From ANALOG Computing - Issue 60 - May 1988 - Page 27

The MAC/65
De-Tokenizer

Convert your tokenized MAC/65
files into plain ol' English

by Charles Bachand

MAC/65 is an outstanding 6502 macro assembler for the Atari 8-bit computer from Optimized Systems Software, Inc. Our staff, and most of our regular contributors, use it whenever they do any assembly language programming. The cartridge contains one of the fastest assemblers on the market today, and the tokenized source-code files that it generates can save quite a bit of disk space. We highly recommend MAC/65 to anyone doing 6502 machine language software development.

However, (you knew that this was coming, didn't you?) the Catch-22 with MAC/65 is that the file format used to save code is not compatible with anything; if you want to use MAC/65 files, you need MAC/65. Or, at least until now, you did. **The MAC/65 De-Tokenizer** presented here will free you from this requirement.

The De-Tokenizer is a small Atari BASIC program that will convert a tokenized MAC/65 file back into readable English - or at least what passes for English from some of the programmers I know.

How *The MAC/65 De-Tokenizer* works

To understand the internal workings of The De-Tokenizer program, one must understand the rules by which MAC/65 compresses its files. There are actually two parts to a MAC/65 tokenized file. The first four bytes comprise the file header, which is broken into two parts.

```
Byte  1  2  3  4
     -- -- -- --
     FE FE xx xx ...
     \___/ \___/
       |     |
       |     Byte count (low-byte/high-byte)
      MAC/65 file identifier bytes
```

Bytes 1 and 2 each have the value of 254 (or $FE in hex notation.) If, when you try to load a file, MAC/65 sees something other than a 254 in these positions, it knows that the file is the wrong format and the editor will display a *File Type Error #23.*

Now, assuming that the load process gets you past this check, MAC/65 reads in the next 2 bytes (3 and 4), which specify the number of bytes of actual tokenized data that the file contains. This is a 16-bit value which is stored in the classic "low-byte/high-byte" format. Adding 4 - the number of bytes we've just read - to this number, will give us the total number of bytes in the file.

Once we get past these first 4 bytes, we begin to encounter the tokenized data - the place where all the real fun begins.

Getting on the token express

Atari BASIC is tokenized. This means that the programmers of this language devised a scheme to compress the data, in order to reduce memory usage and to increase the execution speed of its programs. A similar savings is also realized when storing tokenized programs out to cassette or disk.

This concept of file tokenization has been carried over into the design of OSS's MAC/65 as illustrated in the following example:

```
Assembly listing           Hex data
10 LOOP     LDA $0600      OA 00 OC 84 4C 4F 4F 50 51 05 00 06
20          CLC            14 00 04 2C
30          ADC -10        IE 00 07 4F 3E 08 OA
40          STA $0600      28 00 07 50 05 00 06
50          JMP LOOP       32 00 09 21 84 4C 4F 4F 50
```

What do the five lines of assembly code and the five lines of hex data in the above example have in common? They happen to be one and the same as far as MAC/65 is concerned! The listing takes 78 bytes as text, but only 39 bytes to store it in the computer's memory or out to a disk file. In this example alone, we've cut our storage requirements in half, and this degree of savings is not an uncommon occurrence when using MAC/65.

One line at a time

Just as every MAC/65 file has a file header, so does every tokenized line in the file have a 3-byte line header associated with it. For instance, the first three bytes in Line 10 from our example above are $0A, $00 and $0C in hex. This is further broken dowm as follows:

```
OA 00 OC 84 4C 4F 4F 50 51 05 00 OS
\___/ |  \_______Token data_______/
  |   Line length
  Line number (low-byte/high-byte)
```

The first two bytes represent the line number - again in the standard low-byte/high-byte format. It's easy enough to convert these two bytes back to a number that can be printed by multiplying the value of the second byte by 256 and adding the value of the first byte to it. Using our example, we have $00 (or 0 decimal), multiplied by 256, giving us 0 (so it's not an exceptionally exciting example) and then adding $0A (or 10 decimal) to it, which finally gives us a line number of 10. Wasn't that easy?

The third byte in the line tells us the number of bytes of tokenized data associated with that line. This count also includes the two line number bytes and the count byte. Our example of $0C (12 decimal) in this position signifies that the line contains 9 bytes of data, plus the 3 bytes of header data, giving us a total of 12 bytes.

A token for your thoughts

Now that we're finally past all this header stuff, we get to look at what you've all come here to see - namely MAC/65 tokens.

An assembly source statement line is in the form:

```
[label] [ (6502 instruction) or (assembly directive) ] [comment]
```

A string of characters, such as a label or a string in quotes, is identified by a token made up of the string's character count, with the high bit set, followed by the ASCII values of the string. So, if we have a 4-character string like "TEST" it will have a token of 132 ($84) or 128 + 4, followed by the ASCII values of the four characters T, E, S and T. String tokens are always greater than 128 in value.

If the value of the token is less than 128, then it's either a MAC/65 assembler directive or a 6502 instruction. Table 1 shows the allowed token substitution values.

Token	Replacement				
0*	"ERROR -"	32	"JSR"	64	"TSX"
1	".IF"	33	"JMP"	65	"TXA"
2	".ELSE"	34	"DEC"	66	"TXS"
3	".ENDIF"	35	"INC"	67	"TYA"
4	".MACRO"	36	"LDX"	68	"BCC"
5	".ENDM"	37	"LDY"	69	"BCS"
6	".TITLE"	38	"STX"	70	"BEQ"
7*	""	39	"STY"	71	"BMI"
8	".PAGE"	40	"CPX"	72	"BNE"
9	".WORD"	41	"CPY"	73	"BPL"
10	".ERROR"	42	"BIT"	74	"BVC"
11	".BYTE"	43	"BRK"	75	"BVS"
12	".SBYTE"	44	"CLC"	76	"ORA"
13	".DBYTE"	45	"CLD"	77	"AND"
14	".END"	46	"CLI"	78	"EOR"
15	".OPT"	47	"CLV"	79	"ADC"
16	".TAB"	48	"DEX"	80	"STA"
17	".INCLUDE"	49	"DEY"	81	"LDA"
18	".DS"	50	"INX"	82	"CMP"
19	".ORG"	51	"INY"	83	"SBC"
20	".EQU"	52	"NOP"	84	"ASL"
21	"BRA"	53	"PHA"	85	"ROL"
22	"TRB"	54	"PHP"	86	"LSR"
23	"TSB"	55	"PLA"	87	"ROR"
24	".FLOAT"	56	"PLP"	88*	Comment line
25	".CBYTE"	57	"RTI"	89	"STZ"
26	";"	58	"RTS"	90	"DEA"

27	".LOCAL"	59	"SEC"	91	"INA"
28	".SET"	60	"SED"	92	"PHX"
29	"*="	61	"SEI"	93	"PHY"
30	"="	62	"TAX"	94	"PLX"
31	".="	63	"TAY"	95	"PLY"

(* = see text)

Table 1

Looks simple, doesn't it? If a token value is 33, then it is a 6502 "JMP" instruction. If the value is 17, it's a MAC/65 ".INCLUDE" directive.

Of course, there are exceptions. A line that could not be tokenized properly, due to an error in syntax, has a 0 as its token byte. A comment line - one that does not contain any label or instructions - has a token value of 88. The rest of the line in both cases is stored as pure ASCII text.

If the token has a value of 7, then the label following it identifies the name of a macro that is to be called at this point in an assembly. Macro names are usually indented more than the regular mnemonics, so we output a few extra spaces before printing the macro name.

Now, once we get past the initial label and instruction/directive phase of the tokenization process, we encounter a second set of tokens for the MAC/65 operands which are listed in Table 2. These are the ones that we'll be using until we reach the end of the tokenized line.

If we bump into a token that is greater than 128, it is a label and it follows the same rules we outlined previously in handling labels. All the other operand tokens - those with values less than 128 - are handled with a simple substitution for their corresponding text values. The exceptions are for the tokens with values of 5, 6, 7, 8, 10 and 59.

Token	Replacement				
5*	"$"	28	">"	56	",Y"
6*	"$"	29	"<"	57	",X"
7*		30	"-"	58	")"
8*		31	"["	59*	""
10*	" ' "	32	"]"	61	","
11	"%$"	36	"!"	62	"#"
12	"%"	37	"^"	63	"A"
13	"*"	39	"\"	64	"("
18	"+"	47	".REF"	65	" " "
19	"-"	48	".DEF"	69	"NO"
20	"*"	49	".NOT"	70	"OBJ"
21	"/"	50	".AND"	71	"ERR"
22	"&"	51	".OR"	72	"EJECT"
24	"="	52	"<"	73	"LIST"
25	"<="	53	">"	74	"XREF"
26	">="	54	",X)"	75	"MLIST"
27	"<>"	55	"),Y"	76	"CLIST"
				77	"NUM"

(* = see text)

Table 2

Token 5 is used for word-length hex constants and is followed by two bytes containing the value of the constant in low-byte/high-byte format. For example, a hex constant like $0600 is tokenized as the three bytes 5, 0 and 6.

Token 6 is very similar to 5, but is used for 1-byte-long hex constants. A value like $80 would tokenize as the two bytes 6 and 128.

Tokens 7 and 8 are used for word and byte decimal constants and follow the same structure as the hex constant tokens 5 and 6. Thus, a decimal word constant like 256 stores as 7, 0, 1, while a byte-sized constant like 255 ends up as 8, 255.

Token 10 defines a character constant, which in MAC/65 is an apostrophe, followed by any displayable character. Thus, a reference to the ATASCII value of the capital letter A would appear in MAC/65 as A and be stored as 10, 65.

That just leaves us with the value of 59, which signifies the begirming of the comment field, to round off our oddball token list. All the bytes that follow the token, up to the end of the line, are part of the comment field and are output as ATASCII text.

How the *MAC/65 De-Tokenizer* works - really!

Sorry about suckering you into reading about MAC/65 tokenization methods with that false "How the program works" title near the begirming of the article, but you might have skipped over some wonderful pieces of information. I promise, the following paragraphs do in fact describe how the program works. Honest!

The De-Tokenizer is a bare-bones utility program. All you need do to make it run is supply it with the names of the MAC/65 file and the text file or device that you want the output sent to. If you're not sure of the name of the MAC/65 file, just hit RETURN and the program will list the directory of the disk in drive 1.

The program takes several seconds to initialize, and in that time, it reads a hst of strings, representing the MAC/65 tokens, from DATA statements into the array TOKEN$. It also reads the corresponding decimal values of the tokens, which are used as indexes in building two arrays that hold the indexes for the starting (Array TS) and ending locations (Array TE) of each substring in the TOKEN$ array. Once initialized, it's a simple matter to print out the ATASCII value of any known token.

For example, to print out the text for token 11 - the ".BYTE" directive - we could type the following line:

```
PRINT TOKEN$(TS(11),TE(11))
```

This, of course, will give us only one possible string, but it is easily modified by replacing the two numeric constants with variables. This was done in Line 1320 of the BASIC program to print out the token associated with the contents of the numeric variable A.

You might have noticed in Line 1390 that the routine used above to print out a token has a value of 128 added to the indexes, and, of course, there must be a very good reason for this. If you take a look at Tables 1 and 2, you'll see that quite a few of the token values used are common to both tables. MAC/65 doesn't mind this at all, since internally it uses two tables for doing substitutions. **The De-Tokenizer** program could have also used two sets of tables, but it was more to our advantage to employ only one.

If you remember from my little MAC/65 tutorial (the one I tricked you into reading), the token for a label is 128, plus the number of characters in the label, leaving all other tokens with a value less than 128. Hmm... If we leave the numbering of the instruction/directive tokens (Table 1) alone, and offset the numbering of the operand tokens (Table 2) by adding 128 to them, then they'll all fit into a 256-entry table. Yeah, yeah, that's the ticket! Then, to print an operand, we look at the second half of the look-up table (tokens 128-255), by adding 128 to the token value - just like the one being done in Line 1390.

Taking up space with source code

I wanted the output from **The MAC/65 De-Tokenizer** to be usable by owners of Atari's Assembler/Editor cartridge, so I elected not to try to convert the directives to those used by other 8-bit assemblers (like SynAssembler or Atari's Macro Assembler). It's up to the programmer to check for any incompatibility in this area.

The program does not output text in nice neat columns (remember, I told you this was a bare-bones program!), because it is largely a waste of precious disk space to do so. Most assembler programs won't care anyway, but if it bothers you that much, I'll be happy to let you modify my code.

Well, that's about it. I hope you're able to make as much use out of **The MAC/65 De-Tokenizer** as I had fun writing it (intense sarcasm here).

Charlie would like to thank Matthew J.W. Ratcliff, a heavy-duty proponent of MAC/65, for his aid in the testing of this program. The two-letter checksum code preceding the line numbers here is not a part of the BASIC program. For further information, see the "BASIC Editor II," in issue 47.

```
TQ 1000 REM MAC/65 TOKEN CONVERTER
LE 1010 REM (C) 1987 ANALOG COMPUTING
WL 1020 REM WRITTEN BY CHARLES BACHAND
IF 1030 REM
GC 1040 DIM TS(255),TE(255),HEX(15)
IQ 1050 DIM A$(40),TOKEN$(500)
TE 1060 GOSUB 1470:G05UB 16680
OT 1070 TRAP 1070:? :? "RETURN for direct
   ory or nane of":? " MAC/65  file";
PO 1080 INPUT A$:IF A$="" THEN GOSUB 1740
   :G0T0 1070
```

```
EL 1090 OPEN #1,4,0,A$
KA 1100 GET #1,A:GET #1,B
HZ 1110 IF A=254 AND A=B THEN 1130
BD 1120 ? " Not a MAC/65 File!":GOTO 1070
KJ 1130 GET #1,A:GET #1,B
PQ 1140 ? :? "File length = ";A+B*256+4
WT 1150 TRAP 1150:? :? "RETURN for direct
   ory or name of":? " OUTPUT file";
NO 1160 INPUT A$:IF A$="" THEN GOSUB 1740
   :GOTO 1150
GX 1170 OPEN #2,8,0,A$
IW 1180 REM
BJ 1190 REM PROCESS A LINE
IA 1200 REM
OZ 1210 TRAP 1440
YM 1220 GET #1,A:GET #1,B:GET #1,L
LZ 1230 L=L-3:PRINT #2;A+B*256;" ";
SO 1240 GET #1,A:L=L-1:IF A<>88 THEN 1270
WN 1250 FOR B=l TO L:GET #1,A:PUT #2,A
MV 1260 NEXT B:PRINT #2:G0T0 1220
MS 1270 IF A<128 THEN 1320
BR 1280 FOR I=129 TO A:GET #1,A
NZ 1290 PUT #2,A:L=L-1:NEXT I
OJ 1300 IF L=0 THEN ? #2:G0T0 1220
XE 1310 GET #1,A:L=L-1
FX 1320 ? #2;" ";T0KEN$(TS(A),TE(A));" ";
XI 1330 IF A=0 THEN 1250
OV 1340 IF L=0 THEN ? #2:G0T0 1220
XQ 1358 GET #1,A:L=L-1
HB 1360 IF A>128 THEN C=A:FOR B=129 TO C:
   GET #1,A:PUT #2,A:L=L-1:NEXT B:GOTO 13
   40
BG 1370 IF A=7 THEN GET #1,A:GET #1,B:? #
   2;A+B*256;:L=L-2:G0T0 1340
TL 1380 IF A=8 THEN GET #1,A:? #2;A;:L=L-
   1:GOTO 1340
KD 1390 ? #2;TOKEN$(TS(A+128),TE(A+128));
   :IF A=6 THEN GET #1,A:GOSUB 1680:L=L-1
   :GOTO 1340
XP 1400 IF A=5 THEN GET #1,B:GET #1,A:L=L
   -2:GOSUB 1680:A=B:GOSUB 1680:GOTO 1340
MH 1410 IF A=59 THEN FOR B=l TO L:GET #1,
   A:PUT #2,A:NEXT B:? #2:GOTO 1220
DI 1420 IF A=10 THEN GET #1,A:PUT #2,A:L=
   L-1
QP 1430 GOTO 1340
LL 1440 A=PEEK(195):IF A=136 THEN 1070
MY 1450 ? " ERROR #";A;" AT LINE ";PEEK(1
   86)+PEEK(187)*256:STOP
IW 1460 REM
IL 1470 REM SET-UP TOKEN TABLES
JC 1480 REM
PY 1490 ? :? " MAC/65  TOKEN CONVERTER"
KO 1500 ? :? "Initializing arrays, ";
IL 1510 ? "please wait...":C=1
MF 1520 READ TOKEN,A$:IF TOKEN<>1 THEN G
   OSUB 1600:GOTO 1520
```

```
JN 1530 T0KEN=185:A$=",X":GOSUB 1600
JX 1540 T0KEN=184:A$=",Y":GOSUB 1600
GC 1550 T0KEN=7:A$=" ":GOSUB 1600
RT 1560 T0KEN=189:A$=",":GOSUB 1600
QX 1570 T0KEN=193:A$=CHR$(34):GOSUB 1600
SJ 1580 T0KEN=182:A$=",X)":GOSUB 1600
MD 1590 T0KEN=183:A$="),Y"
GQ 1600 TS(TOKEN)=C:TOKEN$(C)=A$
MM 1610 TE(TOKEN)=LEN(TOKEN$)
QM 1620 C=LEN(TOKEN$)+l:RETURN
IR 1630 REM
JP 1640 REM HEX CONVERSI0N
IX 1650 REM
SD 1660 FOR A=0 TO 15:READ A$
VG 1670 HEX(A)=ASC(AS):NEXT A:RETURN
WL 1680 C=INT(A/16):B=A-C*16
CZ 1690 PUT #2,HEX(C):PUT #2,HEX(B)
AN 1700 RETURN
IN 1710 REM
FS 1720 REM READ DISK DIRECTORY
IT 1730 REM
UC 1740 OPEN #3,6,0,"D:*.*":TRAP 1755
QV 1750 INPUT #3,A$:? A$:GOTO 1750
FY 1755 CLOSE #3:RETURN
JC 1760 REM
WC 1770 REM TOKEN TABLE
JI 1780 REM
HP 1798 DATA 79,ADC,77,AND,84,ASL,68,BCC
OG 1800 DATA 69,BCS,70,BEQ,71,BMI,72,BNE
EL 1810 DATA 73,BPL,74,BVC,75,BVS,42,BIT
KT 1820 DATA 43,BRK,44,CLC,45,CLD,46,CLI
WS 1830 DATA 47,CLV,82,CMP,40,CPX,41,CPY
LL 1840 DATA 34,DEC,48,DEX,49,DEY,78,EOR
YQ 1850 DATA 35,INC,50,INX,51,INY,33,JMP
OO 1860 DATA 32,JSR,81,LDA,36,LDX,37,LDY
MW 1870 DATA 86,LSR,52,NOP,76,ORA,53,PHA
LK 1880 DATA 54,PHP,55,PLA,56,PLP,85,ROL
TP 1890 DATA 87,ROR,83,SBC,59,SEC,60,SED
HY 1900 DATA 61,SEI,80,STA,38,STX,39,STY
SY 1910 DATA 62,TAX,63,TAY,64,TSX,65,TXA
KT 1920 DATA 66,TXS,67,TYA,57,RTI,58,RTS
LC 1930 DATA 21,BRA,90,DEA,91,INA,92,PHX
IR 1940 DATA 93,PHY,94,PLX,95,PLY,89,STZ
SU 1950 DATA 22,TRB,23,TSB,29,*=
QR 1960 DATA 14,.END,26,;,19,.ORG,30,=
IV 1970 DATA 20,.EQU,11,.BYTE,12,.SBYTE
TL 1980 DATA 25,.CBYTE,13,.DBYTE,9,.WORD
HG 1990 DATA 18,.DS,2,.ELSE,3,.ENDIF
PJ 2000 DATA 10,.ERROR,24,.FLOAT
CS 2010 DATA 1,.IF,17,.INCLUDE,27,.LOCAL
IV 2020 DATA 15,.OPT,8,.PAGE,28,.SET
AU 2030 DATA 0,ERROR - ,4,.MACRO,5,.ENDM
YP 2040 DATA 6,.TITLE,31,.=,16,.TAB
YK 2050 DATA 190,#,187, ,134,$,133,$
UG 2060 DATA 180, <,181, >,138,',159,[
PZ 2070 DATA 160,],146,+,149,/,148,*
GN 2080 DATA 167,\,150,&,164,!,165,^
```

```
FU 2090 DATA 152,=,156,>,157,<,158,-
NJ 2100 DATA 147,-,155,<>,154,>=,153,<=
WT 2110 DATA 179, .OR ,197,NO ,201
XA 2120 DATA LIST,178, .AND ,199
HO 2130 DATA ERR,177, .NOT ,200
YO 2140 DATA EJECT,176, .DEF ,198
YN 2150 DATA OBJ,175, .REF ,203,MLIST
BR 2160 DATA 204,CLIST,205,NUM,202,XREF
SX 2170 DATA 192,(,186,),139,%$,141,*
AY 2180 DATA 191,A,140,%,-1,XXX
CM 2190 DATA 0,1,2,3,4,5,6,7,8,9
KR 2200 DATA A,B,C,D,E,F
EV 2210 END
```