# AMAC - The Atari-Macro-Assembler

**Contents**



---

**Introduction:**

AMAC came out in 1981, I believe. It wasn't very successful because it ran headlong into the famous MAC-65 cartridge. AMAC had a number of disadvantages. It was disk based (as opposed to keeping source code in memory), which tended to make it very slow on big files. It didn't have an integrated environment (editor was separate). It had a bug in its macro facility which caused it to blow up if acros over a certain length were written. (I wrote Atari about this bug, but they didn't respond. My guess is that they decided not to support AMAC since it was getting creamed by MAC-65 anyway.) It also had horrible copy protection of the type that causes the disk to grind and reset several times on each loading. Since programming sessions often include multiple toggles between assembler and editor, this was intolerable.

Why would I now recommend AMAC despite these flaws? Simply because most of them can be eliminated, making it a fine assembler. The copy protection was easy to remove. The macro bug wasn't a big deal after I found that only long macros cause the problem. (Most macros are small, anyway.) The lack of a built in editor now looks like an advantage. AMAC will work with plain text source files from any editor. Integrated environment type cartridges like BASIC and MAC-65 now seem hokey with their forced line numbers and the requirement to hit Return on each edited line, etc. The speed problem went away with the advent of things like ramdisks, SIO2PC, and Darek's Xformer. AMAC doesn't flinch at large files. I now like to edit on a PC plain text editor, and use Xformer to run AMAC on the PC, accessing the file directly (after converting CR/LFs to ATASCII EOLs).

I've pointed out AMAC's flaws and how they are overcome. Outside of the flaws I mentioned, AMAC is a great 6502 assembler: robust, flexible, and full featured.

---

**Using AMAC:**

The syntax used by AMAC is pretty standard. Knowing the command line functions is critical.

First, run AMAC. Then, you're prompted for a source file and options. The first entry must be the filespec you want to assemble. The other options are:

| | |
|---|---|
| H=Dn | put output file on disk #n |
| H=<filespec> | put output file to spec given |
| H=0 | don't generate object code |
| L=P | listing to printer |
| L=Dn | listing to disk #n, (name.PRN is filename) |
| L=S | listing to screen |
| L=0 | default (no listing) (this can speed assembly) |

| | |
|---|---|
| O=n | make run address = n (same as END n statement) |
| S=<filespec> | specify systext file |
| S | use default systext D:SYSTEXT.AST |
| S=0 | specify no systext file for this assembly |
| R=F | full reference map |
| R=S | short reference map |
| R=0 | no reference map |

Here's a sample command line to put AMAC into operation:

D:KEYER.SOR S=D2:SYSTEXT L=S:

Assembles my file KEYER.SOR. Looks for a systext file on D2:. Puts a listing to the screen. The object file will be KEYER.OBJ by default and will go to D:. Break stops assembly; CTRL-1 pauses assembly if listing to screen is in effect. I'm not going to try to list everything in AMAC's syntax; just stuff I use a lot.
Remember, a label (if any) starts in the first column; then tab or spaces; then an opcode or pseudo op; then data; then a semicolon starts a comment field if used.

Labels end in a space. Ending in a colon is acceptable but not required and the colon isn't part of the label when you reference it.

Labels can be made from UC or LC letters, numbers, and the symbol @. A colon starts a local label. The underscore character can appear in a label for readability but isn't considered part of the label. So AB_CD is the same label as ABCD. Only the first 6 characters of a label are significant.

Here are some examples:

```
STORE: DS 1 ;define one byte of memory storage
EOL: DB $9B ;define a variable called EOL
START: LDA #2 ;load accumulator with the number 2.
```

Note that the # meant load A with the immediate value 2. Without the #, the same line would load A with the value stored at memory location $0002.

Confusing memory labels with immediate values is one of the most vexing problems for beginners.

Usually, memory is referenced by a name (label) rather than absolute value.

STA STORE ;store accumulator at memory location called STORE.
LDX EOL ;load X with number stored at EOL
END START

An END statement must appear at the end of the file. If you follow it with an address (here, the label START), the address will be the run address of the program created.

Here's an example of a macro:

```
; This MACRO puts a record. The
; parameters are IOCB # and
; address of RECORD.
;
PUTREC: MACRO I, A
LDX #%1*$10
LDA #0
STA ICBLH,X
LDA #$80
```

```
STA ICBLL,X
LDA #9
STA ICCOM,X
LDA #LOW %2
STA ICBAL,X
LDA #HIGH %2
STA ICBAH,X
JSR CIOV
ENDM
```

The I, A arguments are dummies; you refer to them by %1 and %2 in the macro.

The ENDM statement ends the macro.

You must precede it with an ATASCII tab.

If your editor uses multiple spaces for tabs, hit ESC-TAB on your atari to put in the right pointing triangle TAB symbol.

I'd use this macro like this:

LABEL: PUTREC 0, TEXT

0 is for IOCB 0 (the screen) and TEXT would be a label for the address where I'd stored a string, like this:

TEXT: DB `whats going on???',$9b

As you can see, the macro has been used to make a kind of simple PRINT statement.

You will usually start your file with a bunch of `equates' which are labels for addresses or numbers you want to refer to by name. (Or you might put these in a "systext" file.) Here's an example of what might go in one:

```
WSYNC EQU $D40A
COLPF1 EQU $D017
COLPF2 EQU $D018
COLBK EQU $D01A
COLOR1 EQU $2C5
COLOR2 EQU $2C6
COLOR4 EQU $2C8 ; Background
NMIEN EQU $D40E ; b7/DLI b8/VBI
VDSLST EQU $200 ; NMI OS vector
DOSVEC EQU $000A ;this program's start vector
DOSINI EQU $000C ;this program's init vector
POKMSK EQU $0010 ;mask for POKEY IRQ enable
RTCLOK EQU $0012 ;60 Hz clock
VIMIRQ EQU $0216 ;immediate IRQ vector
SSKCTL EQU $0232 ;serial port control
MEMLO EQU $02E7 ;start of user memory
CH EQU $02FC ;character buffer
ICCOM EQU $0342 ;CIO command block
```

Note that you can use the EQU pseudo-op or = interchangeably, as in:

ICCOM = $0342

Note that these equates don't actually put anything into memory or into the object file. They just associate a number with a symbol, so you can refer to it by symbol name in your program.

Here's some more example source code:

```
; Clear the message buffer area:
LDA #0; Set up FILLA routine to
STA ZP1L; clear BMESS thru +FFF
LDA #HIGH BMESS; to all 0's.
STA ZP1L+1
LDA #$FF; Low byte is abs $FF
STA SIZEB
LDA #HIGH [BMESS + $FFF]
STA SIZEB+1
LDA #0
JSR FILLA

JSR RINIT; `Real' Initialization
; Init ZBUF to point to TBUF:
LDA #LOW TBUF
STA ZBUF
LDA #HIGH TBUF
STA ZBUF+1

; Point vertical blank to my custom
; VBI routine:
DOMYV LDX #HIGH MYVBI
LDY #LOW MYVBI
LDA #6; Immediate VBI
JSR $E45C
```

You can see some operators here. HIGH and LOW extract the high and low bytes of a word, for example. (Addresses are stored as words.) Note also that expressions can include math, such as ZBUF+1, which refers to address ZBUF plus 1. Here are some operators AMAC uses:

| | |
|---|---|
| +, -, /, * ; | add, subtract, divide, multiply |
| NOT | bit by bit complement |
| AND | logical and |
| & | same as AND |
| OR | logical OR |
| XOR | logical exclusive OR |
| = | logical equal, also EQ |
| <> | not equal, also NE |
| < | less than, also LT |
| > | greater than, also GT |
| <= | less than or equal, also LE |
| >= | greater or equal, also GE |
| SHR | right shift n bits |
| SHL | left shift n bits |

Brackets [] (not parenthesis) set precedence levels.

As you'd expect, there are IF and ENDIF statements to use the logical tests and conditionally skip sections of code.

There's a PROC ... EPROC feature to allow defining procedures, which just means local symbol ranges. Local labels must start with a colon. I use it because on big files it gets hard to think of unique labels for jump targets, etc, after you've used a zillion of them. Within a PROC you can assign local labels accessible only within the PROC and reusuable elsewhere. Like this:

```
; Copy the character set down to
; $4000:
```

```
PROC
LDA #$40
STA :MVCH+5; Init self altering
LDA #0; code.
STA :MVCH+4
STA :MVCH+1
LDA #$E0
STA :MVCH+2
:LOOP LDY #0
:MVCH LDA $E000,Y; Get char from ROM
STA $4000,Y; put it into RAM
INY
BNE :MVCH; Done with a page?
INC :MVCH+2; Increment pages, ROM
INC :MVCH+5; and RAM
LDA :MVCH+2; Done all 4 pages?
CMP #$E4
BNE :MVCH; If not, loop...
LDA #$40; Point CHBAS to new base:
STA $2F4
STA $D409; Hardware CHBASE

; Now make return and carat print
; as spaces:

LDY #7
:LP2 LDA #0
STA $41F0,Y; $41F0 starts ^
LDA #$FF; Return prints inverse
STA $42D8,Y; $42D8 starts ret.
DEY
BPL :LP2
EPROC
```

The ORG statement is used to tell the assembler where in memory to put the code it generates. Therefore, there's got to be an ORG statement in the program before any code or data statements. You can also put in additional ORGs if you need to make the code discontinuous.

Also, the star * is the current value of the location counter: it's the value of the next memory address the assembler is going to use. See the ORG in the program starting below:

```
; KEYER

; An electronic keyer program by
; Nick Kennedy. Begun: 5/17/87.

; r1.1: I moved the ORG up to $8000 to
; get out of the EMDE expansion window
; because MYVBI and the new DLIST have
; NMI's which may occur during RAMDISK
; I/O, causing lockup. Could just get
; the VBI AND DLIST out if space were
; a problem.

; POKEY TIMER AND AUDIO USAGE:
; TV sidetone source: Channel 3
; Sidetone via console spkr: Ch. 4

; Code element timer (16 bit res.):
; Channel 1 into Channel 2, IRQ from
```

```
; Channel 2. Note: Hi byte to 2; Low
; to 1.
VTIMR4 EQU $214
VTIMR2 EQU $212
FPREG EQU $CB; Point to speed/ratio
KBAT EQU $4D; OS sets to 0 when key
; pressed.
ATRCT EQU $4F; My attract flag. Go
; to attract mode when b7 is set.
; BMESS is the start of the message
; storage area. Note there is a $1000
; byte limit (4K) due to ANTIC scroll
; counter limit.
BMESS EQU $3000
; TBUF is the 256 byte transmit buffer
; for the keyboard sender...
ORG $8000
TBUF EQU *-$100

; INITIALIZE:

INIT: JSR POKINI
```

You can see above, I ORG'd the program to $8000, then used the program counter to compute an address $100 bytes lower to use for a buffer. The program will start at $8000 which is the value of label INIT and contains the code, JSR POKINI.

Oh yeah, you can have INCLUDE files. This means you can break your source code down into multiple files and a statement within one file will cause the assembler to open another and resume assembly there:

INCLUDE D2:FILE2.SOR

for example. At one time, I had a shell type executive source file which just had and ORG and a few equates, then half dozen INCLUDE statements to bring in other files, then an END statement. Now I use emulators and PC editors which aren't daunted by big files, so I don't split `em up anymore. There can be other reasons to split files up, though.

Getting back to pseudo-ops that declare memory space or data storage...

DB is flexible in storing single or multiple bytes or character strings.
DB 100 ; store single byte 100
DB 1, 7, $25 ; store three bytes (separate in the DB statement by commas)
DB `Howdy', $9B ; store a string, followed by a byte MWEN DW BMESS + 320; End of M-window

; Above stores a word with calculated value of BMESS + 320 and assignes the label MWEN to it.

Note that labels don't have to end in colons, although they can if you are used to that style.

DB HIGH MWEN ; store high byte of word MWEN

Note again that EQU and = don't actually cause anything to be stored in memory, but DB and DW do.

Numbers of different bases:

Use % for binary, as in %11001010
Use $ for hex, as in $01FF
Use plain for decimal, as 300
Use @ for octal as in @212 (why use octal? don't know)

Symbols (labels) must be unique in first 6 chars (others are ignored). Must start with a letter or number, or colon for local. Can start with ? and be excluded from reference map.

---

**Some Error Codes:**

A Address error, addressing mode not supported by opcode
D Duplicate label; using same label twice
E Expression error; can't understand expression in the address field of the source line
I Instruction field not recoginzed, three NOPs are generated
L Label field not recognized, three NOPs are generated
N Number error, doesn't match radix; > 16 bits, etc.
S Syntax error in statement; too many or two few address subfields
U Reference to an undefined symbol. (Very popular error! You mistyped a label name.)
V Expression overflow; value truncated

Here's a file I wrote to remind myself of how I use AMAC to assemble a big project on the PC using Xformer:

TO_NICK.TXT

This file is intend to remind me of what form the KEYER program is in and how I currently assemble it:

I've copied all the individual files into one big file called BIGKEY.PCF. PCF means PC file. This file has the all the EOLs (155 or $9B) converted to CR/LF (13,10) and the ATASCII TABs converted to ASCII TABs (127 or $7F to 09).

I edit the PCF file on the PC. Then I use batch file BIG_2_ATR.BAT to convert the CR/LFs and TABs back to Atari style using the utility TT. I make TT create a separate file called BIGKEY.ATF to be accessed by AMAC. Next, I run XFORMER on the PC, loading a disk image with DOS and AMAC as D1: and BIGKEY.ATF as D2. AMAC WILL run on D1 and assemble the native PC file BIGKEY.ATF from D2. It takes a couple of minutes to assemble. I use this command line:

L AMAC [return], then AMAC runs, then

D2:BIGKEY.ATF H=D:KEYER.OBJ

So the object code is put on the disk image D1.

How to get rid of the copy protection:

(Plus a little tutorial on Atari DOS file structure and function.)

You need a sector editor and you need to know how to look at the directory to locate the first sector number of the file AMAC.

My disassembly of the first sector of the file showed that the first thing it does is set up to read a bad sector. The way to fix it is to make the RUN (or maybe it's INIT?) address

start right after the bad sector check, effectively bypassing it. If you fix it with a sector editor, you'd see that the first sector of the file starts with the six byte header FF FF 00 26 2B 26. That tells DOS to load the file starting with the 7th byte (meaning the 6 byte header isn't part of the file), putting it at memory $2600 thru $262B. (The FFs are a RUN file identifier, 00 26 means start loading at $2600 and 2B 26 means end loading at $2600.)

This first segment loaded ends with the 50th byte of the sector (which is the 44th byte of the file). Next comes the header (Note: no FF's, which are optional except for the ones in the first segment's header) for the next load segment. It's just two bytes, loaded at $02E2 thru $02E3. This has special meaning to DOS. In a multi-segment load, if an address is loaded into $02E2, DOS begins execution at that address. So you see that starting with byte 51 of the segment is this data:

E2 02 E3 02 00 26

Which can be translated as, "start loading at $02E2, end loading at $02E3, data to load: $00, $26

By the way, you did know that 16 bit addresses are stored backwards (low byte first), right? This load segment stores address $2600 at address $2E2. That's the very start of the program where all that bad copy protection stuff takes place. Execution after the copy protection check succeeds continues at $2626, therefore we just want to change the header of the 2nd segment to look like this:

E2 02 E3 02 26 26

There, mission accomplished. Now the copy protection routine is still part of the program, but it no longer gets executed.