFF
* Assembles 6500 lines/minute
* Local labels
* Offending line is listed after an ERROR occurs
* Value of .EQ and address of .BS are printed on listing
* ASSEMBLER, DOS, HARDWARE, and OS locations protected during assembly
* ATASCII literals in address expressions
* Symbol table printed in alphabetical order

Synassembler requires 48K of RAM and one disk drive to operate. Very large programs can now be developed, using the "INCLUDE" and "TARGET FILE" capabilities. These allow the assembly of multiple source files, and direct storage of object code on binary files.

We have included several sample programs to illustrate the use of the SYNASSEMBLER as well as being useful routines. These include:

1. HARSH SCROLL :A Utility to coarse scroll the Atari screen display

2. FINE SCROLL .:A utility to fine scroll the Atari screen display

3. PM MOVER.....:A utility to move players and missiles on the screen

4. BELL.....:Ring the bell using Atari sound generators.

Blanks are compressed in source files to conserve memory and save space on disks. The compression algorithm replaces any string of consecutive blanks with a single code byte. Also, Atari assembler files are compatible and only require minor modification to assemble correctly. (See APPENDIX for further detail). Synassembler uses a memory-efficient method of storing the symbol table, with variable length entries. The symbol table is maintained in alphabetical order, using a high speed hashing scheme. The symbol table is maintained in memory until a new assembly is started or the NEW command is typed. This allows the RUN and VAL commands to be more useful and effective.

Assembler error messages are printed on the Screen and accompanied by a pleasant bell like tone on the speaker. (At least as pleasant as an ERROR message can be). After an assembly error is detected the offending line is listed to the screen automatically, in a position for easy editing.

COMMANDS

There are three types of commands in the SYNASSEMBLER: EDITING commands DOS commands and MONITOR commands. The EDITING commands are used to control the Editor and the assembler. Commands are typed immediately after the prompt symbol, which looks like this [Ok.].

EDITING COMMANDS

There are seventeen editing commands in the SYNASSEMBLER.
All editing commands may be abbreviated to the first three letters if you so desire.

ASM.....Assemble source program, put object program into memory, and produce assembly listing.

COPy.....Duplicates specified lines in the source.

DELete.....Delete specified line.

FINd.....List all lines containing the specified string.

HIDe.....Changes the HIMEM pointer to 'hide' current source code prior to a MERge command.

INCrement.....Set the line # increment for automatic line numbering

LISt.....List the source program or specified lines of source code.

MEMory.....Display the beginning and ending address

MERge.....Use with HIDe to join source programs

MOVe.....Moves a line of source code from one specified location to another

NEW.....Delete the entire source program

RENumber.....Renumber all or specified lines of source code

REPlace.....Replace a specified string with another specified string

REStore.....Restores HIMEM pointer after HIDe and MERge.

RUN.....Begins execution of your object program

VAL.....Evaluates an operand expression and prints the value in hexadecimal

MON.....Exit the editor and enter the monitor

EDITING COMMAND DETAIL

The SYNASSEMBLER editor, combines the powerful Atari screen editing features with a BASIC-like line editor, Source programs are entered and edited in almost exactly the same way you would enter and edit an Atari BASIC program.

ASM command:

SYNASSEMBLER is a two pass assembler. The ASM command initiates assembly of your source program. During the first pass it builds a symbol table with the definition of every label that is used in your program. During the second pass the assembler stores the object code into memory or disk and produces an assembly listing. At the end of the second pass a list, in alphabetical order, of all the labels and their definitions is produced.

If any errors are detected during either pass, an error message will be printed as well as the offending line. The error message will briefly explain the type of error encountered and the line will be positioned for easy editing. All of these messages abort the assembly process so that as soon as you correct the error condition you may immediately restart the assembly.

If you are listing the assembly to the screen you may use the [CTRL]+[1] control to start and stop your listing. You may abort the assembly process by hitting the [BREAK] key in pass two of the SYNASSEMBLER.

COPY L1 L2

This command places a copy of line L2 just before L1 in the source. The new line is assigned line number L1. The old line L2 remains in the source. This command should be followed by a renumber command if there are multiple lines with the same line number in the source.

COPY L1 L2 L3

This command places a copy of lines L2 through L3 just before line L1 in the source. The old lines are assigned line number L1. The old line L2 through L3 remain in the source. This command should be followed by a renumber command if there are any multiple lines with the same line number in the source.

DELete command:

Deletes a line or a range of lines from your source program. Another way to delete a sinqle line is to type it's line #, followed immediately by a carriage return.

HIDe AND MERge

These two commands, when used with the LOAD command allow you to join a program from disk or tape to a program that is already in memory.

HIDe temporarily changes the HIMEM pointer so that it appears as if there were no source proaram in memory. To remind you that you are HIDe-ing, the prompt symbol changes to [H] ok. After HIDe-ing a program, you can load another one from disk or tape. Then you type MERge to join the two programs together.

After this sequence of commands the program which was already in memory will follow after the program just LOADed. If the line numbers are not already as you wish them to be, you can use the RENUMBER command to assign new ones.

For example, suppose that we have 2 source programs on the disk named "PART1" and "PART2". We want to join them together so that "PART1" precedes "PART2".

```
Ok.
LOAD "D:PART1"
Ok.
LIST
00010 *      PROGPAM NUMBER ONE
00020 MAIN   JSR SUBROUTINE
00030        RTS
Ok.
LOAD "D:PART2"
Ok.
LIST
00010 *       PROGRAM NUMBER TWO
00020 SUBROUTINE
00030         LDA BOAT.LOC
00040         ASL
00050         ASL
00060         RTS
Ok.
HIDE
[H] Ok.
LOAD "D:PART1"
[H] Ok.
LIST
00010 *       PROGRAM NUMBER ONE
00020 MAIN    JSR SUBROUTINE
00030         RTS
[H] Ok.
MERGE
Ok.
```

INCrement
Sets the increment used for automatic line number generation. The increment is normally 10, but you may set it to any value between 0 and 9999.

INC 5

FINd
The FINd command allows you to search through your source program for a given text string, and list all the lines that contain that string. The correct procedure for use of this command is as follows: Type FINd, followed by a space and then the string for which you are searching. Every character you type between the space and the carriage return is part of the search key. (NOTE: you may append or prefix spaces to any string to perform label searches.)

LIST L1 L2

Lists a single line, a range of lines or your entire program. It works just like the list command in BASIC. While a program or range of lines is listing you can use the standard Atari pause control [CTRL1], to start and stop the listing to the screen. You may abort the listing by pressing the [BREAK] key.

MEMory

Displays the beginning and ending memory address of the source program and of the symbol table.

```
Source program: $9B99-9C1F
 Source length: $0086
  Symbol table: $1F00-1F00
```

Memory between the top of the symbol table and the bottom of the source program is free to be used without clobbering anything.

MERge

SEE "HIDe and MERge"

MOVe L1 L2

This command places a copy of line L2 just before L1 in the source. The new line is assigned line number L1. The old line L2 is removed from the source. This command should be followed by a renumber command if there are multiple lines with the same line number in the source.

MOVe L1 L2 L3

This command places a copy of lines L2 through L3 just before line L1 in the source. The new lines are assigned line number L1. The old lines L2 through L3 are removed from the source. This command should be followed by a renumber command if there are multiple lines with the same line number in the source.

RUN

Begins execution of your object program. An expression MUST follow the RUN command to define the place to begin execution of the program. For example, "RUN BEGIN" will cause execution to begin at the point in your program where the label BEGIN is defined. Your program will return to SYNASSEMBLER by using an RTS instruction in your program. You may abort your program by hitting the [RESET] key. Or, you may use the [BREAK] key to break and fall back to the monitor.

NEW

This command acts just like it's BASIC counterpart. It deletes the current source program from memory and restarts SYNASSEMBLER just as though you were to reboot the program.

NOTE: A source program must, of course, be assembled into memory before it can be executed with the RUN command.

RENumber

Renumbers all or part of the lines in your source program with the specified starting line number and increment. There are three optional parameters for specyifying the line number to assign the first renumbered line (base), the increment, and the place in your program to begin renumbering (start). There are four possible forms of the command:

```
REN          Renumber the whole source program:
                BASE=1000, INC=10, START=0

REN #              Renumber the whole source program:
                BASE=#, INC=10, START=0

REN #1,#2     Renumber the whole source program:
                BASE=#1, INC=#2, START=0

REN #1,#2,#3  Renumber all lines from #3 through the end.
                BASE=#1, INC=#2, START=#3
```

The last form above is useful for opening up a "hole" in the line numbers for entering a new section of code.

```
OK.
LIST
00000 * A RENUMBER EXAMPLE
00003 START     LDA #100
00013           STA $95
00058           LDA #99
00103           STA $A0
00110           RTS
Ok.
REN
Ok.
LIST
00010 * A RENUMBER EXAMPLE
00020 START     LDA #100
00030           STA $95
00040           LDA #90
00050           STA $A0
00060           RTS
Ok.
REN 100
Ok.
LIST
00100 * A RENUMBER EXAMPLE
00110 START    LOA #100
00120          STA $95
00130          LDA #90
00140          STA $A0
00150          RTS
Ok.
```

```
REN 2000,4
Ok.
LIST
02000 * A RENUMBER EXAMPLE
  02004 START    LDA #100
  02008          STA $95
  02012          LDA #90
  02016          STA $A0
  02020          RTS
  OK.
  REN 3000,10,2008
  OK.
  LIST
  02000 * A RENUMBER EXAMPLE
  02004 START    LDA #100
  03000          STA $95
  03010          LDA #90
  03020          STA $A0
  03030          RTS
```

REPlace dS1dS2d

This command replaces all occurences of string S1 with string S2 in the source. d is a delimiter and must be a non-space printable character that does not appear in either, S1 or S2.

REPlace dSidS2dtP

This command causes a search to be made for string S1. The search starts at the beginning of the source. Whenever S1 is found, the line containing it is listed and the user is prompted for 1 of 3 actions:

Y or [RETURN]-relace S1 with S2 and continue.
N-do not replace S1 with S2 and continue search.
X-do not replace S1 with S2 and stop search.

d is a delimiter and must be a non-space, printable character
not appearing in either SI or S2.

REStore

Restores the root source program if an assembly is aborted while inside an "included" module.

The 'root source program" is the source program that is in memory at the time the "ASM" command is issued. If this source program uses the ".IN" directive to include additional source files, it is possible that assembly might be aborted either manually by typing a [BREAK] key during the listing phase, or automatically due to an error in the source program.

If the assembly is aborted during the time that the root program is hidden, the prompt character changes from "Ok" to "[I] Ok". The RESTORE command will reset the memory pointers so that the root program is no longer hidden, and change the prompt character back to "Ok".

You do not have to use the REStore command after an abort unless you wish to get back to the root source program for editing purposes. If you type the ASM command, the assembler automatically restores before starting the assembly.

If an assembly aborts due to an error in a source line, you may correct the source line, SAVE the module on the appropriate file, and type ASM to restart the assembly.

VALue

The VAL command will evaluate any legal operand expression., and print the value in hexadecimal. It may be used to quickly convert decimal numbers to hexadecimal, to determine the ASCII code for a character or to find the value of a label from the last assembled program.

EXAMPLE:

```
VAL 'T
$0054         00084
OK.
VAL 3493 + $3493
$4238          16952
OK.
VAL START + S12
$4200         16896
OK.
```

DOS commands:

LOAD and SAVE

These commands are used to store your source files onto Disk or Tape in the internal compressed form. This saves disk space and speed.

EXAMPLE: LOAd "D:GAME1.TXT" or SAVE "D:MISC.SRC"

BLOad and BSAve commands:

These commands are used to load and save BINARY files to disk.

NOTE: BLOad and BSAve function in the same manner as the L and K options in Atari DOS II.

EXAMPLE: BLOad "D:GAME.OBJ".
(This will load the binary file called GAME.OBJ into memory at the address where it was saved.)

BLOad "D:GAME.OBJ",$2000
(this will load GAME.OBJ starting at HEX 2000, not at the address where i t was saved.)

EXAMPLE: BSAve "D:GAME.OBJ",$2000,$4000
(This saves a binary file called GAME.OBJ from Hex location $2000 to $4000.

NOTE: the $ always must precede a hexadecimal number.
SYMASSEMBLER assumes a decimal number if the S sign is not present.

ENTer

This command allows you to enter ASCII text directly from tape or disk. It functions like the ENTER command in Atari BASIC. You can use this command to ENTER Atari assembler source files and then convert them to SYNASSEMBLER format.

EXAMPLE: ENTer "D:ATARIFIL.SRC" or ENTer"C:".

TYPe

The TYPE command is used to save your source to any device in full ASCII format.

EXAMPLE: TYPe "D:MYSOURCE.TXT"
(This comand saves the full ASCII source under
the MYSOLRCE.TXT file, to disk drive 1)
TYPe "P:" ... sends the source file to the printer.

DIRectory

The directory command is used to examine the contents of your diskettes.

EXAMPLE: DIR by itself will show you the catalog for disk drive 1
DIR "D:*.OBJ" will show anything in the catalog on drive 1 with an OBJ extender.
DIR "D2:*.TXT" will show anything in the catalog on drive 2 with an OBJ extender.

DOS

The DOS command jumps from SYNASSEMBLER into the resident DOS in your system.

OUTput

The OUTput command is used to redirect the output of SYNASSEMBLER to another device; eg. printer, disk, screen etc.

EXAMPLE: OUT "P:"

After changing the output you may use the ASM command to send assembled listings to the device specified. To cancel the redirection simply type OUTput without a filespec.

EXAMPLE: OUTput.

SYNAPSE MONITOR

The SYNAPSE monitor in SYNASSEMBLER allows you to examine, change, move, and verify memory. You may read and write to disk and cassette, dis-assemble machine-language programs; execute programs; perform hexadecimal arithmetic; read and write sectors directly to and from disk; and monitor program execution for debugging purposes.

MONITOR COMMANDS

DISPLAY MEMORY: adrs1.adrs2 [RETURN]
This command allows you to display the memory from address1 to address2.

EXAMPLE: 2000.4000 and [RETURN]

CHANGE MEMORY: adrs;data data .....

In order to change data at a particular address enter the address (in HEX of course), and then a semi-colon(;) after which you may enter as much data as you wish making sure that each byte is separated by exactly one space.

EXAMPLE: 2000;4C 00 9D

After having entered an address, you may just use a semi-colon to indicate the next location for your next data entry.

```
  EXAMPLE: 2000;4C
              ;00
              ;9D
```

This example has the same effect as the previous example.

DIS-ASSEMBLING MEMORY: adrsL

This command allows you to dis-assemble 20 instructions starting at the specified address. By typing L L again the next 20 instructions will be dis-assembled.

ADDITION AND SUBTRACTION [HEX): data+data or data-data

You may add or subtract data (in HEX) simply by entering data and pressing [RETURN].

MOVING MEMORY: adrs1<adrs2.adrs3M

You may easily move data from one part memory to another. You first specify the address into which you wish to move, and then the range of memory that is to be moved.

EXAMPLE: 2000<3FF0.4000M

VERIFY MEMORY: adrs1<adrs2.adrs3V

If you wish to compare two blocks of memory, you can easily do so by specifying the starting address of the block you wish to compare and then the range that you wish it compared to.

EXAMPLE: 2000<3FF0.4000V

DISK (READ and WRITE): adrs<sec1.sec2r (READ)
adrs<sec1.sec2w (WRITE)

This unique feature of the SYNAPSE monitor allows you to access the disk directly. The first parameter is the starting address of the buffer in which you wish to store the contents of the sec1 through sec2. Note:The READ and WRITE commands are lowercase. (CAUTION: BE EXTREMELY CAREFUL WHEN ACCESSING THE DISK DIRECTLY. YOU CAN EASILY OVERWRITE THE CONTENTS OF YOUR DISK.)

EXAMPLE: 2000<1.4r (reads sector 1 through 4 into
buffer starting at 2000)
2000<1.4w (wites the contents of buffer
starting at 2000 to sectors 1
through 4.)

RESTORING NORMAL MODE: N

This command tells the assembler to restore the original screen color and tab stops to the power-up specifications.

OTHER COMMANDS

These commands are mainly used for execution and debugging assembly language programs.

EXECUTE

The G command is used to execute a program from the monitor, by typing the program address and the G command.

[*] Ok.
4000G

This will execute a program at 4000

EXAMINE and MODIFY registers

The R command allows you to examine and modify the 6502 registers (A,X,Y,P,S).

```
[*] Ok.
R
A=05 X=10 Y=50 P=30 S=F7
```

They can now be modified with the ";" command

STEP and TRACE (see also DEBUGGING)

The S and T commands are for single stepping your assembly language program but the T repeats the S command indefinitely. The S command will execute one instruction:

```
[*] Ok
4000S
4000: A9 03    LDA   #$03
A=03 X=00 Y=00 P=90 S=F0
```

At this point you may modify the register.

The T command will do the same thing as the S command except it will just repeat it forever. To get out of this mode, just tap the [BREAK] key.

QUIT

The Q command will return you to the assembler.

SOURCE PROGRAM FORMAT

Source programs are entered a line at a time, with a five digit line number identifying each line. The line numbers may run from 00000 through 63999. Source program line numbers are kept sorted in line-number order; the numbers are used for editing purposes just as in BASIC. A blank must always follow the line number. After the blank, there are four fields of information: the label, opcode, operand, and comment fields. Adjacent fields must be separated by at least one blank.

Although the fields are not restricted to begin in any particular columns, it is convenient to enter them in this way for neatness. Therefore tab stops are built in to the SYNASSEMBLER at columns 9, 13, and 21.

LABEL FIELD:

May be left blank, or may contain a label of from one to 32 characters. The first character of a label must be a letter; remaining characters may be either letters or numbers. Labels are used to name places in your program to which you will branch, as well as constants and variables.

The standard tab settings leave enough room for only 9 character labels; however, you can go ahead and use 32 characters as long as there is at least one space between the label and the opcode. If you like, you may type labels on a separate line, with the opcode and the following fields left blank. The label will be defined as the current value of the location counter. There are some examples of this in various listings throughout the manuals and in the example source code on the disk.

OPCODE FIELD:

Contains a three-letter machine language mnemonic opcode, or assembler directive. However, opcodes may begin in any column after at least one blank from a label or two blanks from a line number.

OPERAND FIELD:

Usually contains an operand expression of some sort. Some of the 6502 instructions have no written operand, such as NOP, BRK, DEX, and others. In these cases the comment field may be started right after the opcode. Four of the opcodes (ROL, ROR, ASL, and LSR) may be used both with and without an operand. If no operand is present, you must type at least TWO blanks before a comment with these four opcodes.

COMMENT FIELD:

Comments are separated from the operand field by at least one blank. Actually, comments may begin earlier or later on the line, just so at least one blank separates them from the operand expression.

COMMENT LINES:

Full lines of comments may be entered by typing an asterisk
or a semi-colon (;) in the first column of the label field. This type of comment is useful in separating various routines from each other, and labeling their contents. It is analogous to the REM statement in BASIC.

DIRECTIVES

Twelve assembler directives are available through SYnASSEMBLER.

.AS (stores ASCII literals.)

.AT (stores ATASCII literals)

.BS [expression]. (RESERVE [expression] bytes at the current

.DA [expression] enter data

.EN ENd of sources optional

.EQ [expression] (EQuate labels)

.HS (define Hex data)

.LI OFF (Turn off the assembly listing.)

.LI ON (Turn on the assembly listing.)

.IN [filename] .(Include a source program the specified file.)

.OR [expression] Indicates the originating address of your assembled code.

.TA [expression] (Target Address)

.TF [filename] --Put the object program on the specified file.

ASCII STRING: .AS daaa...ad

The .AS directive stores the binary form of the ascii characters "aaa...a" in sequential locations beginning at the current location. If a label is present, it is defined as the address where the first character is stored. The string "aaa ... a" may contain any number of printing ASCII characters. You indicate the beginning and end of the string, by using any delimiter ("d" in the example), that you choose.

ASCII character codes are seven bit values. The .AS directive normally sets the high order bit (8th), to zero. Some people like to use ascii codes with the high order byte set to one, so SYNASSEMBLER includes an option for this.

.AS daaa....d sets the high order bits=0
.AS -daaa...d sets the high order bits=1

This syntax restricts the choice of the delimiter slightly, in that the delimiter can be any printing character other than a space or a minus sign. Since the Atari inverse characters are set with the high order bits to zero, you merely put the minus sign before the string and you will get the characters in inverse mode.

ATASCII STRING: .AT daaa...ad

This is the same as .AS except that output is in ATARI ASCII.

BLOCK STORAGE: .BS exp

Reserves a block of <exp> bytes at the current location in the program. The expression specifies the number of bytes to advance the location counter. If there is a label, it assigned the value at the beginning of the block.

The address of the beginning of the block will be printed in the address column of the assembly listing. If the object code is being stored directly into memory, no bytes are stored for the .BS directive. However, if the object code is being written on a file using the .TF directive, the .BS directive will write <exp> bytes on the file. All the bytes written will have the value of $00.

DATA ....label .DA expression (two bytes, LSB first)
     .DA #expression (one byte, LSB of expression)
     .DA /expression (one byte, MSB of expression)

Creates a constant or variable in your program. The value of the expression as one or two bytes, is stored at the current location. If a label is present, it is defined as the address where the first byte of data is stored.

```
4000: 0A 00 64
4003:   00 E8 03
4006:   10 27    00020 DEC.NUM       .DA 10,100,1000,10000
4008:   64       00030 VALUE         .DA #100
4009:   FB       00040 VAL2          .DA #-5
400A:   FF       00050 VAL3          .DA #$FF

                     ---     Symbol table ----

4000:   DEC.NLM
4009:   VAL2
400A:   VAL3
4008:   VALUE
```

END OF PROGRAM .: .EN

This defines the end of the source program. You would normally make this the last line, but you may place it earlier in order to assemble only a portion of the source program. If no .EN is present anywhere in your program, the assembler will assume you meant to put this the last line.

EQUATE.....: label .EQ <expression>

Defines the label to have the value of the expression. If the expression is not defined, an error message is printed (UNDEFINED LABEL), and the offending line is listed out. If you neglect to use a label with an equate directive an error message (UNDEFINED LABEL), is printed. In either case, the assembly is aborted so that you can correct the error. All page zero references must be made before they are used or all labels defined after that reference will be off by 1.

EXAMPLE

```
          00020        .OR $80
0080:     00030 NUM1   .BS 1
0081:     00040 NUM2   .BS 1
0082:   00050 TABLE1   .BS 1
00E6:     00060 ADR          .BS 100
        00070           .OR $4000
        00080 * main program *

        ---     Symbol table ---

00E6: ADR
0080:   NUM1
0081:   NUM2
0082:   TABLE1
```

HEX STRING: label .HS hhh...h

Converts a string of hex digits (hhh...h) to binary, two digits per byte, and stores them starting at the current location. If a label is present, it is defined as the address where the first byte is stored. If you do not have an even number of hexadecimal digits, the assembler aborts with an error message, (BAD ADDRESS), and lists the offending line. NOTE: Unlike hexadecimal numbers used in the operand expressions you must NOT use a dollar sign with the .HS directive.

LISTING CONTROL: .LI OFF and .LI ON

This pair of directives turns the assembly listing on and off. If .LI OFF is put at the beginning of the source program, and no LI ON is used, no listing at all will be produced. The program will assemble much faster without a listing, as most of the time is consumed putting the characters on the screen, and scrolling the screen up.

If you put .LI OFF at the beginning of your source program and .LI ON at the end, only the alphabetized symbol table will print.

You may also use this pair of directives to bracket any portion of the listing you may wish to see or not see.

INCLUDE: - .IN [file name]

Causes the contents of the specified source file to be included in the assembly.

The program which is in the memory at the time the ASM command is typed is called the "root" program. Only the root program may have .IN directives in it. If you attempt to put .IN directives in an included program, the "NESTED INCLUDE FILE" error will print.

When the .IN directive is processed the root program is temporarily "hidden" and the included program is loaded. Assembly then continues through the included program. When the end of the included program is reached, it is deleted from memory and the root program is restored. Assembly then continues with the next line of the root program.

The .IN directive is useful in assembling extremely large programs which cannot fit in memory all at once. It is also useful for connecting a library of subroutines with a main program.

The [filename] portion of the directive is in standard FILESPEC format.

```
00020   * START OF PROGRAM
00030                   .OR $5000
00040                   .IN "D:PART1"
00050                   .IN "D:PAPT2"
00060                   .IN "D:PART3"
00070                   .EN
```

ORIGIN: .OR <expression>

This sets the program origin and the target address to the value of the expression. Program orign, is the address at which the object program will be executed. Target address is the address is the memory address at which the object program will be stored during the assembly. The .OR directive sets both of these addresses to the same value, which is the normal way of operating. If you do not use the .OR directive the assembler will set both the program origin and the target address to $4000. If the <expressions> is not defined during SYNASSEMBLERS pass 1 prior to it's use in the .OR directive, an error message is printed and assembly is aborted. The error message that appears is "UNDEFINED LABEL" and the offending line is listed for easy editing.

TARGET ADDRESS: .TA <expression>

Sets the target address at which the object code will be stored during assembly. The target address is distinct from the program origin (which is either set by the .OR directive or default at $4000). The .OR directive directive as we have seen, sets both the origin and target address. The .TA directive allows the added control of setting only the target address. Object code is produced and ready to run at the program origin, but is stored starting at the target address.

TARGET FILE: .TF Ifilenarml

Causes the object code generated to be stored an a binary file rather than in memory. Only the code which follows the .TF directive will be stored on the file. Code will be stored on the file until another .TF directive is encountered, or until a .TA or .OR directive is encountered. The [filename] format is the standard ATARI filespec format.

When you wish to assemble a program which will execute at an address normally occuppied by the assembler ($9C00 through $C000) or an already resident source program, you need to use the .TA and the .OR directives. Set the origin first, using the .OR directive and then set the target address to a safe value using the .TA directive. It is always safe to start the target area at $4000.

```
                          00010 *    SAMPLE PROGRAM TO ILLUSTRATE
                          00020 *    THE .TA DIRECTIVE
                          00030              .OR $1000
                          00040              .TA $4000
       1000:  AD 0C 10  00050 MAIN     LDA TEMP.A
       1003:  AE 0D 10  00060          LDX TEMP.X
       1006:  AC 0E 10  00070          LDY TBIP.Y
       1009:  4C 00 10  00080          JMP MAIN
       100C:  32          00090 TEMP.A         .DA #50
       100D:  64          00100 TEMP.X         .DA #100
       100E:  22        00110 TEMP.Y   .DA #34
--- Symbol table ---
1000: MAIN
100C: TEMP.A
100D: TEMP.X
100E: TEMP.Y
```

As you can see in the example, the assembly language listing looks as though the program was stored at $1000. However, the object code is actually stored at $4000, which is the target address set in the .TA directive. If we disassemble memory starting at $4000, we see:

```
Ok
MON
Synapse monitor
[*] Ok
4000L

4000:   AD 0C 10 LDA $100C
4003:   AE 0D 10 LDX $100D
4006:   AC 0E 10 LDY $100E
4009:   4C 00 10 JMP $1000
400C:   32         ???
400D:   64         ???
400E:   22         ???
400F:   00         BRK
4010:   00         BRK
4011:   00         BRK
4012:   00         BRK
```

After the assembly is complete, there are several ways to position the code in memory where it really should be. You can save the object code on cassette or disk from its current location and reload it at the correct location for execution. Be sure not to reload it while executing the assembler or you may clobber it.

Another method is to enter the monitor and use the monitor move command (addr1 addr2.addr3M). This command will move the block of memory from addr2 through addr3 to the area beginning at addr1.

LABELS:

There are two types of labels used in SYNASSEMBLER: normal labels and local labels.

NORMAL LABELS:
The first character of a normal label must be a letter; subsequent characters may letters, digits, or the period character (".") The period is useful for making long labels readable. For example, the subroutine used to read the next source line might be named "read.next.source.line".

Tab stops are set up within the editor assuming that most of your labels will be 9 characters or less. However, since the assembler is relatively free-formatt you may type any length label followed by a blank and the opcode, operand, and comment fields. Or, if you wish, you may type the long label on a line all by itself. In this form the label is assigned the current value of the location counter, just as if you had appended ".EQ" to the line.

```
01000  * SAMPLE PROGRAM WITH LONG LABELS
01010  SOURCE.LINE.POINTER .EQ $13 AND $14
01020  CHAR.POINTER .EQ $12
01030  *
01040  READ.NEXT.CHAR.FROM.LINE
01050         LDY CHAR.POINTER
01060         LDA (SOURCE.LINE.POINTER),Y
01070         INC SOURCE.LINE.POINTER
01080         RTS
```

LOCAL LABELS:

SYNASSEMBLER introduces a new kind of label called "local labels". The main purpose for the local labels is to make programs more readable by reducing the number of label names you must invent. As a side effect, local labels save considerable space in the symbol table during assembly; they only reqire two bytes each. The use of local labels also encourages structured programming habits.

Local labels have a period as the first character followed by one or two digits. Any label from .0 through .99 may be used.

The local label must be within a page of the normal label or an error will result.

Since each set of local labels is associated with a particular normal label, you may reuse the same local label

Here is an example of three little routines in the same source program using normal and local labels:

```
01000 PRINT.MESSAGE
 01010             PHA  SAVE A-REGISTER
 01020   .1        JSR PRINT.CHARACTER
 01030             INY
 01040             LDA MESSAGES,Y
 01050             BNE .1              =0 FOR END OF MESSAGE
 01060             PLA         RESTORES A-REGISTER
 01070             RTS
 01080 *
 01090   GET.NEXT.CHARACTER
 01100            LDY CHAR.POINTER
 01110            LDA INPUT.BUFFER,Y
 01120            CMP #RETURN
 01130              BEQ .1    END OF LINE
 01140              INC CHAR.POINTER
 01150   .1         RTS
 01160   *
01170 GET.NEXT.NONBLANK.CHAR
 01180   .1         JSR GET.NEXT.CHARACTER
 01190              BEQ .2    END OF LINE
 01200              CMP #BLANK
 01210              BEQ .1
 01220   .2         RTS
```

MEMORY USAGE

The SYNASSEMBLER program is about 8000 bytes long, and occupies $9C00 through $BC1F in memory. The screen begins at $BC1F and goes through $BFFF, while the source program begins at the top of DOS and goes to $9C00.

During source program entry or editing, memory usage is monitored so that the source program does not grow so large as to overlap the symbol table. Overlapping will cause the "MEMORY FULL" error. message to print. During assembly, memory required by the symbol table is monitored to prevent the symbol table from overlapping the source program. Overlapping will generate the "MEMORY FULL" error message and abort the assembly.

In addition, memory usage by the object program is monitored, so that it will not destroy the source program, DOS, the Operating System, and hardware. Therefore, if the object program bytes are directed at any memory protected addresses the "MEMORY PROTECT" error message will be printed and assembly.

OPERAND EXPRESSIONS

Operand expressions are written using terms and operators. The valid operators are + and -. Terms may be decimal numbers, hexidecimal, labels, or an asterisk (*). The first term in an expression may be preceded by a + or a - sign.

DECIMAL NUMBERS

Any number in the range from 0 through 65535, written in the normal way.

HEXIDECIMAL NUMBERS:

Any number in the range from $0 through $FFFF. Hexidecimal numbers are indicated by a preceding dollar sign ($), and may have from one to four digits. Beware of leaving out the dollar sign; the assembler may be quite satisfied to think of your hexadecimal number as a decimal one if you omit the ($). In some cases even a number with letters in it, such as 23AB, may be acceptable; it may be interpreted as decimal 23 and a comment "AB".

BINARY NUMBERS

Any number in the range from 00000000 to 11111111. Binary numbers are indicated by the preceding percent (%) sign.

NUMBERS EXAMPLE:

```
Ok.
                  00010 * A DECIMAL NUIBER
4000: F1          00020 NUM1          .DA #241
                  00030 * A HEX NUMBER
4001: E3          00040 NUM2          .DA #$E3
                  00050 * A BINARY NUMBER
4002: D6          00060 NUM3          .DA #%11010110
                  00070               .EN
       ---     Symbol table ---
4000: NUM1
4001: NUM2
4003: NUM3
```

LABELS:

One to nine characters; the first character must be a letter, while the others may be either letters or digits. Labels must be

defined somewhere in the program if they are to be used in an expression. In some cases they must be defined prior to use in expressions to prevent an undefined or ambiguous location counter. For example, if the expression in the operand field of an origin (".OR") directive is not defined prior to use, the assembler will not know how to define any subsequent labels.

A problem can occur if you postpone the definition of page-zero variables until after their use in operand expressions. If these labels are used with instructions which could assume both absolute and zero-page address modes, a discrepancy in the location count will occur between pass 1 and pass 2, of the assembler. This discrepancy cannot be detected by the present design of the assembler, so make it a habit to always define your page-zero variables at the beginning of your program.

ASTERISK [*]:

Stands for, the current value of the location counter. This is useful for storing the length of a string as a constant in a program.

```
080A-   0B         1070 QT      .DA #QTSZ      #BYTES IN MSG
080B-   41 4E 59
080E-   20 4D 45
0811-   53 53 41
0814-   47 45      1080  .AS /ANY MESSAGE/
                   1090 QTSZ .EQ *-QT-1  # BYTES IN MSG
0816-   00 00      1100 VARW .DA 0      2-BYTE VARIABLE
0818-   00         1110 VARB .DA #0     1-BYTE VARIABLE
                   1120 HERE .EQ *
```

It is considered VERY POOR programming practice to include branch instructions in your program with operand expressions in the form "*-5" or "*+7". If you value your sanity and time, avoid them like the plague! They breed bugs that can be very difficult to find. Dont't be afraid to use another label or two, no matter how silly the names might sound.

ADDRESSING MODES

The MOS Technology 6502 microprocessor- used in the ATARI has many great features; one of the greatest is its variety of addressing modes. There are thirteen different modes in allt though no single opcode can use every one of them. The chart in the appendix shows which modes can be used with each opcode. But fir-stt here is a chart showing an example of each mode and the way it is written in assembly language.

MODE	EXAMPLE	
lmplied	DEX	at least two
Accumulator	ROL	blanks before
Relative	BNE expr	comments begin
Immediate	LDA #expr	
	LDA /expr	
Zero Page	LDA expr	
Absolute	LDA expr	Assembler uses
Zero Page,X	LDA expr,X	Zero Page form
Absolute,X	LDA expr,X	if possible;
Zero Page,Y	LDA expr,Y	if not, it uses
Absolute,y	LDA expr,Y	Absolute form.
(Zero Page,X)	LDA (expr,X)	
(Zero Page),Y	LDA (expr),Y	
(Absolute)	JMP (expr)	

For a full explanation of the modes and how to use them, I refer you to the MOS Tecnology Hardware and Programming Manuals, as well as the other references mentioned in the bibliography in Appendix IV.

SYNASSEMBLER has one syntactical addition. The immediate mode may be indicated by either a pound sign (#) or a slash (/). The "#" means that the least significant byte of the 16-bit expression value should be used.

The "/" means that the most significant byte should be used.

One use for this feature is in setting up the address of a subroutine or a buffer in a pointer. (A pointer is a pair of bytes containing an address which "points" at a subroutine or into a buffer.) For example:

```
0080:                  00010 ADR          .EQ $80
4000:  A9 0D           00020 START        LDA #SOUND
4002:  85 80           00030              STA ADR
4004:  A9 40           00040              LDA/SOUND
4006:  85 81           00050              STA ADR+1
4008:  A0 00           00060              LDY  #0
400A:  B1 80           00070              LDA  (ADR),Y
400C:  00              00080              BRK
400D:  10 20   50      00090 SOUND     .HS 102050
4010:  20 40   90      00100              .HS 204090

               ---             Symbol table ---
0080:  ADR
400D:  SOUND
4000:  START
400D:  SOUND
```

Trying to comprehend and remember thirteen different addressing modes can be very difficult therefore it is convenient to try to group them into categories. You may wish to consider the following breakdown: implied mode, relative mode, and other modes. "Other" modes now includes eleven modes, so you can break it down further: accumulator, immediate, direct, and indirect. Each of direct and indirect modes can be either indexed or not indexed, and either Zero page or Absolute. The following outline will give you a better idea of what has been described:

```
  I.    Implied
 II.     Accumulator
III.    Direct
        A. Not Indexed
           1. Zero Page
           2. Absolute
        B. Indexed by X-register
           1. Zero Paqe,X
           2. Absolute,X
        C. Indexed by Y-register
           1. Zero Page,Y
           2. Absolute,Y
 IV. Indirect
     A. Not Indexed - (Absolute)
     B. Indexed by X-register
        (Zero'Page,X)
     C. Indexed by Y-register
        (Zero Page),Y
```

IMPLIED MODE

In this mode, the address is implied by the nature of the instruction, the operand field is left blank. All of the opcodes in this class are only one byte long. They are:

```
        BRK     DEX     PHA     RTS  TAY
        CLC     DEY     PHP     SEC  TSX
        CLD     INX     PLA     SED  TXA
        CLI     INY     PLP     SEI  TXS
        CLV     NOP     RTI     TAX  TYA
```

RELATIVE MODE

This mode is used only by the conditional branch instructions. The expression is converted to a signed offset from the location following the branch instruction. The result must be in the range from -128 through +127 to be legal. All of these instructions occupy two bytes. They are:

BCC/BGE*	BEQ	BNE	BVC
BCS/BLT*	BMI	BPL	BVS

* you may use either form for greater than/less than branches.

OTHER MODES

Usage of the other eleven modes is much more complex. The table in the appendix shows which modes are defined for each of the remaining opcodes. These instructions occupy one byte in the accumulator mode, two bytes in any zero page modes, and three bytes in any of the absolute modes. They are:

ADC	AND	ASL	BIT	CMP
CPX	CPY	DEC	EOR	INC
LDA	LDX	LDY	LSR	ORA
ROL	ROR	SBC	STA	STY
STX	JMP	JSR		

You might notice especially that only four, opcodes are usable in the accumulator mode (ASL, LSR, ROL, ROR); that only two opcodes use the "ZP,Y" mode (LDX and STX), and that only one opcode uses the indirect absolute non-indexed mode (JMP).

EDITING FEATURES

Any time the cursor is at the beginning of a line, typing [TAB] will cause the next line number to be generated. Immediately after loading, the "next line number" will be 10. The number will be displayed as five digits and a trailing blank. The cursor will be in a position for the first character of a label, or the asterisk for a comment line, or a semi-colon.

The "next line number" is always the value of the previously entered line number plus the current "increment". The increment is normally 10, but you can set it to any reasonable value with the INCREMENT command.

If you type the [TAB] in any other position than the beginning of a line, it will cause a "tab" to the next tab stop.

TAB STOPS

The standard tab stops have been changed to allow for a nine character label before the opcode. Of course, you may use any length label from 1 to 32 characters followed by a blank and an opcode; but the use of the tab stops make for nicer looking programs. (Longer labels look nicer, if left on a line by themselves.)

CURSOR CONTROL

SYNASSMBLER allows continued use of the ATARI cursor controls by pressing the [CTRL] key plus one of the four arrow keys on the right side of the keyboard. In addition, SynAssembler makes full use of the ATARI screen editor.

DEBUGGING PROGRAMS

Each step ("S") command decodest displays and executes one instruction at a time, and the trace ("T") command quickly steps through a program, stopping when a BRK instruction is executed.

Each step command causes the monitor to execute the instruction in memory pointed to by the program pointer. The instruction is displayed in its disassembled form, then executed. The contents of the 6502's internal registers are displayed after the instruction is executed. Then the program counter is bumped up to point to the next instruction in the program.

Here's what happens when you list and then step through a sample program:

```
Ok.
MON
Synapse monitor
[*] Ok.
4000L
4000:   A9 05          LDA #$05
4002:   0A             ASL
4003:   0A             ASL
4004:   8D 00   20     STA     $2000
4007:   00             BRK
4008:   00             BRK
  *
  *
  *
4015:   00        BRK
4016:   00        BRK
[*] Ok.
```

STEP EXAMPLE:

```
[*] Ok.
4000 S
4000:    A9 05   LDA #$05
A=05    X=00 Y=00 P=30 S=FD
[*] Ok.
S
4002:  0A      ASL
A=0A X=00 Y=00 P=30 S=FD
[*] Ok.
200
0200:    90
[*] Ok.
S
4003:    0A      ASL
A=14    X=00 Y=00 P=30 S=FD
[*] Ok.
S
4004:    8D 00 20 STA
A=14 X=00 Y=00 P=30 S=FD
[*] Ok.
2000
2000:    14
[*] Ok.
S
4007:    00      BRK
4007:   A=14 X=00 Y=00 P=30 S=FD
[*] OK.
```

TRACE EXAMPLE:

```
Ok.
MON
Synapse monitor-
[*] Ok.
4000T
4000:   A9 05     LDA #405
A=05    X=00 Y=00 P=30 S=FD
4002:   0A          ASL
A=0A    X=00 Y=00 P=30 S=FD
4003:   0A          ASL
A=14    X=00 Y=00 P=30 S=FD
4004:   8D 00 20  STA #$2000
A=14    X=00 Y=00 P=30 S-FD
4007:   00        BRK
4007:   A=14 X=00 Y=00 P=30 S=FD
[*] Ok.
```

EXAMINING AND CHANGING REGISTERS

The EXAMINE command is invoked by pressing "R" which tells the monitor to display the contents of the five 6502 registers on the screen. To change these values, type the semi-colon and the new values:

```
[*] Ok.
R
A=0A X=FF Y=D8 P=B0 S=F8
*;B0 02
R
A=B0 X=02 Y=D8 P=B0 S=F8
```

APPENDIX I

MONITOR TRICKS

There are few tricks that you can use in the SYNAPSE monitor. These will generally make your life easier and programming less of a chore.

All monitor commands may be put on a single line if you separate the commands with a space.

EXAMPLE: D01F D01F D01F
This will display memory location D01F 3 times.

Some commands are only one letter long, so you needn't separate them with a space.

EXAMPLE: 2000LLLL
This will disassemble 80 instructions

Sometimes you may need to utilize certain commands repeatedly. You can do this by typing the command many times, or type it once and tell the monitor to repeat it for you.

EXAMPLE: N 2FC AD;0N
This will display location 2FC until
[BREAK] or [SYSTEM RESET) is pressed.

APPENDIX II

SYNASSEMBLER Memory Map

(assumes 48K memory)

0000-00EF : O.S. and Assembler zero page usage.

00F0-00FF : Free space

0100-01FF : 6502 hardware stack

0200-02FF : Operating System vector table

0300-03FF : IOCB vector table

0400-047F : DOS usage area

0480-04FF : Assembler usage

0500-05FF : Assembler input buffer

0600-06FF : Free space (if REPlace not used)

0700-1D00 : DOS II

1D00-9BFF : Free space for source, symbol table, and object code

9C00-BC1F : SYNASSEMBLER

BC20-BFFF : Screen display list and data

APPENDIX III

CONVERTING ATARI ASSEMBLER FILES

In order, to convert your ATARI Assembler/Editor files to SynAssembler format follow these simple instructions:

.Read the ATARI editor/assembler files into the SynAssembler using the ENTer command.

.Save the file back to the disk using the SAVE command. This will store your file in compacted format.

.Make the following changes to your source code:

1. Remove all references to "A". For examplet in the instruction LSR A, the "A" should be removed, since SynAssembler assumes the "A" reference.

2. SynAssembler has no multiply or divides so these must be put in by long hand.

3. To get the Hi and Lo bytes make the following changes:

```
Atari Assembler
        LDA #PLACE/256 high byte
        LDA #PLACE&255 lo byte

SynAssembler
        LDA /PLACE     high byte
        LDA #PLACE     lo byte
```

4. All of the Atari directives must be changed to the SynAssembler equivalents.

Now that you have your file in SynAssembler format, you may use the local labels and long 32 character labels.

APPENDIX IV

BIBLIOGRAPHY

Publishers have begun to release some good technical resource books for learning to program the 6502 microprocessor.

The ATARI Assembler, Don Inman & Kurt Inman.RESTON Publishing Company, 1981. Designed for the beginner to intermediated this book has 270 pages including many illustrations, diagrams and examples.

6502 Software Design, Leo J. Scanlon. One of the Blacksburg Continuing Education Series, published by Howard W. Sams & Co.,1980. 270 pages, paper, $10.50.

6502 Assembly Language Programming, Lance A. Leventhal. Osborne/McGraw Hill, Inc., 1979, over 80 programming examples.

Programming and Interfacing the 6502, with Experiments, Marvin L. DeJong. One of the Blacksburg Continuing Education Series, published by Howard W. Sams & Co., 198O. 414 pagest paperback.

6502 Software Gourmet Guide and Cookbook, Robert Findley. Scelbi Publications, 1979. 204 pages, paperback. Includes listings of conversion routines, search and sort routines, and floating point routines.

6502 Games, Rodney Zaks, SYBEX. The third in the SYBEX series on programming the 6502. Includes listings of games in assembly language.

Practical Microcomputer, Programming: the 6502, W.J. Weller, Northern Technology Books, 1980. 459 pages, includes a listing of a 6502 assembler and a debugging package.

APPENDIX V

	ACCUI-ULATOR	IMMED-IATE	DIRECT			INDIRECT		
				INDEXED			INDEXED	
	blank	#expr /expr	expr ZP/ABS	expr,X ZP/ABS	expr,Y ZP/ABS	(expr)	(expr,X)	(expr),Y
ADC	--	69	65/6D	75/7D	--/79	--	61	71
AND	--	29	25/2D	35/3D	--/39	--	21	31
ASL	0A	--	06/0E	16/1E	--/--	--	--	--
BIT	--	--	24/2C	--/--	--/--	--	--	--
CMP	--	C9	C5/CD	D5/DD	--/D9	--	C1	D1
CPX	--	E0	E4/EC	--/--	--/--	--	--	--
CPY	--	C0	C4/CC	--/--	--/--	--	--	--
DEC	--	--	C6/CE	D6/DE	--/--	--	--	--
EOR	--	49	45/4D	55/5D	--/59	--	41	51
INC	--	--	E6/EE	F6/FE	--/--	--	--	--
LDA	--	A9	A5/AD	B5/BD	--/B9	--	A1	B1
LDX	--	A2	A6/AE	--/--	B6/BE	--	--	--
LDY	--	A0	A4/AC	B4/BC	--/--	--	--	--
LSR	4A	--	46/4E	56/5E	--/--	--	--	--
ORA	--	09	05/0D	15/1D	--/19	--	01	11
ROL	2A	--	26/2E	36/3E	--/--	--	--	--
ROR	6A	--	66/6E	76/7E	--/--	--	--	--
SBC	--	E9	E5/ED	F5/FD	--/F9	--	E1	F1
STA	--	--	85/8D	95/9D	--/99	--	01	91
STX	--	--	86/8E	--/--	96/--	--	--	--
STY	--	--	84/8C	94/--	--/--	--	--	--
JMP	--	--	--/4C	--/--	--/--	6C	--	--
JSR	--	--	--/20	--/--	--/--	--	--	--