MAE 0.95
by John Harris

postscript file by Fredrik Lundholm
d1dol@dtek.chalmers.se 1995-05-09

Contents:

- INTRODUCTION -

Welcome to the MAE assembler. This is a work-in-progress, and as such is not quite finished. It is functional however, and provides a very comfortable environment for programming the 65816 processor. Any unfinished features, or planned improvements, will be addressed in this documentation when they come up.

The design goal of MAE was to provide a highly integrated environment for an editor, assembler, and debugger. It is very easy to use, and contains many features to save the programmer keystrokes, and development time. It does not take the approach of, "If I make the assembly speed fast enough, nothing else matters." I believe that only a small portion of a project's development time is spent waiting for the assembler to assemble your file. Therefore, programming efforts were mainly concentrated on making the editing and debugging processes easier. MAE is fairly quick on small program files, but it does not use a hashed symbol table routine, and so it can really slow down on large assemblies. If you program in a modular fashion, where you don't need to continuously reassemble program segments that are already done, speed problems can be avoided. For example, the editor module of this assembler is 2155 lines of source, and assembles to a disk file in 25 seconds.

MAE uses the standard E: device to allow easy compatibility with the XEP80 and other 80 column devices or software drivers. The speed of Atari's built-in 40 column routines is not the greatest, and so you will also find the HYPER E: screen accelerator on the disk. This will more than double the screen performance when it is installed, and it is highly recommended.

Here are some of the features of the MAE development system.

MAE provides an excellent full-screen type editor with many features such as key macros, automatic 'JSR' and return to subroutine labels, block moves and copies, and multiple undo.

Compatible with the XEP80, and probably most other video boards that provide an E: handler interface.

High level of integration between editor, assembler, and debugger. The editor can take you directly to lines that had assembly errors. The debugger can reference labels in the symbol table and assemble single program lines.

Very user friendly **--** it will save you a lot of typing.

True local labels that are referenced between global labels.

Can assemble directly to bank select memory or bank select cartridges.

Uses very little system RAM. Most of the code resides in bank select memory.

Full 65816 and 24 bit support.

- MEMORY USAGE AND CONFIGURATION -

This assembler is designed to run in one bank of the extended memory available on the 130XE or other memory expanded Atari. This bank number is configurable. A portion of the assembler must reside outside of bank select RAM, and in this version, it will occupy $A900-$BC1F. This is a much larger amount than what will eventually be needed, because the debugger is currently sitting entirely in main RAM, taking up 4K of it. When the debugger is moved to bank select memory as well, main memory usage should only be 1-2K.

Memory usage for the source file and symbol table is configurable. In addition, memory for the symbol table will use bank select RAM when the addresses are within the $4000-$7FFF range, while memory for the source file will always be in normal system RAM. This allows memory for the source file and symbol table to overlap, and gives you the largest possible free memory buffer. Having the symbol table in bank select RAM has one other benefit. It will usually stay intact so that the labels can be used from the debugger. The default values for memory usage are to place the symbol table from $6400-$7FFF and the source buffer from LOMEM to $A8FF.

This assembler isn't designed for a 64K machine, but it can be configured to operate in a limited fashion. Because the bulk of the code sits at $4000, that will be right in the middle of RAM if you don't have bank select memory. The best you will be able to do, is set the text buffer from $6400-$A8FF, and the symbol table from wherever your LOMEM is up to $3FFF. That will only give you about 17K for your text buffer, and your symbol table may not always be available for the debugger, but otherwise, all functions should work as described. There is a second version of the assembler on the disk, called MAE64.COM, that is configured for 64K use.

Zero page memory is saved and restored from the editor and assembler, so that they will effectively leave all of ZP available to the user.

All three modules use a $100 byte buffer in low memory that is currently at $400-$4FF.

When assembling to disk files, a $100 byte buffer is used for disk I/O. The location of this buffer is configurable, and is currently set to use $500-$5FF.

Buffer memory addresses and several configurable parameters are stored at the beginning of the program file. Eventually, I hope to provide an editor for these parameters, but for now, they will need to be modified by using a sector/file editor. The parameters are located at +3 bytes from the start address, which will be +9 bytes when counting the DOS binary header. Here they are: (Words are 6502 standard Lo,Hi)

Word - Text buffer start adr. If 0, the editor will use MEMLO. (Default)
Word - Text buffer end. If 0, the editor will use MEMTOP. Default $A8FF.
Word - Symbol table start adr. Default $6400. This is located just past the part of the assembler that resides in bank select RAM.
Word - Symbol table end. Default $7FFF. This configuration puts the symbol table entirely in bank select memory.
Word - Address for a 256 byte disk I/O buffer that is used only when assembling to disk files. Default is set to $500.
Byte - 0=Replace mode, 1=Insert mode (for the editor)
Byte - 0=Use spaces only, 1=Convert spaces into Tabs.

3 Bytes - Tab settings for Assembly code fields.

Byte - When drawing the screen for a particular location, such as a Find or Goto command, this byte sets the screen row where the desired line will be located. If you set this to 0, the line will be on the top row. $0C will put the line in the middle of the screen. Default is 6.

Byte - # of lines the PgUp & PgDn command will move by. Setting this to $18 gives single screen paging. If you normally use scrolling for short moves, you can set this value to something like $60 to jump through the file in larger steps.

Byte - Line length saved in the Undo buffers. Default is 39. You can increase this to 79, if you want entire XEP80 lines saved. You can also decrease this number to something like 25, if you want to increase the number of history buffers without using extra memory. You would only lose the comment field for any restored lines.

Byte - # of Undo buffers. This number sets how many lines of history can be undo'ed. Default is 16. Multiplied by the number above, equals the total size needed for undo storage, located at the end of the text buffer.

Byte - Sets what bank of extended memory to put the assembler. It should be a value appropriate for the $D301 register. Default is $E3.

String - Default drive and dir spec. Must be in the form Dn:?????, like "D3:*.*", or "D3:*.S" if you want dir listings to only show a certain file type. Filenames for loading and saving do not require a full "Dn:" filespec. If you type just a name, it will be preceded with the default drive selected here.

- THE MAIN MENU -

The program will display the main menu once it has loaded. During an editing session, the Esc key will return to this menu, and Esc will also return to the editor when you are in the menu.

Main Menu Commands:

A - Assemble current file. Hold the Shift key when pressing A to turn on the assembler listing. The only current way to send an assembler listing to the printer, is to first use the "O" debugger command to enable output echo to the printer.

B - Break to the monitor. Actually issues a 00 BRK instruction.

D - Go to DOS. If you return to the editor by running at the start address, the source file will still be intact. This is automatic in SpartaDOS by using the RUN command. For other DOSes, you will need to supply the starting address which is currently $A900, but may change in later versions.

L - Load file. You do not need to type an entire filespec. The default drive 'Dn:' will be prepended if the entered name doesn't have a ':'. Press Shift-L to append to the end of an existing file.

P - Do Pass 2 only of the assembly. A full assembly must have already been performed, and then this function may be used if the source code was changed in such a way that didn't effect any label addresses. This can save a lot of assembly time, but please do not use it unless you understand what it does and are certain that no label addresses have been changed since the last assembly.

S - Save File. Save displays the last loaded filename. Press Return to accept it, or backspace and change. Press Shift-S to save a marked text block. To do this, mark the starting line with ˆZ, then move to the ending line and enter the Esc-Shift-S command. To print a file or text block, you may need to convert the text to spaces only, and then save to P:.

T - Tab convert. It will prompt to convert to Tabs or Spaces. Tab converted text will have $7F TAB characters for any sequences of spaces that can be compressed. The conversion is done from back to front for best speed, but it can still take awhile. It is also possible to run out of RAM when converting to spaces. After this command is entered, the editor will remain in the specified mode. In other words, if you do a tab convert to spaces, all future entered lines will remain in an expanded space format.

V - (Value) Will allow you to enter any expression, and then displays the calculated value in both hex and decimal. This can be handy for getting the value of labels, or as a simple calculator.

X - Perform cross reference listing. The source file must already have been assembled with the A command. It allows you to enter a label name to start from, or just press return to create a cross reference for the entire symbol table. Be prepared for this to take awhile. This is a simple implementation that requires a full pass of assembly for each label. On the positive side, it doesn't require any memory to build an XRef table, and so generating an XRef on large files should not have any problems running out of memory. Plus, being able to specify label names directly makes it very easy, and much quicker, to get a report for one or two labels of immediate interest.

1-9 Get directory, and set default drive. If you just want to change the default drive without getting a dir listing, press Esc after pressing the desired number.

Shift-Clear - Clear text buffer.

- EDITOR SECTION -
- EDITOR FILE FORMAT -

The editor saves files in straight ATASCII. It can either keep all spaces expanded, or can use $7F TAB characters to reduce the size of the file. The text can be freely converted between these formats. TAB compression and expansion is done on a line by line basis while you are editing, which is different from the way most editors handle TABs. While you are editing a line, TAB characters are not present, and the line will edit in the same way as if it had only spaces. When the line is saved back into the file, the program will see if it can convert any sequences of spaces into TABs, based on the configured TAB fields. TABs are not rigidly enforced. Meaning, if you slide a comment field a little to the left to make more room, that spacing will be retained, and that particular place simply won't be TAB converted. Thus, it is simply a manner of saving memory and file space. The editor will actually run faster with the TAB setting on. Especially with the XEP80.

Currently, this editor will not allow lines longer than the right margin. It will allow 79 column lines with the XEP80 device though. Actually, the max line length is one shorter than RMARGN to avoid problems with the E: device sending an extra EOL. Any lines longer than RMARGN-1 will be split in two. One other problem with the line length should be pointed out. When using the character insert, it is possible to push a character into the last column on the screen. Please note that this character will NOT be saved as part of the text, and will disappear the next time the editor gets a chance to draw the line. In fact, I had to set LOGCOL equal to RMARGN*2 to prevent inserts from doing line wrap.

- EDITING COMMANDS -

Standard Atari editing keys apply, with a few exceptions. Clr & Set Tab have no effect. Use the configure bytes to change tabs. Pressing the Caps key without Shift will always set lower case instead of toggling. Use Shift-Caps to set upper case. 1200XL function keys, as well as the standard Atari arrow keys, can be used to move the cursor.

Return will insert a new line if pressed at the end of the current line. Otherwise, it will just move to the line below. Return does not break a line in the middle, which works out better when entering source code.

In the following tables, a "^" symbol means to press Control along with the key after the "^". "S^" means press both the Shift and Control keys.

- MOVING AROUND -

S^Up	Scroll Up
S^Dn	Scroll Down. These two commands are handy, because they scroll immediately, without waiting for the cursor to reach the screen's edge.
Sˆ<	Page up by configurable # of lines
Sˆ>	Page Down
ˆ,	Start of line
ˆ.	End of line
ˆT	Top of file
ˆB	Bottom of file
ˆS	Set mark at current location
ˆM	Go to Mark
ˆG	Goto line number. When entering the line number at the prompt, you can also enter a '+' or '-' as the first character to move a number of lines relative to the current location.

- FIND & REPLACE -

ˆF	Find text -- not case sensitive. '?' can be used as a wildcard. There is currently no way to change the wildcard character from within the editor, however, the editor uses the same wildcard configuration as the debugger does. See the documentation for debugger commands to find out how to change the wildcard character.
SˆF	Find by searching backwards towards the start of the file.
ˆR	Replace -- forward direction only. It will ask for a Find string and then a Replace string. For all matches of the find string, the editor will display an inverse '>' symbol in front of the occurrence. You may press Y to replace it, N to skip it, A to replace all occurrences to the end of the text, or Esc to abort. Max length for Find or Replace is 15 characters.
ˆN	Next. If the previous operation was a replace, then you will be in the prompted replace mode if the string is found. Otherwise, if the previous operation was a Find, then you will simply be taken to the next occurrence.
ˆP	Find Previous match, by searching backwards.
ˆL	Enter a label name, and the editor will jump to where the label is defined. (By searching from the first column only.)
ˆJ	The editor's version of a JSR. It looks at the operand field of the current line, and jumps to where that label is defined. It also sets a mark at the current line so that you can return with a ˆH. The use of this function is not limited to JSR instructions. JMPs, branches, even data variable locations can be traced with this command. Basically, for any line that contains an operand field, ˆJ will attempt to find the location where that label is defined.
ˆH	Return to previous position where a ˆJ command was entered. ˆH is also used to jump to locations that gave assembly errors. During the assembly, up to 16 error positions will be remembered. All bookmarks, and marks set from the ˆJ and error position reporting, will auto-adjust to any changes in the source text, so that they will always point to the correct line in the source file.

- BLOCK MOVES & COPIES -

^Z Set block start.
^X Cut from block start to current line, in a forwards direction only. If you find you want to mark a block backwards, the easiest way to do this, is to set the mark at the current line, then move backwards to the start of the block. Press ^Z, then press ^M to return to where you started, and give the ^X or ^C command. A clear to end of file operation can be done by pressing the keys, ^Z, ^B, ^X.
^C Copy text from block start to current line, into the cut buffer. The text is left unchanged.
^V Paste the cut buffer at current line.

- KEY MACROS -

^W (Write) Begin key macro recording. When done, press ^3. Up to 60 keystrokes can be recorded, including Esc-Menu commands.
^E Execute Macro
^K Prompts for a number, and then will repeat the next entered key that number of times. ^K cannot be used within a macro, but it can be used to execute ^E multiple times.

- MISC OTHER STUFF -

S^Del Delete to end of line
^I Toggle Insert/Replace mode.
^A Accept next keypress as literal ATASCII value. This lets you enter Control graphic or international characters that would otherwise be treated as editor commands.
^D Duplicate line.
^U Undo line deletes, or changed lines. Does not undo block operations. A handy way to move one or more lines from one place to another, is to delete the original lines, move to the destination, and then use the Undo operation.

- ASSEMBLER SECTION -
- EXPRESSIONS -

Expressions can be made from decimal numbers, hex numbers by using "$", binary numbers by using "%", single ASCII characters with a "'" (single quote), and label names.

Any of these values can be mixed with math operators +-*/, ! (bitwise OR), & (bitwise AND), and unary -. The vertical bar can be used in place of !.

There are also special operators that refer to the low byte, high byte, and bank byte (24-bit highest byte) of the calculated expression. These operators are <, >, and ˆ.

There is no operator precedence. All math is evaluated left to right, with the exception of <, >, and ˆ, which are done after the rest of the expression has been evaluated.

Examples of valid expressions:

```
    LDA #'A-$20          ;= $21
    LDA #-1              ;= $FF
    LDA #%101&3          ;= 1
    LDA #>$1234+1        ;= $12
    LDA #>$1234+256      ;= $13
    LDA #ˆ$123456        ;= $12
    LDA #>$123456        ;= $34 (mid byte)
```

All of these expression types can be used in .BY statements as well. Like:

```
LOWS    .BY <LABEL1 <LABEL2 <LABEL3
HIGHS   .BY >LABEL1 >LABEL2 >LABEL3
        .BY 15+3!%1000
    etc...
```

- LABELS -

The first character of a label may be any letter, or the symbols @, \, _, or ?. All remaining characters may also include numbers plus the symbols ., :, ;, <, =, >, and ˆ. Labels may be up to 15 characters long.

Label names, and for that matter all text entered with the assembler, can be entered in upper or lower case. Labels are not case-sensitive.

When the first character of a label is '?', the label is a 'local label'. Locals are defined only in the source code segment between two global (i.e. non-local) labels. References to local labels cannot cross a global label definition.

Internally, the assembler creates local labels by appending the local onto the end of the previous global label. Thus in the following code segment:

```
DELAY LDX #100
?L    DEX
      BNE ?L
```

'?L' is a local label, and will be entered in the symbol table as DELAY?L.

Knowing how the label is stored, allows you to access it from the debugger or the Esc-V expression evaluator. You can also code a direct reference to the label DELAY?L if you need to access the local from the other side of the global label DELAY.

Locals are not printed in X-reference or symbol table listings, which makes them very useful for simple loop and branch structures where you don't want to think up unique label names for all occurrences.

- ADDRESSING MODES -

All 6502 and 65816 addressing modes are supported. Any addresses that evaluate less than \$100 will use zero page modes when possible. Thus, zero page labels must be defined before being used, or assembly errors will result.

There is also a way to force 8 bit, 16 bit, or 24 bit addresses using the operators <, !, and >. (Yes, I know this is inconsistent with the immediate operators for low, high and bank bytes -- I didn't write the 65816 assembler specifications). This can be really useful for forcing absolute 16 bit addressing on zero page labels, to add 1 cycle in time critical applications. For the 65816, it can force direct page addressing for non-ZP labels, (which of course requires you to move the direct page register to the proper page address). ALL 24-bit addresses must be preceded by the > character.

The operands for the 65816 MVP and MVN instructions should be simple bank bytes -- not full addresses. Ex:

```
MVP $40 $80
```

moves memory from bank \$40 to bank \$80, using the addresses in X and Y.

Or:

```
MVP ˆSRC ˆDEST
```

Use the bank byte of the source and destination addresses.

- PSEUDO-OPS -

Note that only 2 letters are required, but if additional letters are present they will be truncated without assembly errors. For example, you may use pseudo-ops like '.byte' and '.org'. Personally, I really like having the pseudo-ops the same width as all 6502 instructions, and only use 2 letters.

```
.24
```

Sets the symbol table and program counter to use 24 bit addresses. This is only useful for 65816 programs, and may crash your machine if you try to use it without having a 65816 CPU.

```
.AB
```

The assembler will generate byte-sized values for accumulator-related immediate constants. (Default)

```
.AW
```

The assembler will generate word-sized values for accumulator-related immediate constants. This is only useful for 65816 programs.

```
.BA byte
```

For bank addressing, you can specify an operand to force assembler generated object code into bank select RAM. If the operand is >= $80, this value will be stored into location $D301 when storing bytes of object code into RAM. Operands less than $80 are placed in the X register, and a STA $D580,X is performed. This can control certain bank select cartridge devices. NOTE: For using the second method of bank addressing, the assembler needs to be able to return the bank select cartridge to normal. There is currently a 'STA $D5DC' for this purpose, but this may not be the right address for your cartridge setup. You should search the disk file for this instruction, ($8D $DC $D5), and replace it with the appropriate address.

    .BI filename

Includes the contents of a binary disk file into the assembly. If this file does not contain a DOS binary header, it will be assembled as in-line data at the current PC. Otherwise, a file that contains a header will be loaded at its load address.

    .BY [+byte] bytes and/or ASCII

Store byte values in memory. ASCII strings can be specified by enclosing the string in either single or double quotes.

If the first character of the operand field is a '+', then the following byte will be used as a constant and added to all remaining bytes of the instruction.

Ex:

```
    .BY +$80 1 10 $10 'Hello' $9B
```

will generate:
81 8A 90 C8 E5 EC EC EF 1B

Values in .BY statements may also be separated with commas for compatibility with other assemblers. Spaces are allowed since they are easier to type.

See also .SB which creates ATASCII screen codes.

    .CL

Close output object code file. When using the .OU pseudo-op to create object code files on disk, the file will normally be closed at the end of assembly. However, if you wish to close the file before that, it can be forced closed with the .CL pseudo-op. You may use this to create multiple output files in one assembly, or to place something in RAM in addition to the disk file.

    .DC word byte

Define constant-filled block. This will fill an area of size 'word' with the constant 'byte'.

    .DS word

Define storage. This will reserve an area of storage equal to size 'word'.

    .EC

Do not display macro generated code in the assembly listing. Only the macro call itself will appear.

    .EJ

Eject -- Send a form feed code to eject the page in an assembly listing.

.EN
Mark End of assembly. This pseudo-op *must* be present to end the assembly, or an error will result.

.ES
Display the code resulting from Macro expansions.

.HE hex bytes
Store hex bytes in memory. This is a convenient method to enter strings of hex bytes, since it does not require the use of the '$' character. The bytes are still separated by spaces however, which I feel makes a much more readable layout than the 'all run together' form of hex statement that some other assemblers use. Example:

```
.HE 0 55 AA FF
```

.IB
The assembler will generate byte-sized values for index register-related immediate constants. (Default)

.IW
The assembler will generate word-sized values for index register-related immediate constants. This is only useful for 65816 programs.

.IN filename
Include additional files in the assembly. Only the main source file can contain .IN pseudo-ops. You cannot nest them. Default drive processing works the same here as it does when loading files from the editor, and so you will usually not need any 'Dn:' types of filespecs. The file name only should be sufficient. No quotes are needed either.

.LC
Turn off (clear) the display of the assembly listing. (Default)

.LL
Display the assembly listing on this line only, even if the full listing is turned off. This can be extremely handy to display the program counter value at important positions in the source file.

.LO long
Stores a longword, 3 byte value in memory. Only one operand is supported.

.LS
Turn on (set) the display of the assembly listing.

.MC adr
Move Code to a different address when storing object code in memory. This is in case the .OR assembly address conflicts with something already at that location in memory.

!!!name .MD ([label1] [label2]...)
Begin macro definition. Described in a separate section.

.ME
End macro definition.

.MG
Mark the current .IN include file as Macro Global. This keeps this file in memory throughout the assembly, which is required if the file contains macros that are referenced in other included files.

.OC
Turn off (clear) the storing of object code in memory.

.OR adr
Sets the origin address for the assembly.

Note: If there is a label on this line, it will be given the value of the new origin. This is not the same as in Mac/65 which could use its origin directive to reserve space (*= *+1). You should use the .DS pseudo-op for reserving space.

.OS
Turn on (set) the storing of object code in memory. (Default)

.OU filename
Create an output disk file for the object code. Regretfully, this file is made up of individual 256 byte segments much like Mac/65 does. I apologize for the laziness here on my part, but it really was a lot easier to do this way. You will need to run some type of strip program to de-segment the file. The .OU pseudo-op should be placed above the .OR pseudo-op.

.PR "text"
Print a text message to the screen on pass 1 of the assembly. This is generally used with the .VA pseudo-op when prompting for values to be entered from the keyboard.

.SB [+byte] bytes and/or ASCII
This is in the same format as the .BY pseudo-op, except that it will convert all bytes into ATASCII screen codes before storing them. The ATASCII conversion is done before any constant is added with the '+' modifier.

label .VA
Will print a '?', and then accept input from the keyboard. You may enter any value, which will be given to the label in front of the .VA.

.WO word
Stores a word in memory. Only one operand is supported.

SET label = expression
Set the specified label to a new value. This instruction allows a label to be redefined with different values during the assembly. Any label can be SET.

- CONDITIONAL ASSEMBLY -

There are four conditional instructions IFE, IFN, IFP, and IFM, that represent conditional assembly if Equal, Not equal, Positive, and Minus. The operand of the IF instruction will be evaluated, and if the processor status codes match the type of IF statement, then the source code following the IF will be assembled. Mark the end of the conditional block of code with the pseudo-op '***'. (*** is like an ENDIF statement).

There is no 'ELSE' instruction, and so you must use complementing IF statements.

Examples:

```
IFN  FLAG
.           ;This block of
.           ;code gets asm'ed
.           ;when FLAG <> 0
***

IFE FLAG
.           ;This block does
.           ;when FLAG=0
***

IFN FLAG1!FLAG2
.           ;asm'ed if FLAG1
.           ;or FLAG2 <> 0

IFE  WIDTH-40
.           ;This gets asm'ed
.           ;when width=40
***

IFM WIDTH-40
.           ;asm'ed if WIDTH
.           ;less than 40
***

IFP WIDTH-40
.           ;if WIDTH greater
.           ;or equal to 40
```

- MACROS -

Macros must be defined before they are used in your source. The definition looks like this:

!!!name .MD ([label1] [label2]...)

Where 'name' is the name of the macro, and 'label1' etc. are its parameters, separated by spaces. The three exclamation marks are a special macro identifier, and must precede the macro name. If the macro takes no parameters, then omit the parenthesis after the ".MD". The body of the macro definition will follow, and should be ended with a .ME pseudo-op.

The number of parameters used when calling the macro must always match the number of labels in the definition. When called, these parameters will be placed into the label names, in order, where they can be used in the body of the macro. Parameters can only be expressions -- there is no method for passing arbitrary text strings. Labels used in the macro definition are stored in the symbol table along with regular labels. Thus, there must not be name conflicts between macro parameter labels and program labels. I suggest you adopt a naming convention for macro labels, like always starting them with "Z" or something, to make it easier to avoid name conflicts.

Macros can pass up to 8 parameters.

Any labels defined within a macro must use a special form. Because macros can be expanded multiple times, a special label type exists to avoid errors from multiple label definitions. These label types start with three periods, followed by any normal label name. These special macro labels will be given unique numbers with each macro expansion to keep them separate. You can consider them local labels to each macro expansion.

Here's an example of a macro to increment a two byte value:

```
!!!IND  .MD     (ZLOC)
        INC     ZLOC
        BNE      ...SKP
        INC     ZLOC+1
...SKP  .ME
```

To call this macro, you would use:

```
        IND     $80
```

There are more macro examples in the supplied include file MACROS.

- ERROR MESSAGES -

These are the error messages that can be produced by the assembler. Error messages are marked with an '!', and also include the source line number that they occurred on. If you are assembling a single file, or if the errors occurs in your main file, you will be able to use the editor ˆH command to jump directly to the errors. For errors that occur in included files, you will need to load in that file, and jump to those line numbers manually using the ˆG goto line number command.

BRANCH
Branch instruction out of range.

OPCODE
Error in opcode field. This can be either a bad 65816 instruction, bad pseudo-op, or an undefined macro.

DUP
Duplicate label definition.

EOF
End of File error. All assemblies must end with a .EN pseudo-op. This should be in the main source file, not in any included files. This error can also occur if a conditional or macro definition is pending at the .EN.

UNDEF
Undefined label reference.

NEST
Nested definition. Conditional IFs may not be nested. .MD macro definitions cannot contain additional definitions.

OPERAND
Error in operand field.

ADR MODE
Addressing mode not supported.

BAD LABEL
Bad characters in label name.

MACRO OV
Macro overflow in either the number of expansions, or level of nested expansions.

SYM OV
Symbol table overflow.

PARMS
Number of macro parameters in the call does not match the definition.

LABEL MISSING
Missing label on either a SET pseudo-op or in an = equate definition.

- DEBUGGER -
- GENERAL INFORMATION -

Note: 24 bit support, and the full 65816 instruction set are not completed yet. This is basically a 6502 debugger right now.

Filenames default to the current drive number which can be changed. (input of 'FILE' = 'D1:FILE') A full filespec will override the default.

Non destructive prompt character (.) for ease in full screen editing. Also, the prompt does not interfere with command decoding. If the cursor is moved up to redo a prior command, the '.' does not need to be deleted.

Upper and Lower case accepted.

The debugger is ZP clean, so all of ZP is available for the user.

You can look at RAM under the OS, by resetting the bit in $D301, as long as you are using SpartaDOS or some method of handling interrupts when the OS is disabled.

The debugger uses the E: handler, which can allow two screen debugging with some 80 column devices. (Your program is displayed through the Atari, while debugging output is on the 80 column device.) Currently, the XEP80 does not work very well in this manner, because its screen drivers require the Atari DMA to be turned off. You can partially support this by adding an external user function to toggle DMA. More information about this will be given in a later section. For machines without an 80 column device, the debugger supports flipping between two display lists, one for the E: screen, and one for your program. In all cases, there can be potential conflicts when trying to debug programs that use the E: handler themselves, as both the debugger and your program struggle for the same locations. The debugger's design is admittedly not ideal for use in this situation, but it works out well for programs that create their own screen.

Any continuous displays can be paused and stepped one line at a time with the space bar. Press 'C' to return to continuous display. ESC, RETURN, or BREAK will stop the display. While the display is paused, the V command for switching view screens, and also the U user function, can both be used.

ALL addresses and data bytes can be entered in HEX (default), in DECIMAL with # (#1234), in BINARY with % (%10011010), in ASCII with ' ('A) or as a label currently defined in the MAE symbol table with . (.LABEL). Arithmetic operators `+-*/&!` can also be used, and will be performed left to right. Any combination of these can be mixed at any time in a completely free format scheme, with no limits on length. (Ex: `2000-#256+'W/100`) Very little will be mentioned about this feature later on, but ALL numbers for ALL commands accept this versatile entry system.

All commands use spaces as delimiters. A '?' indicates a command error. Parameter uses for commands are abbreviated to:

adr:	a 16 bit address.
by:	an 8 bit byte. ('by' with numbers indicates a string of bytes.)
bit:	a 0 or a 1.
char:	an ASCII character.

Quantities in [brackets] are optional parameters. Default values will be used if they are not entered. All non-bracketted values must be entered. Any other upper case characters or symbols should be entered as stated.

'Current address' refers to the last displayed or changed address, (+$1), and is separate from the current program counter or PC.

- COMMANDS -

Display Memory M [adr] [adr][/]

Displays hex and ASCII. Displays 24 locations if only 1 parameter. Displays from current adr if no parms. '/' = to $FFFF. The '/' can be used on all other commands as well. Does not display ASCII control characters when output is being sent to an external device.

Peek Memory P adr1 [adr2..] [*]

Special memory display that allows multiple addresses to be entered, and only prints one byte per address. * causes a continuous print of the list of addresses, and is really useful for finding keycodes from $D209, or examining any locations that have changing data. Push Break to abort the continuous peek.

Change Memory :adr by1 [by2..by8]

The change memory command ':' can be entered directly, or edited from the display memory command. Only 8 data bytes will be changed. You can substitute the character = for the adr, which will then use the current address. This allows you to enter successive lines of bytes without requiring any other addresses. Ex:

```
:600 1 2 3 4 5 6 7 8
:= 9 A B C D E F
```

ASCII Mem Change C adr ASCII_STRING

Stores ASCII string at adr.

Disassemble D [adr]

Disassembles memory starting at adr, or the current adr if not entered. The disassembly code, (the instructions -- not the hex bytes), can be modified using normal screen editing. This gives you a single line assembler process that is a direct link to the syntax processor in the main assembler section. Therefore, it uses the same format, and has all of the same features as any one line of code that you could enter in the assembler section. You can use labels, < and > operators, and even pseudo-ops! You can enter branch instructions with an address like "*+8", which means the current PC +8. The only restriction is that you cannot use a macro call. You can enter 816-only instructions, because the assembler understands them all. However, after you press return, the disassembler will not be able to properly display the code you just typed. So even though it will assemble the code correctly, you won't be able to see it in the debugger. Another problem, is that the disassembler will not know how many bytes were required by a line of 816 code. It will assume that all unknown instructions are one byte, so anytime you enter a multi-byte 816 instruction, the address displayed for the next line will be incorrect. It will be one byte past the line you just entered, which will be in the operand field of the last instruction. For now, it is up to the programmer to be aware of how long the 816 instructions really are, and ensure that the next line is typed at the proper address. Anytime you need to change the address on a disassembly line, you need to move the cursor up and then back down, to reset the Atari screen editor for accepting the entire line of text.

Single line assembly can be started from scratch, (as opposed to editing an existing disassembly), by typing, "-adr ." followed by an Assembly mnemonic. (The '.' is necessary). Such as:

    -600 .LDA #0

Because the period is a marker for the beginning of the instruction field, entering a pseudo-op will require two periods. Such as:

    -600 ..HE 55 AA FF

This gives you additional methods for putting bytes into memory. Since the regular Change Memory command is limited to 8 bytes, you can use the above .HE format when you want to enter more bytes than that. Or use .BY when you want to enter mixed strings of ASCII, HEX, and DECIMAL. Maximum line entry length is always limited to 80 characters though. Other pseudo-ops that can be useful are, .DC for blocks of constant data, and .SB for ATASCII screen code bytes. You can also enter the .24, .AB, .AW, .IB, and .IW pseudo-ops to control the size of the operands that you enter, just as you would need to do in the assembler. None of the other pseudo-ops produce useful results, and some can be hazardous to use.

From within the single line assembler, you may enter '*' as the first mnemonic character to continue disassembly from that address forward.

Display Registers R

Displays 6502 registers in this form:

```
,A  X  Y  NV-BDIZC SP
;AB 5D FA 10110001 FF 7014 LDA #$00
```

Change Registers ; register bytes

Supports screen editing of R command. Status flags can be modified in bit form. When entering values directly, a comma will skip to the next register, and you don't need to enter all the values. EX: ';55' will change the A register to 55. ';,,20' will change Y to 20.

Goto G[S] [adr] [*brkpt] [C by] [r by] [Pf bit]

Run program at adr, or PC if not entered. At any time during execution, the Break key will return to the debugger and display the current registers and PC. Use the 'S' option to run code that ends in an RTS. (Note: When using the S option, the PC adr in the register display on return is an internal address, not the address where the actual RTS occurred.)
A breakpoint will create a return point to the debugger whenever a particular address or condition is reached. *brkpt will place a 00 (BRK) at the breakpoint address. For this reason, breakpoints can not be used for programs in ROM. A '?' will be printed in this case. The breakpoint must also be set at an opcode rather than an operand location so that it will execute. The rest of the parameters add conditions to the breakpoint.

C + by Counts the number of times the breakpoint is reached. Execution continues until the BRK is passed the specified number of times. Breakpoints can also test for specific conditions by specifying (r) reg name and (by) byte it must contain in order to BRK. Processor flags can also be tested by 'P' + flag character + (bit) for condition. Use the flag characters as in the register display.

The breakpoint will be skipped over until the specific condition is reached. When both count and condition options are used, the count will apply to the number of times the condition is met. Execution speed will be slightly slower than real time in this mode. Actual speed will depend on how often the program is interrupted to check conditions.

NOTE: A peculiar bug in the 6502 chip causes breakpoints to be intermittently skipped over. When the BRK interrupt occurs, the program counter+2 is pushed on the stack, but instead of jumping through the interrupt vector, the OS will occasionally just return to the program at PC+2. This is usually a very rare occurrence, but can happen more often when using conditional breakpoints on very small and quick loops, thus BRK interrupts are occurring very rapidly. It took many years before I was able to really understand what was going on, and be assured that the problem was indeed in the 6502, and not a bug in the debugger.

ADDITIONAL NOTE: This bug does not occur on the 65816 processor!

Go command examples.

```
G 2000            = Run program at $2000
G 4000 *4124      = Run at $4000, and break at $4124
G *3100 A'Q       = Run at current PC and break at 3100 when A register
                    equals ASCII 'Q'
G *4200 C10 PZ1 = Run at PC and break at 4200 the 16th time the zero
                    flag is set
```

Remove Breakpt *

Brkpts remove themselves, and replace what was there when the BRK is executed. However, in case the program stops at other than the brkpt, * will remove it. This can occur when the Break key is pressed, conditional or count values are not reached, or when the BRK is set in an operand rather than an opcode. Setting a new brkpt with the G command will also remove an unused BRK.

Exit to DOS X

Return to Assembler A

Fill Memory F adr1 adr2 [by1] [by2 by3...]

Fill memory with 0 if no data bytes. Otherwise enter 1 byte, or a sequence of any number of bytes to fill with.

Transfer Mem T adr1 adr2 adr3

Move memory from adr1 through adr2 to adr3. Handles overlapping moves.

Hunt for chars H adr1 adr2 by1 [by2...][?]

Hunt for String H adr1 adr2 'ASCII string [?]

Hunt memory for ASCII string or string of hex bytes up to length of 30. Use '?' for a wildcard to match anything. Note that the default wild card byte is also $3F hex, meaning that any searches with 3F in a hex string will be treated as a wildcard as well. See the next command for changing the wildcard character in cases of interference. Realize the number entry system will let you search for things like "A9 'A", (as in LDA #'A), but not the reverse of this. Entering "'A A9" will put the hunt into full ASCII form, and search for the literal string that you typed in. The second example can actually be entered in the form "? 'A A9", using a wildcard to avoid the initial ' identifier. For one more example, let's say you wanted to search for a JSR to a MAE defined label. This can be entered as "20 .<LABEL .>LABEL".

Searches through the OS ROM area will automatically skip $D000-$D7FF. So you may simply enter a search range of $C000-FFFF.

Change wild card ? char

Change the wild card for the Hunt command to 'char'. This is used in case a character in the search string needs to be '?' or HEX 3F.

Compare mem K adr1 adr2 adr3

Compare memory from adr1 to adr2 with memory starting at adr3. Displays all addresses with differences.

DEC to HEX # decimal number
HEX to DEC $ by

Displays hex values of decimal numbers and vice versa.

Change Output O [filespec]

Send output to screen and filespec. O by itself returns to just screen output. Default drive processing is not done on this command. Full filespecs must be entered for disk files. Because of this, other devices do not need a ':'. ('O P' is sufficient to send output to printer)

Re-open Editor IOCB E

This is useful for returning to the text screen from a graphics mode, or 80-column display, or to reset the screen after changing RAMTOP. The other IOCB's used are: #3 disk reads, #4 disk writes, and #6 external output.

Change View V

When debugging a program that creates a new display list, the V command can toggle between the program's screen and the debugger text screen. The debugger stores the display list address for the text screen, initially at $BC20, and updates this whenever the E command is used. The V command checks this address against what is currently in the display list pointer, to decide whether it needs to restore the text screen, (saving the previous value), or return to the last saved value of your program's screen. While the debugger doesn't initially know where your program's screen is going to be, it picks this information up the first time you issue the V command with your program's screen active.

In some cases, swapping the display list pointer may not be sufficient to display both screens in their proper format. Such as when using different character sets, or different GPRIOR modes. For this reason, operation of the V function can be extended through the user function, explained later in the user function section.

Query MAE symbol table Q adr
(Sorry, I was running out of letters)

Search the current symbol table for a label that matches the value entered for adr. If found, the label will be printed. This is basically the reverse procedure for symbol table lookup, and as such will only work well when the requested value has only one label associated to it.

Trace Instr I [adr]

Traces program an instruction at a time. Trace normally works on programs in ROM, except 'G' and 'R' options as noted below. After each step, the debugger will wait for one of the following keypresses to control the tracing mode:

Space - Steps one instruction at a time.
C - Continuous trace.
D - Disassemble next instructions. Useful for previewing code that you are about to step through. The program counter will remain at its current location.
G - Execute all instructions up to current 'D' command listing. Use to quickly execute loops or other structures. First use 'D' to find a spot past the structure, then 'G' will execute everything up to that point. This command puts a Breakpoint at the end position, and therefore cannot be used if the program is in ROM.
S - Execute entire subroutine as one step.
R - Return from subroutine. Use this command if you are already in a subroutine, and wish to return to the previous level. A BRK will be placed at the instruction the subroutine returns to, and therefore cannot be used for programs in ROM.
P - Peek the value of the operand of the current instruction. Operand calculation is crude, using simple absolute or direct page addressing on the operand value. It does not attempt to calculate indexed or indirect operand addressing. Thus, if you do a P on an instruction like "STA ($80),Y", it will return the contents of location $80 -- which can theoretically be useful, but the function is intended for use on instructions with simple absolute or direct page addressing modes.
Q - Perform a Q debugger command on the operand of the current instruction. If the operand value is defined in the current MAE symbol table, the label name will be printed. You can use this on JMP, JSR and branch instructions to get an idea where you're going, and also on any memory references that have you thinking, "What the heck is *that*?"
U - Execute the user function. The carry will be clear, and $F0 will contain the PC for the currently displayed instruction.
V - Execute the V command to change DLIST views.
X - Ignore instruction. Skip to the next one without executing.
ESC, RETURN, or BREAK exits trace mode.

Change Default Drv / 1-9

All default drive accesses change to drive number entered, including uses in the assembler section. The starting default drive number will be the same as the current SpartaDOS drive.

Binary Load L [@adr][-adr] filename

Load DOS II binary file where it was saved, or at @adr if entered. Prints a '?' if the file is not DOS II format. Loads appended files, but @adr only works on 1st part. -adr loads raw data with no header using a straight CIO transfer. Both PC and default address are set to the load address.

Binary Save S [-]adr1 adr2 [@adr3] [+]file

Save DOS II binary file from adr1 to adr2. If @adr3 is entered, it will be used as the header allowing the file to load in at a different address than where it was saved. Use '-' for a CIO save without header. If + is entered, append to existing file.

Directory \ [name or spec]
(Sorry, I *am* out of letters)

Displays disk directory. Default of Dn:*.*. A filespec of D2: = D2:*.*

Sector Read R adr sector# [ending sector]

Sector Write W adr sector# [ending sector]

Direct sector I/O to default drive. Reads single and double density disks automatically including single density sectors 1-3 of a DD disk.

Evaluate Exp. = by1(+-*/&!)[by2..] (no spaces)

Prints hex and decimal values of expression evaluated left to right. Can also be used for ASCII convert. (='A)

User Function U [adr1] [adr2] [adr3]

Accepts up to 3 parameters, which will be stored at $F0, $F2, and $F4. $F2 and $F4 will be 0 if not entered, while $F0 will have the current address if not entered. In addition, the carry flag will be set if no parms. Then jumps to the end address of the debugger-3. ($BBFD in top of RAM version) User function expects an RTS return.

When the user function is called from a paused Trace or other display, the carry will be clear, and $F0 will contain the PC or current address respectively.

It was mentioned earlier that the user function can be used to extend the functionality of the V command to switch screen views. Access to the V command is provided by doing a JSR $BBD1 in this version. After the JSR, the carry flag will be clear of the text screen has been made active, or carry set if the program's screen has been made active. Thus you can follow the JSR with a BCC or BCS, and set up any additional locations that are required. For example, let's say your program uses a GTIA screen with a GPRIOR value of $40, and a CHBASE of $80. Here is a user function that will accomplish this:

```
$BFFD  JSR    $BBD1
       BCC    $BC0D
       LDA    #$40
       LDX    #$80
$BC06  STA    $26F
       STX    $2F4
       RTS
$BC0D  LDA    #0
       LDX    #$E0
       BNE    $BC06
```

Once this code has been entered, it can be saved to disk for further use.

- THE DEBUG80 USER FUNCTION -

There is a file on the disk called DEBUG80. This loads into the area for the debugger's user function, and can be loaded either from DOS, or from the debugger with the L command. This function is intended to help debug Atari programs while using the XEP80. As mentioned earlier, the driver for the XEP80 will not run unless the Atari's DMA turned off. DEBUG80 provides a toggle for the DMA control, so that the program screen can be turned on or off as needed.

XEP80 debugging is still very limited, because you cannot issue any commands while the Atari DMA is enabled. Any attempts to do so, will corrupt the XEP80 screen, and probably require turning the power to the XEP80 off and on to recover it. You can only toggle the DMA when the screen display is paused, like from a memory dump or trace mode. Still, you may find applications where it is very helpful to view debugging information and the screen display at the same time, and this extension will let you do this.

- NOTE 64K -

This assembler isn't designed for a 64K machine, but it can be configured to operate in a limited fashion. Because the bulk of the code sits at $4000, that will be right in the middle of RAM if you don't have bank select memory. The best you will be able to do, is set the text buffer from $6400-$A8FF, and the symbol table from wherever your LOMEM is up to $3FFF. That will give you about 17K for your text buffer.

Instructions on how to configure the addresses, and where they are located, are contained in the file ED.DOC.

You will need to use a disk file editor to change the configuration. Skipping 9 bytes from the start of the file, (6 bytes for the binary header plus 3), you will want to change the following bytes to:
00 64 FF A8 00 00 FF 3F
(Using 00 00 for the symbol table start, will use the value in lomem.)

You should also set the Bank select configure byte to $FF, which will cancel the bank selecting.

- HISTORY -
- New since version .93 -

This will be the last version of the assembler that will run in 64K of memory. Because the symbol table takes up half of the bank select RAM area, there is just not enough room to fit the monitor into banked RAM, or add the improvements I would like to do to the editor and assembler modules. Thus, future versions will be using two banks of XE bank select memory, and it will no longer be possible to make do with a 64K machine.

When recording key macros in the editor, you must now use Ctrl-3 to end recording, instead of Esc. This allows Esc menu commands to be entered into macros, primarily to support a chain of assemble commands when your program contains several modules. The next version of the assembler should allow loading and saving macros to disk, which will further enhance the macro usefulness.

Hunt routine in the monitor now automatically skips over the area from $D000-$D7FF. So you can search the OS using $C000-$FFFF and not generate any hardware accesses.

Hunt and Memory display routines would not always stop when the address reached $FFFF. This has been fixed.

I removed the automatic OS routine detection from the trace function. Now, you must use the S key to trace through OS functions in one step, just like any other subroutine. You can also use the R key if you are already within the OS code. The reason for doing this, is that it makes things more consistant, and also allows you to trace code in the $C000-$FFFF area if you need to.

Pseudo-ops are now available in the debugger's single line assembler.

The debugger now includes a built-in function for switching between display lists for the debugging text screen, and your program's screen. It uses the letter "V", for change View. Both V and the "U" user function can be called from both the trace mode, as well as any paused memory or disassembly listing.
The "%" key did not work as a wildcard in the debugger, since it was interpretted as the start of a binary number. I have changed the default wildcard to "?" in both the debugger and editor. This propagated through a few of the debugger command key assignments, along with a few other changes as well. Overall, I feel the key assignments have been improved, and they won't be changed from now on. Here is a sumnmary of the changes:

    ? - Change Wildcard
    = - Evaluate expression
    V - Change display view
    \ - Disk Directory

The editor uses the same wildcard configuration byte as the debugger. You can use the debugger's "?" command, or a Cntl-? in the editor to change the wildcard character. Both modules will use the new assignment.

1200XL function keys are now supported for moving the cursor.

You may enter Ctrl-key graphic symbols or international characters into the editor by pressing Ctrl-A, and then the key you wish to enter.

Now uses an improved method for detecting the default drive when first loaded. This should be compatible with all SpartaDOS versions, and cause no problems for non-Sparta DOSes. It also allows you to specify a different default drive from the command line, such as, "MAE D2:". If a drive is not specified, the default drive will be where the MAE program was loaded from. Note that this is slightly different from the previous behavior. If D1: is the current drive, and you type "D2:MAE", this version will set the default drive to D2:. The previous versions would set the default drive to D1:, being the drive Sparta was logged on to.

The MAE.COM file now comes with a RUNAD address installed. The SpartaDOS bug that prevented using the RUN command to return to a program which used RUNAD has been fixed in 3.2g and later, so I have decided to include RUNAD in the file now.

Fixed a stack corruption problem when disk I/O errors occurred during assembly with a .IN include file.

Improved documentation.

- New since version .92 -

The editor's label search using either the ˆL or ˆJ commands has been made a bit cleaner. Searching for a label "TEST" used to stop if it found a label "TESTING", since "TEST" is a part of it. The search will now find the unique label. Also, the ˆJ JSR function will operate correctly on lines such as "LDA LABEL+1", and will take you to where "LABEL" is defined. Previously, things such as "+1" or ",X" used to confuse the editor about where the label name ended.

An example macro file is now included.

Note that version 1.0 of the asembler is going to use two banks of XE banked memory. I received some good suggestions that I want to implement, plus moving the monitor into banked memory will use up too much space in a single bank, leaving insufficient space for the symbol table. I hope this does not inconvenience anyone, but it is the best way to ensure the largest amount of main system RAM will be available to the user, without making any compromises.

- New since version .9 -

A long standing bug which could trash the Esc menu has been fixed.

The TAB compression could sometimes mess up .BY statements with ASCII strings.

The debugger is now ZP clean, so all of ZP is available for the user.

You can look at RAM under the OS, by setting the bit in $D301, as long as you are using SpartaDOS or some method of handling interrupts when the OS is disabled. Previously, this used to crash the debugger, which uses the E: OS routines for text output.

A faster version of the Hyper_E screen accelerator is included.