MAE A8-bitRI

(John Harris: john@pulsarinteractive.com)
(HTML conversion by SysOp Fox-1, February 2000)

HOME BACK

TABLE OF CONTENTS

BACK TOP

WELCOME to the MAE assembler

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, but not the fastest assembler available. For example, the editor module of MAE is 2850 lines of source, and assembles to a disk file in 18 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 device 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. MAE also includes software drivers that allow 64 column and 80 column editing on a high-speed Gr.8 screen. Thanks goes to Itay Chamiel for his work on these drivers.

I am always interested in hearing comments or suggestions about MAE. You can reach me at:

John Harris
45346 Graceway Dr
Ahwahnee CA 93601
USA
internet: jharris@poboxes.com
or: john@pulsarinteractive.com

Table of Contents

Features of the MAE development system

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. Software drivers for 64 and 80 column screens are also provided.

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 efficient -- it will save you a lot of typing.

True local labels that are referenced between global labels.

Full text substitution macros.

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.

Welcome Table of Contents

Memory Usage and Configuration

This assembler is designed to run in up to 3 banks of the extended memory available on the 130XE or other memory expanded Atari. The bank numbers are configurable. A portion of the assembler must reside outside of bank select RAM, and in this version, it will occupy $B700-$BBFC.

Memory usage for the source file and symbol table is configurable. In addition, they can use bank select RAM for addresses within the $4000-$7FFF range, which keeps this area in main RAM free for the user. 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 will place the symbol table in one bank by itself, and the source buffer from LOMEM to $B6FF, with the segment from $4000-$7FFF in another bank of extended RAM. $4000-$7FFF in main RAM is free to the user, unless running the 64 or 80 column handlers which

use part of this space, from $5600-$7FFF.

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-$7FFF, it is 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 $8000-$B6FF, and the symbol table from LOMEM to $3FFF. That will only give you about 14K for your text buffer, and your symbol table may not always be available for the debugger, but otherwise, all functions should work as described. See INSTALL.DOC for more details.

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.

Buffer memory addresses and several configurable parameters are stored at the beginning of the program file. The parameters are located at +3 bytes from the start address, which will be +9 bytes when counting the DOS binary header. A utility, CONFIG.COM is now provided to edit these parameters. A description of them follows. (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 $B6FF.
Word	- Symbol table start adr. Default $4000, in bank select memory.
Word	- Symbol table end. Default $7EFF. (The code for symbol table search is at $7F00).
Byte	- Assembly CPU mode. 0=65816, $80=6502. Other bits may be used in the future.
Byte	- Editor mode. Three bits are currently defined:
	- When bit 7 is on ($80), Insert mode is active.
	- When bit 6 is on ($40), spaces are converted into Tabs.
	- When bit 5 is on ($20), editor will start with Caps lock on.
	- Other bits may be used in the future.
Byte	- Debugger mode. Currently unused, but reserved for future options.
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 80 column 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.
Byte	- Sets what bank to put the symbol table. Default is $E7. Be aware that there is a small piece of code from $7F00-$7FFF that gets stored in the same bank as the symbol table.
Byte	- Sets what bank to put part of the source text. Default is $EB. If source memory is configured from LOMEM to $B6FF, the segment from $4000-$7FFF will be put in this bank, leaving that region of main RAM free for object code or other uses.
2 Bytes	- An optimistic number of bytes for future expansion.
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.

Welcome Table of Contents

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.

Starting in version 1.2, MAE now displays the amount of free memory in the source and symbol table buffers.

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 $B800, 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.
M	- Macro load and save. The contents of the keyboard macro buffer can be saved to and loaded from disk files. This allows you to create useful key macros and save them to disk for later use.
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 cut assembly time about in half, 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 ^Z, 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.

Welcome Table of Contents

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.

The editor allows line lengths up to 79 columns, and will scroll horizontally for displays that are narrower than this.

Welcome Table of Contents

Editing Commands

Moving Around Find & Replace Block Moves & Copies Key Macros Misc Other Stuff

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.
S^<	Page up by configurable # of lines
S^>	Page Down
^,	Start of line
^.	End of line
^T	Top of file
^B	Bottom of file
^S	Set mark at current location
^M	Go to Mark
^G	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
^F	Find text -- not case sensitive. '?' can be used as a wildcard.
^?	Change the wildcard character. After issuing the command, press any other key to set the wildcard to that character. The wildcard character is shared between the editor and debugger, and changing it within either module will affect uses in both modules.
S^F	Find by searching backwards towards the start of the file.
^R	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.
^N	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.
^P	Find Previous match, by searching backwards.
^L	Enter a label name, and the editor will jump to where the label is defined. (By searching from the first column only.)
^J	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 ^H. 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, ^J will attempt to find the location where that label is defined.
^H	Return to previous position where a ^J command was entered. ^H 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 ^J 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.
^D	If there is a ^Z block mark set, ^D will duplicate the text block. Effectively, it does a ^C followed by a ^V. Because this process clears the ^Z mark, multiple presses of ^D will not produce multiple copies of the same block. Use ^V to paste additional blocks. If there is no ^Z mark, then ^D will duplicate only the current line.
	KEY MACROS
^W	(Write) Begin key macro recording. When done, press ^3. Up to 80 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.
^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.
S^(	If a label exists on the current text line, move it up to the previous line. (lines that already contain labels or comments are skipped over)
S^)	Same as above, but moves the label down.
S^[	Move the current line up one position relative to the lines around it.
S^]	Move the current line down one position.
^;	Comments or uncomments a block of text. You can first mark the start of a text block with ^Z, and then press ^; on the last line of the block. If the block does not start with a commented line, then ";" characters will be added to the front of all lines in the block. If the block is already commented, then the ";"s will be removed. If you do not set a ^Z block mark first, then this command will process only the current line, and move the cursor down. This can be a faster way of commenting just a few lines.

Welcome Table of Contents

ASSEMBLER SECTION

Welcome Table of Contents

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), ^ (exclusive OR), \ (modulo), and unary -. The vertical bar | can be used in place of !.

There are four logical operators that will return values of either 0 (false) or 1 (true). These are <, >, =, and # (not equal). These operators are primarily for conditional assembly .IF statement use.

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

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

Do not put any spaces in the middle of expressions. Spaces are considered by MAE to be separators between different expressions.

Examples of valid expressions:

```
  LDA #'A-$20     ;= $21
  LDA #-1         ;= $FF
  LDA #-1+2       ;= $FD (the unary - is done last)
  LDA #%101&3     ;= 1
  LDA #>$1234+1   ;= $12
  LDA #>$1234+256 ;= $13
  LDA #^$123456   ;= $12
  LDA #>$123456   ;= $34 (mid byte)
  LDA #1>3        ;= 0 (false)
```

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...
```

Assembler Section Table of Contents

Labels

The first character of a label may be any letter, or the symbols @, _, or ?. All remaining characters may also include numbers plus the '.' symbol. 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.

Assembler Section Table of Contents

Addressing Modes

All 6502 and 65816 addressing modes are supported. Any addresses that evaluate less than $100 will use zero-page modes when possible. Zero-page labels must be defined before being used, because when MAE encounters a reference to a label which is not yet defined, it will assume absolute addressing. If that label is later defined to be zero-page, MAE will use zero-page addressing on pass 2, but it won't know that it used absolute addressing on pass 1, and thus the program addresses from this point on will be incorrect.

Beginning with version 1.2, MAE checks for such phase errors by verifying that program labels are at the same PC address on pass 2 as they were on pass 1. If a mismatch is found, a PHASE error will be displayed. Also new in version 1.2, is the .ZP pseudo-op, which pre-defines a label as zero-page type. This can be useful if you have labels defined in multiple modules, and the label needs to be used prior to the module where it is defined.

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 ^SRC ^DEST
```

Use the bank byte of the source and destination addresses.

Assembler Section Table of Contents

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.

.02

Set 6502-only mode. In this mode, all 65816-specific instructions will be flagged as "NOT 6502" errors. The code will still be assembled in these cases, however it will not run properly on 6502 based machines.

.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.

.816

Set 65816 mode, so that non-6502 instructions will not be flagged as errors. The initial setting of the .02 versus .816 assembly mode depends upon which processor version of MAE you are running. The opening menu display shows the current version number, and also an indication of whether it is a 6502 or 65816 version of the program. The initial assembly mode will match this.

.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. This byte will be stored into location $D301 when storing bytes of object code into RAM.

.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, and .CB which creates strings in which the last byte is EOR'ed with $80.

.CA byte

This is to allow for assembly directly into a bank select cartridge environment. The byte is placed in the X register, and a STA $D500,X is performed when object code bytes are stored into memory. The only catch, is that 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.

NOTE: The standard public version of MAE resides partially in the cartridge address space, and as such this pseudo-op will not work properly. Custom versions of MAE that reside in different areas of system RAM, such as just above your LOMEM, can be provided upon request.

.CB [+byte] bytes and/or ASCII

This is in the same format as the .BY pseudo-op, except that the last character on the line will be EOR'ed with $80.

.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.

.EL

Used after a conditional .IF statement, this marks the "ELSE" portion of assembly. See the section on conditional assembly for more details.

.EJ

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

.EN

This is an optional pseudo-op to mark the end of assembly. It can be placed before the end of your source file to prevent a portion of it from being assembled.

.EN can also be used to mark the end of a .IF conditional assembly section, (as in .ENDIF). Because pseudo-ops are only recognized to two characters, the .EN command will perform an ENDIF function when encountered within a conditional assembly section, and will end the assembly otherwise. The "***" ENDIF operator used in pre-1.1 versions of MAE is still supported, and actually preferred since there is no ambiguity here. It is also a little more visible at the source level.

.ES

Display the object code resulting from Macro expansions.

.FL floating point numbers

Stores 6-byte BCD floating point numbers for use with the OS FP ROM routines.

.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.

.IF expression

The expression will be evaluated, and if true, (non-zero), the statements following the .IF, up to a .EL or .EN (or ***) will be assembled. If the expression is false, then the block of statements will not be assembled. See the section on conditional assembly for more details.

.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

.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 longwords

Stores longwords, (3 byte values) in memory.

.LS

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

.MC adr

Move Code to a different address than the .OR assembly origin. If you are assembling to RAM, your code will be stored starting at the address after the .MC pseudo-op. When assembling to disk, the .MC address will be used when creating the binary file headers, affecting where the code will be loaded into.

!!!name **.MD**

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 should run some type of strip program to de-segment the file for optimal size and speed. 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 words

Stores words in memory. Multiple words can be entered.

label **.ZP**

Pre-defines a label as zero-page type. This can be useful if you have labels defined in multiple modules, and the label needs to be used prior to the module where it is defined. Normally, this would create a phase error during assembly whenever MAE encounters a forward label reference (where it will assume absolute addressing)

during assembly whenever MAE encounters a forward label reference (where it will assume absolute addressing) which later turns out to be zero page. The error can be prevented by specifying such labels as .ZP in the first module. Note that .ZP usage must precede the actual label definition.

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.

Assembler Section Table of Contents

Conditional Assembly

Conditional assembly allows the programmer to adapt the assembly process to different conditions. Blocks of code can be included or skipped over based upon the value of an expression. The format of conditional assembly is:

```
  .IF expression
        ;This block of code is assembled if the
        ;expression is true.
  .EL
        ;Else, this block of code gets assembled
        ;(when the expression is false)
  .EN   ;Marks the end of the conditional block.
```

The operand of the .IF instruction will be evaluated, and if the expression is true, then the source code following the .IF will be assembled until reaching a .EL or .EN pseudo-op. (Once again, two-letter pseudo-ops are a convenience, not a requirement. You are free to use the more standard .ELSE and .ENDIF if you prefer). The .EL portion is optional, and is used when you want one block of code to be assembled when the condition is true, and a different block when the condition is false. The end of the conditional is marked with either .EN or three asterisks ***. '***' is equivalent to an ENDIF statement, and is somewhat preferable since .EN will be interpreted as 'end of assembly' when it occurs outside of a valid conditional assembly block. '***' is non-ambiguous and will flag an assembly error if it does not have a matching .IF.

Examples of Conditional Assembly:

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

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

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

  .IF FLAG1^FLAG2
  .               ;asm'ed if FLAG1
  .               ;or FLAG2 <> 0,
  .               ;but not both
  ***

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

  .IF WIDTH = 40  ;This is INVALID.
                  ;Do not put spaces in expressions.
                  ;Spaces separate expressions from
                  ;each other.

  .IF WIDTH#40
  .               ;This gets asm'ed when width <> 40
  ***

  .IF WIDTH<40
  .               ;asm'ed if WIDTH less than 40
  .EL
  .               ;asm'ed if WIDTH greater or equal to 40
```

```
***

.IF WIDTH>40
.               ;if WIDTH greater than 40
***
```

Assembler Section Table of Contents

Macros

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

```
!!!name .MD
        ; body of the macro
        .ME
```

Where 'name' is the name of the macro. The three exclamation marks are a special macro identifier, and must precede the macro name. The body of the macro definition will follow, and should be ended with a .ME pseudo-op.

The macro definition must be resident in memory when it is called. If you link multiple source files with .INclude pseudo-ops , then you need to ensure that any macro definitions are forced to be memory resident by using .MG within the file that contains the macros. Typically, you can put all your definitions in one file, put in the .MG option, and then include it at the beginning of your assembly. The root source file, that is, the one that is in memory when you issue the Esc-A assemble command, is always memory resident anyway and thus macros defined in your root source file are always available to other included files without the need for .MG.

Beginning with MAE version 1.1, the assembler now has free-format and full text substitution macros. Macro parameters can be anything, and will be passed to the macro routine in their original text form. The number of parameters passed by the macro call is not rigidly enforced, and in fact the macro definition no longer has to specify the number of expected parameters. Within the body of the macro, parameters are accessed by using a ':' followed by a number from 1 to 9, corresponding to the order of parameters on the calling line. (Parameters are separated by spaces, and nothing else). A special macro parameter, ':0' can be used to get the actual number of parameters passed in. When a macro is expanded, any ':n' strings that are not within quotes will be replaced with the text from the calling line. Text within quotes will normally be left as-is, which means there needs to be a special method of getting a macro parameter expanded inside of quote marks. Two double quotes in a row, "", will be replaced with one double quote, and subsequent macro parameters will be expanded. Then use another set of two double quotes to close. See the macro examples below for more details on how this works.

An individual macro may pass up to 9 parameters, but there is also a limit on the total number of parameters including all nesting levels. This limit is 16. If a macro uses 8 parameters, then any nested macros it calls can use at most 8 additional 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
       INC :1
       BNE ...SKP
       INC 1+:1
...SKP .ME
```

To call this macro, you would use:

```
         IND $80
```

Since a macro call will pass any text characters, you could call the same macro with:

```
         IND $80,X
```

Note that the structure of the second INC instruction in the macro body is important here for this to work correctly. If the line were written "INC :1+1", it would get expanded to "INC $80,X+1" which is not valid.

Here are more examples that don't do anything specific code-wise, but serve to demonstrate various macro techniques. The calling line will be listed first, followed by the definition and what it will actually be expanded to.

```
       PRT "HELLO" $9B "THERE" $9B

!!!PRT .MD
       JSR PRINT
                  ;JSR PRINT
       .BY :1 :2 :3 :4 :5 :6 :7 :8 0
                  ;.BY "HELLO" $9B "THERE" $9B 0
       .ME
```

Parameters that are not defined on the calling line are simply replaced with null strings. They do got generate errors. The PRINT subroutine in this example would pull the return address off the stack, display the string that it points to until 0 is reached, and then push that address back on the stack so program flow continues with the next line of code.

```
       TST $80

!!!TST .MD
       .BY :1       .BY $80   ;direct substitution
       .BY ":1"     .BY ":1"  ;strings inside quotes
                              ;are not expanded
       .BY """:1""  .BY "$80" ;two quotes get converted
                              ;to one, and the
                              ;parameter gets expanded
                              ;since it is not
                              ;between quote marks.
```

Here's a more complicated example that can be used as a debugging aide during development.

```
      ASSERT INDEX CC #$80

!!!ASSERT .MD
      .IF DEBUG
      PHP          PHP
      PHA          PHA
      LDA :1       LDA INDEX
      CMP :3       CMP #$80
      B:2 ...OK    BCC ...OK
      JSR PRINT    JSR PRINT
      .BY "Assert Failed: " "":1 :2 :3"" 0
                   .BY "Assert Failed: " "INDEX CC $80" 0
...OK PLA
      PLP
      ***
```

The idea behind the ASSERT macro, is that it can be used to verify the value of key variables, notify the programmer when the value is not in range, and all the code disappears when you assemble the final version simply by setting the DEBUG flag to 0.

There are more macro examples in the supplied include file MACROS. If you create some really useful macros, please send them to me and let me know if they can be included in future MAE distributions.

Assembler Section Table of Contents

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 ^H 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 ^G 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

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
Nesting error. .MD macro definitions cannot contain additional definitions. .IN included files may not include additional .IN files. Endif "***" mark without associated .IF statement.

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.

NOT 6502
This instruction is only valid on 65816 processors, and will not run on 6502-based computers. This error is only generated if you have used the .02 pseudo-op to generate 6502-only code.

PHASE
Zero-page variables must be defined, or pre-defined with .ZP, before they are used.

Assembler Section Table of Contents

DEBUGGER

Assembler Section Table of Contents

General Information

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.

Debugger Table of Contents

Commands

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

Displays 8 bytes of hex and ASCII when using M, or 16 bytes of ASCII only when using MM. Displays 3 lines worth if you enter only one address, otherwise it will display up to the second address, if entered. Displays from current address if no parameters. '/' = 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.

The hex bytes in the hex and ASCII display (but not the ASCII bytes), or the ASCII bytes in the ASCII-only display, may be changed using standard screen editing.

When displaying hex and ASCII from a 24-bit address, the last two bytes of ASCII will not be displayed, due to screen width limitations.

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 [M][X][R] [adr]

Disassembles memory starting at adr, or the current adr if not entered. When disassembling 65816 code, instructions that change the register sizes will automatically be detected, and adjust the immediate operands in the listing accordingly. When beginning a disassembly however, it will not know the current state of the register sizes and will default to 8 bits. The M, X, and R options in the command line will force 16-bit M, X, and both Registers to be used at the starting address. The single instruction that gets disassembled as

part of the register display or trace mode will always be correct, since the register sizes can be obtained directly from the processor status register.

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.

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
```

The 65816 version of MAE displays registers in this form:

```
,NVMXDIZCE 0000 00 00
;00AB 005D 00FA FF 7014 LDA #$00
```

Status flags will be inverse when they are set, and normal when clear. The remaining numbers on the top line are the Direct Page register, the DBR, and the PBR. The upper byte of the stack pointer is not displayed. 16-bit numbers for registers A, X, and Y are displayed on the next line, however only the A register will currently show the correct 16-bit value. Full 16-bit support for X and Y will be provided when there is an OS upgrade readily available to handle the native mode interrupts.

Note: The Direct Page register and DBR are currently inactive, and will always display 0. The PBR will display the proper value, but can not be changed by editing the register display. Currently, it can only be set by entering a 24-bit address into the G or I debugger commands .

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.

When setting flags in the 65816 version, you can enter either normal or inverse flag characters, or enter 0's or 1's, and can freely mix the two.

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. I eventually found written documentation of the problem from other sources.

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 [char]

When no additional characters are entered, the BRK vector at $206 will be restored to whatever it was when MAE was started. If you would like to keep the BRK vector trapped by the debugger, you may enter any character after the X. (I could not come up with a decisive and memorable letter to use for this purpose, so I leave it to you to choose your own.)

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. Use "O P:" to send output to the 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 debugging 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 two user accessible vectors at $BFF7 and $BFFA. $BFFA will be called when switching to your program's screen, and $BFF7 will be called when switching back to the debugger's screen. You can insert JMP intructions here that point to extra routines that update whichever other hardware locations are needed.

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.
L	- Like the return from subroutine, but does an RTL return from 24-bit subroutine calls.
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 24-bit parameters, which will be stored at $F0, $F7, and $F3. $F7 and $F3 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.

Debugger Table of Contents

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.

Debugger Table of Contents

Install Notes for the MAE assembler

MAE.COM as supplied in the archive will run as-is on a 128K+ machine, in standard 40 column mode. Instructions for creating a version for 64K machines and/or 64/80 column versions follow.

Please backup the original MAE.COM file or keep the archive itself before making any customized versions.

Some of these installations require changing the buffer addresses for MAE's source text and symbol table. For these cases you should run the config utility CONFIG.COM. It will try to open the file D:MAE.COM, and will save any configuration changes back to the MAE executable. Thus, you should make sure MAE.COM is on the current working drive, (or on drive 1, depending upon the DOS you are using), before running CONFIG.COM. An error message will be displayed if it fails to load MAE.COM.

Debugger Table of Contents

CONFIG.COM Operation

Once the loading is successful, CONFIG will display a screen full of editable parameters. You may use up and down arrows to move the cursor to different fields, and then type in your new values. Return will advance to the next field. Backspace and left/right arrows are supported within the field.

Press 'Esc' when you are finished with your changes. If no errors are found in the data, CONFIG will prompt if you wish to save the changes back to the MAE.COM file. Press 'Y' to save them, 'Esc' to keep editing, or any other key to exit the program without saving.

If an error is detected, press 'Esc' to revert to the previous value at the start of the session, or any other key to keep editing. The cursor will be placed on the field that had the error.

Please refer to MAE.DOC for details on the parameters themselves, in the section titled "MEMORY USAGE AND CONFIGURATION".

Install Notes Table of Contents

64K Version

The bulk of the MAE executable resides in the $4000-$7FFF address range, designed to be in bank select memory. If you do not have banked memory, you must reconfigure MAE's buffer addresses to not overwrite the program. Free memory will be in two segments, from LOMEM-$3FFF, and from $8000-$B6FF. You should probably set the source buffer to the latter, since it is the larger segment. Use CONFIG.COM to set the first four addresses to $8000, $B6FF, $0000 (MAE will use the current LOMEM), and $3FFF. You should also change the three bank select bytes to $FF, and then save your changes.

Install Notes Table of Contents

64 & 80 Column Drivers

64 and 80 column modes are not available on 64K machines.

Default 64 and 80 column drivers are now included as COL64.OBJ and COL80.OBJ. These use the ANTIC bit compatible handlers, and the 'A' fonts. If you don't have a compatible RAM upgrade, or wish to use the other font choice, please follow the directions below to create new versions of COL64.OBJ and COL80.OBJ.

HAND64.OBJ and HAND80.OBJ are drivers for 64 and 80 column screens. They take advantage of ANTIC memory banking so that their use does not reduce the available memory for source and symbol table. They reside in the base memory range of $5600-$7FFF. If you have a memory upgrade that is not ANTIC-bit compatible, then you must use HAND64X.OBJ and HAND80X.OBJ. These versions will have to place the screen and dislay list from $9500-$B6FF. Code and data will be $6FF0-$79FF. To run the X versions of the handlers, you will need to reconfigure MAE's text buffer to end at $94FF instead of $B6FF using CONFIG.COM.

Copy either HAND64.OBJ or HAND64X.OBJ to a new file named COL64.OBJ, and copy HAND80.OBJ or HAND80X.OBJ to COL80.OBJ. You now need to add character sets to these files.

There are two character set choices for both the 64 and 80 column handlers. A character set must be appended to the end of the handler's .OBJ file before the handler can be used. These files are COL64A.FNT, COL64B.FNT, COL80A.FNT, AND COL80B.FNT. These files should be viewable and editable in standard font editing packages. Choose a font file for each handler, and copy-with-append the fonts to the COL64.OBJ and COL80.OBJ files that you created above. The handlers are now ready for use.

Install Notes Table of Contents

Using the 64/80 Column Drivers

The 64 and 80 column screen handlers can be used in two different ways. The files can be loaded from the debugger, and then when you reenter the assembler the new handler will be active. To do this, press 'B' from MAE's main menu and then use a debugger command like "L COL64.OBJ". Then enter 'A' to enter the assembler again and you should see a 64 column screen. The file "COL40.OBJ" can be loaded to return to 40 column mode. Another option, especially if you find that you want to use an extended column mode most of the time, is simply to append one of the COL64.OBJ or COL80.OBJ files to the end of the MAE.COM executable. In this way, MAE will always start up in the mode you choose, and you are still free to load different modes using the debugger. Personally, I think the 64 column mode works very well for Assembly source, while maintaining much better readability than the 80 column mode.

Note that the debugger will always run on the 40 column screen regardless of which handler is loaded for the assembler/editor.

Install Notes Table of Contents

HISTORY

Install Notes Table of Contents

NEW FOR VERSION 1.2

Editor menu now displays amount of free space in the source and symbol table buffers.

MAE now checks for phase errors caused by referencing zero-page labels before they are defined. It verifies that program labels are at the same PC address on pass 2 as they were on pass 1. If a mismatch is found, a PHASE error will be displayed.

New pseudo-op .ZP, to pre-define labels as zero-page type. This can be useful if you have labels defined in multiple modules, and such labels need to be used prior to the module where the are defined. Note that .ZP usage MUST precede the actual label definition.

Default 64 and 80 column drivers are included, so you don't have to build them if you have an ANTIC-bit compatible RAM upgrade, and prefer the "A" versions of the fonts.

History Table of Contents

NEW FOR VERSION 1.1

Cybergate, my ISP, discontinued their service in my area. Thus, I have another new EMail address: jharris@poboxes.com

New distribution method and installation procedure. Please read INSTALL.DOC for details.

Configuration utility CONFIG.COM now provided.

MAE now has full text substitution macros. This adds a tremendous amount of power and flexibility to the macro processor. The syntax is completely different, so please see the macro section of ASM.DOC.

The ':' character can no longer be used in label names, due to conflicts with the new macro processing. The ';' character can't be used either, but was mistakenly listed as a valid label character in the docs. This hasn't been usable as a label character for many years.

Conditional assembly has been changed to be more standard, and more complete. IFE, IFN, IFP and IFM have been removed and replaced with .IF, .ELSE, and .ENDIF. Complementing the .IF statement are four new expression operators, <, >, =, and # (not equal). <, >, and = join the list of symbols that are no longer allowed as part of a label name. I am sorry for any inconvenience or confusion these changes may be causing, but I suppose their use was non-standard to begin with. Lastly, .IF statements can now be nested up to seven levels deep. Please see the conditional assembly section of ASM.DOC for full details.

Editor key macros can now be saved to and loaded from disk files.

New config bit for editor to start with Caps lock on or off.

Improved screen handling to do minimal text redraws when cutting and pasting. Improves screen redraw speed, most noticably on XEP80

and software 64/80 column modes .

History Table of Contents

NEW FOR VERSION 1.0

I have a new EMail address: jharris@cybergate.com

40 column limit on source code lines has been removed, although text is still limited to 79 columns. Lines will scroll horizontally to display the extra columns.

There are two software screen drivers provided, that allow 64 column and 80 column editing on a high-speed Gr.8 screen. If your machine has Antic-compatible bank select RAM, using the drivers will not decrease the size of your source text buffer! The drivers were written by Itay Chamiel. Thanks Itay! Installation of these software drivers, as well as support for 64K machines , is described in the file INSTALL.DOC .

Disassembly supports 16-bit operands. See the D command in DEBUG.DOC for details.

24-bit address support in debugger, including hex and decimal values for the "=" math function.

Because of 24-bit address support, I had to move the locations of address variables. This affects the User function parameters, and I apologize for any inconveniences. The parameters that used to be at $F0, $F2, and $F4 are now at, $F0, $F7, and $F3. I know the order is unusual, but there are internal reasons.

Debugger memory display command changed somewhat due to screen width limitations with 24-bit addresses. New ASCII-only mode supported. See DEBUG.DOC for details.

While tracing, a new option L has been added to RTL from 24-bit subroutine calls. (65816 version of MAE only).

New assembler pseudo-ops .CByte for making ASCII strings with the most significant bit set on the last character. Also .FLoat for defining constants in the OS floating point format.

New editor command Ctrl-; can be used to comment or uncomment a block of text by adding or removing ";" characters at the start of each line. Mark the start of the block with Ctrl-Z, and then press Ctrl-; at the end of the block. You can also mark one line at a time by using Ctrl-; without a Ctrl-Z block mark.

Editor command Ctrl-D, which used to just duplicate a single line, can now be used to duplicate an entire block if there is a Ctrl-Z block mark set. This makes it consistant with the operation of Ctrl-;.

Shifted 1200XL function keys should now work for moving to the beginning or end of lines, or the beginning or end of the file.

Cursor column position is retained while scrolling in the editor.

MAE was not fully ZP clean, but should be now.

The debugger will use High-speed SpartaDOS SIO routines, if present. Unfortunately, sector reading and writing will no longer work on the old 400/800 operating system as a result, unless you are using SpartaDOS.

In the editor, Ctrl-N did not work properly when pressed on a blank line.

Conditional assembly could get messed up when source code contained a label on a line by itself.

.BI pseudo-op was broken in version .99.

New version of Hyper E: included, which fixes an incompatibility with TextPro, and adds support for the SDX CON: device.

History Table of Contents

NEW FOR VERSION 0.99

65816 opcodes and tracing are now supported in the debugger. There is still no support for 24-bit address entry, so technically, the debugger can be considered 65802-compatible.

There are new pseudo-ops in the assembler. ".02" can be used when you need to assemble 6502-only code. When this opcode is active, all 65816 specific instructions will be flagged with a "NOT 6502" error. ".816" selects the 65816 assembly mode. The initial assembly mode is set to whichever processor version of the MAE assembler you are using. The initial version sign-on message shows the processor version of MAE, which will also be the default assembly mode.

There has been a significant increase in assembly speed. MAE will be about twice as fast, depending on the size of your source files. Small files will show less of an improvement, whereas larger files will show an even bigger difference. Assembly time is closer to a linear relationship to source file size, whereas it used to be somewhat exponential.

There have been big changes in memory configuration, resulting in twice as much symbol table space, a little more source space, and all

There have been big changes in memory configuration, resulting in twice as much symbol table space, a little more source space, and all of main memory from $4000-$7FFF free to the user. The region from $400-$5FF is no longer used by MAE. The debugger has been moved to bank select memory now, reducing the main memory usage to $B700-$BBFC. In its place however, MAE uses up to three banks of extended memory for optimal configuration. Due to these and other changes, the format of the memory configuration bytes at the start of the file has changed. Consult MAE.DOC for full details. MAE actually still runs in 64K machines, and will continue to do so, but the amount of RAM available for source text has dropped from 17K down to only 14K for 64K'ers.

Two new operators have been added to the assembler and debugger expression evaluators. You can now use "^" for exclusive or, and "\" for modulo. Because of this, these characters can no longer be used in label names.

There are two new editor functions . You can move an individual line of text up or down with respect to surrounding lines. Press Shift-Ctrl-[to move a line up, and Shift-Ctrl-] to move it down. In a similar function, you can move a label by itself up or down to adjacent lines. Press Shift-Ctrl-(to move the label up, or Shift-Ctrl-) to move the label down. Lines that start with comments or other labels will automatically be skipped.

Full SpartaDOS directory listings are now supported.

The "*" at the start of a marked text block would not get erased on lines that did not begin with a label. Also, the location of the block start was not getting updated when surrounding text was edited.

There was a bug which prevented macros from being recognized when they were defined after a .IN included files.

JVC and JVS macros were missing from the example MACROS file.

There was a bug that could sometimes clear your entire source text if you used the Esc-V menu option to get the value of a label which was undefined. This bug only occurred after an assembly aborted with one of a few fatal error types.

I didn't realize that MyDOS could use ':' characters as subdirectory path separators. This confused MAE's ':' search routine to determine if full filespecs (or just file names) were being entered. The effect was that MAE would not load files properly from subdirectories unless you used '>' for path separators. It works better now, unless you have subdirectory names that are one or two characters long. If you had a directory named "T", and tried to load "T:FILE", it would think you were referring to a T: device. Thus, it is recommended that you either always use the '>' character as a path separator, or enter complete filespecs. Either "T>FILE" or "D2:T:FILE" will work fine.

Fixed two problems with using "." to get the value of defined labels from the debugger. Because control was passed to the assembler's expression evaluator, the default number base became decimal instead of hex. Thus, if you entered ".LABEL+10" in the debugger, you would get the value at LABEL+$A, and not LABEL+$10. Processing is now returned to the debugger once the label has been decoded, making the rest of the line behave consistantly in regards to hex numbers. Also, expressions like ".>LABEL" were returning the wrong value.

The debugger command V has been changed, and is easier to add user extensions into. Please consult DEBUG.DOC for details.

The editor could corrupt bookmarks and ^J marks if they were within a text block that got deleted, and close to the top of the file.

MAE was not restoring the BRK IRQ vector when it exited, and it now does this by default. Because it is sometimes desirable to leave the BRK vector installed, such as for trapping BRKs in programs called from DOS, you can follow the X command in the debugger with any other character to exit with BRK trapping still active. Note that you must be careful to not overwrite any part of the MAE program, including bank select RAM, if you want MAE to successfully trap BRK instructions that occur after you leave the assembler.

The debugger should no longer lock up if it encounters a BRK when output is redirected to an SIO device. Also, the inconsistancy with supplying filespecs in the O command has been removed. All filespecs are now treated the same way, for supplying Dn: in front of any input that does not in itself contain a ":" character. To send output to the printer, you should now enter "O P:".

History Table of Contents

NEW FOR VERSION 0.95

Conditional breakpoints were not working in the .95 version.

The 65816 version of MAE has a new register display that shows 16-bit registers for A, X, and Y. Processor status bits are now displayed as normal characters for bits that are off, and inverse characters for bits that are on. When changing bits, either normal/inverse, or 0's & 1's may be typed, and are freely mixable. The Emulation Mode shadow bit is also displayed, and can be changed. Make sure you have native mode interrupt handlers available before changing this bit. Also note that this bit, just like the rest of the status register display, only affects the state of programs run or traced from the debugger. Clearing the E bit will not instantly put the machine into native mode, but native mode will be set as soon as any user programs are run or traced.

In the memory configuration bytes, entering 0 for the text buffer start or end would use the value from LOMEM or MEMTOP respectively. Now, this ability also applies to the symbol table addresses.

1200XL function keys were not working, and should be fixed now.

It was pointed out to me that memory expansions above 128K use the high bit of $D301, which is a problem for the way I programmed the .BA pseudo-op. Thus, .BA now stores the entire byte at $D301, and a new pseudo-op, .CA, has been added for bank select cartridge support. I also realized that support for bank select carts is worthless when the assembler resides in the cartridge address space. My personal version is located at my LOMEM, and so I didn't realize the problem here. If anyone wants a custom version of the assembler located at a different address, please let me know and I will be happy to provide it for you.

The .WO and .LO pseudo-ops support multiple addresses now.

There are misc. small cleanups, such as .EN is no longer required at the end of the source text, "()" characters are no longer necessary

There are misc. small cleanups, such as .EN is no longer required at the end of the source text, "()" characters are no longer necessary for enclosing parameters of a macro definition, and other cosmetic changes.

Disregard the earlier note about default drive detection being different in the .95 version. MAE still detects Sparta's default drive correctly, even if MAE is started from a different drive.

History Table of Contents

NEW FOR VERSION 0.93

When assembling code to disk, the .MC pseudo-op can now be used to make the object code load at a different address than where it is assembled, much like the way the function already worked in RAM.

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:".

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.

History Table of Contents

NEW FOR VERSION 0.92

When dinosaurs ruled the Earth. History has been removed to save space. They didn't even have computers back then, did they?

History Table of Contents