ATasm v1.06 ("Clurichaun Release")

A mostly Mac/65 compatible 6502 cross-assembler

e-mail: mschmelzenbach@acm.org

Table of Contents

Introduction

ATasm was born out of the desire for a fast, Atari specific cross-assembler. With the recent advent of some quite complete Atari emulators, I decided to brush the dust off of some (very) old projects and code a few quick programs. Back in the good old days, when I was programming on 'real iron,' my favorite assembler was OSS's amazing Mac/65 cartridge. Sadly, the cartridge was 900 miles away, along with my trusty 130XE. So, I looked around on the Internet a bit, and found FTe's disk release of Mac/65 (v4.2). This was usable, but not nearly as nice as the cartridge version. I also found that over the past few years, I had become used to writing code on a 132x60 character display, with instant compile times. I decided to use a cross-assembler, since that seemed to fulfill my requirements. I tried out as6502 from UMich, and Fachat's XA. Although both produced solid code, neither one had all the Atari specific directives and features I had become accustomed to using Mac/65.

And so, a few days later, ATasm v0.1 was created. For a long period of time, I continued using ATasm, adding features to the assembler as I needed them. Since version 0.9, ATasm has been close enough to the original Mac/65 such that the Mac/65 manual provides a good overview for ATasm. In fact, this manual is very heavily based on the original manual. Reading the Mac/65 manual in addition to this document is recommended, since they develop many more examples in greater detail.

If you are familiar with the original product, then you should only need to read chapter 5, which outlines the known differences between ATasm and Mac/65.

Version History

version 0.90 - initial public release

version 0.91 - added '-x' command-line option, providing initial .XFD support

version 0.92 - added '-u' command-line option, providing undocumented opcodes; also added Appendix C in the manual

version 0.93 - updated e-mail address, removed some spurious warnings when the .DC directive was used, fixed a problem with indirect jmp; Thanks to Carsten Strotman for finding this bug!

version 0.94 - fixed embedded ';' in strings, jmp to zero page locations, mapped 'LDA/STA zero,y => LDA/STA a,y to emulate Mac/65 behavior, a few minor updates to this document

version 0.95 - fixed an error with .incbin that would result in an extra byte being stored

version 0.96 - fixed an error with missing lines at end of files, added .OPT ERR/NO ERR, .OPT OBJ/NO OBJ

version 0.97 - fixed a bug with .incbin introduced in the 0.96

version 1.00 - added added several zp, y -> absolute address operators (sbc/adc/and/eor/ora/cmp); Allowed compiliation to addresses >$fd00; Thanks to Manual Polik for finding these! Fixed problem with immediate value of a comma: #', Fixed some serious problems with macro definitions; Explicitly released the

package under the GPL.

version 1.01 - added raw binary output, fixed a problem with zp,y

version 1.02 - added include path and define command-line options, new .OPT directive enabling illegal opcodes, fixed a bug with data commands emiting code without a set origin; Beginning of a test suite, tweaks to makefile; These changes were all provided by B. Watson.

version 1.03 - added mapping for zero page JSR, enforce label name restrictions, added interpretation of #$LABEL (with warnings); These changes were suggested by Maitthias Reichl.

version 1.04 - fixed some serious problems with macro expansion, added fill byte command-line parameter, limited display of errors and warnings to one pass only, initial support of multiple passes to prevent the dreaded "PHASE ERROR", fixed problem with command-line definitions

version 1.05 - added new directives .BANK, .SET 6, and .OPT LIST/NO LIST; Support for enhanced/single density .ATR files; Preliminary support for Atari++ snapshot files

version 1.06 – applied Maitthias Reichl's patch to allow negative offsets with .SET 6 directives; some internal clean-up regarding predefined directives; allow arithmetic expressions in REPEAT blocks; Better detection of resized labels. fixed a buffer overflow problem; added -l option to allow label output; Compiling Windows executable with mingw.

Chapter 1: ATasm

1.1 Installation

The normal binary distribution will include the following files:

ATasm.txt: this file

atasm.exe: The Windows executable. This program was compiled with TDM/MinGW, an excellent free Windows C compiler based on gcc.

src/.*: The source code for ATasm, including Makefile.

examples/.m65*: example assembly source code

The program should compile cleanly on all UNIX platforms. Simply move into the source directory and type 'make'. Notice that if you want to merge the resultant object code with Atari800 or WinAtari800 emulator save states, you will also need to get the ZLIB library. The zlib home page is <http://www.gzip.org/zlib/>

1.2 Usage

Using ATasm is fairly simple. The program is invoked with the following command line parameters:

```
atasm [options] <file.m65>
```

where available options are:

-v: prints assembly trace
-s: prints symbol table
-u: enables undocumented opcodes
-r: saves object code as a raw binary image
-f*value*: set raw binary fill byte to *value.* This value should be a number between 0-255. Decimal and hexidecimal numbers are accepted.
-m*[fname]*: defines template emulator state file *fname.* If the parameter *fname* is not provided, ATasm will attempt to use the default state file "statefile.a8s"
-l*fname*: exports a symbol table (usable by the Atari800 monitor) to *fname*.
-x*fname*: saves object file to .XFD/.ATR disk image *fname*
-o*fname*: saves object file to *fname*
-D*symbol=value*: pre-defines [symbol] to [value]
-I*directory*: search *directory* for .INCLUDE files

The assembly trace and symbol table dump will be sent to stdout. This can be piped to a file if desired.

Typically, ATasm will generate a single object file 'fname.65o' This file is in Atari's binary file format, suitable for loading into machine memory via Atari DOS 2.5 command 'L' (or other similar methods).

However, it is also possible to assemble directly into an emulator's memory snapshot. Versions of the Atari800 emulator (originally by David Firth) greater than 0.9.8g, and Atari800Win versions greater than 2.5c allow the saving and loading of the machine state. ATasm can read in a state file, compile the source code and produce a new state file which can then be loaded directly into the emulator with the 'Load Snapshot' option. This version of ATasm is compatible with versions 2 and 3 of the state file specification format. As of version 1.05, ATasm can also assemble into the snapshots generated by Atari++ written by Thomas Richter.

Object code can also be assembled to Atari disk images used by many Atari and SIO emulators. Disk images can either be in the raw .XFD format or the the more formalized .ATR format. The disk image must be either a single density or enhanced density disk formatted with Atari DOS 2.0s, Atari DOS 2.5, or compatible formats.

Ex.:

```
atasm sample.m65
```

This will assemble the file sample.m65, and generate an Atari object file 'sample.65o'.

```
atasm -v -s sample.m65 | more
```

This will also generate 'sample.65o', but will also dump the symbol table and verbose assembly output to the paging program 'more'.

```
atasm -DBASIC -DOSB= -DFOO=128 sample.m65
```

This too will generate 'sample.65o', but when assembling the following defines will be observed:

- The label 'BASIC' will have a value of 1
- The label 'OSB' will have a value of 0
- The label 'FOO' will have a value of 128

Values specified on the command-line must be numbers. If you are giving the value in hexadecimal, use the form $xxxx. Notice that the dollar sign may need to be escaped with a '\' depending on your command interpreter.

```
atasm -u iopcode.m65
```

This will generate the binary file 'iopcode.65o'. However the source file can include the undocumented 6502 instructions listed in Appendix C. Without this flag, the undocumented opcodes will generate assembly errors. Notice that due to their undocumented status, use of these opcodes is not recommended. However, many demo coders use them effectively -- just be aware that many emulators may not support their use.

```
atasm -xdos25.xfd sample.m65
```

This will generate 'sample.65o' and create the file 'SAMPLE.65O' on the .XFD image 'dos25.xfd'. There is no space between the -x and the filename. If the file 'sample.65o' already exists on the disk image, it WILL be overwritten.

```
atasm -matari800.a8s sample.m65
```

This will generate 'sample.65o' and create 'sample.a8s' an Atari800 emulator state file. The state file is created by reading in the previously saved statefile 'atari800.a8s' and overlaying the binary generated by the assembler. There is no space between the -m and the filename. If the filename is omitted, ATasm will attempt to load the default statefile 'atari800.a8s'.

To generate a statefile in Atari800Win, start the emulator, then select one of option under the File=>Save State pull down menu. In the Atari800 emulator, press F1 to access the emulation menu and select the 'Save State' option. Once you have a valid statefile, ATasm can use it as a template.

```
atasm -r sample.m65
```

This will generate a raw binary image of the object file called sample.bin. This is useful if you are developing a VCS game to be loaded in an emulator like stella. The image will start at the lowest memory location that has been assembled to, and save a complete block to the highest memory location assembled to. Any intervening space will be filled with the value of the fill byte. By default, this is hex value 0xff. To change the fill byte, specify the desired value with the -f parameter. If you are specifying the byte in hexadecimal, you may need to escape the '$' depending on your command interpreter.

Chapter 2: 6502 Assembly

2.1 The Assembler

ATasm aims to be as closely backwards compatible as possible to the original Mac/65 cartridge. However, some limitations imposed by the relatively small memory size of the

8-bit world have been lifted. See Chapter 4 for a list of differences between the two assemblers.

ATasm is primarily a two-pass assembler, although it will attempt to correct phase errors with additional passes, if necessary. It will read in the assembly source one line at a time and, if no errors are encountered, output a binary file. All input is case-insensitive.

Source lines have the following format:

[line number] [label] [<6502 opcode> <operand>] [comment]

A few items to note:
- Line numbers are optional, and are completely ignored if they exist.
- Labels can start with the symbols '@','?', or any letter. They may then consist of any alphanumeric character or the symbol '_'.
- Labels may not have the same name as a 6502 opcode.
- Comments must be preceded by a ';'.

2.2 Opcode format

Refer to Appendix A for a list of valid instruction mnemonics

2.3 Operand format

Operands consist of an arithmetic or logical expression which can consist of a mixture of labels, constants and equates.

Constants can be expressed either in hexadecimal, decimal, binary, character or string form.

Hex constants begin with '$'
```
Ex: $1, $04, $ff, $1A
As used:  lda #>$601
          .BYTE $02,$04,$08,$10
```

Binary constants begin with '~'
```
Ex: ~101, ~11
As used:  lda #~11110000
          .BYTE ~00011000,~00111100,~01111110
```

Character constants begin with a single quote (')
```
Ex: 'a, 'A
As used:  lda #'a+$10
          .BYTE 'a,'B
```

Strings are enclosed in double quotes (")
```
Ex: "Test"
As used:  .BYTE "This is a tes",'t+$80
```

Decimal constants have no special prefix
```
Ex: 10,12,128
As used: lda #12+8*[3+4]
```

Often, the format of the operand will determine the addressing mode of the operator. Refer to Appendix B for a complete breakdown of valid addressing modes, and examples of their format.

Briefly:
- Immediate operands are prefaced with '#'.
- (operand,X) and (operand),Y designate indirect addressing modes.
- operand,X and operand,Y designate indexed addressing modes.

The symbol '*' designates the current location counter, and can be used in expression calculations.

Notice that 'A' is a reserved symbol, used for accumulator addressing.

2.4 Operators and expressions

The following operators are grouped in order of precedence. Operators in the same precedence group will be evaluated in a left to right manner.

Group 1: Parenthesis

[ ] Notice that these parenthesis are really braces! This allows the assembler to disambiguate parenthetical expressions from indexing methods

Group 2: Unary operators

> Returns the high byte of the expression.
< Returns the low byte of the expression.
- unary minus, negates an expression.
.DEF <label> Returns true if label is defined.
.REF <label> Returns true if label has been referenced.

Group 3: Logical Not

.NOT Returns true if an expression is zero

Group 4: Multiplication and division

/ division
* multiplication

Group 5: Addition and subtraction

+ addition
- subtraction

Group 6: Binary operators

& binary AND
! binary OR
| binary OR (alternative representation)
^ binary EOR

Group 7: Logical comparisons

= equality, logical
> greater than, logical
< less than, logical
<> inequality, logical
>= greater or equal, logical
<= less or equal, logical

Group 8

.OR performs a logical OR

Group 9

.AND performs a logical AND

Chapter 3 Compiler directives, Conditional assembly, and Macros

3.1 Overview

ATasm implements many Mac/65 directives. However, there are several modifiers that are simply ignored (.END,.PAGE,.TAB,.TITLE), or are only partially implemented (.SET). For the most part, the important directives that effect code generation are intact. Some new directives have been added such as .DS, .INCBIN, .WARN, .BANK, and .REPT/.ENDR. In addition, non-standard .OPT directives have been added (see section 3.14)

In the following sections the following notation is used:

<addr> denotes an unsigned word used as a valid Atari address
<float> denotes a floating point number
<word> denotes a word value
<byte> denotes a byte value
<string> denotes a string enclosed in double quotes
<char> denotes a character preceded by a single quote
<label> denotes a legal ATasm label
<macro name> denotes a legal ATasm label used as a macro name
<expression> denotes a legal ATasm expression
<filename> denotes a system legal filename, optionally enclosed in double quotes

Also, symbols enclosed in brackets '[' ']' are optional.

*3.2 *=<addr>*

This sets the origin address for assembly.

3.3 .DS <word>

(Define Storage) This reserves an area of memory at the current address equal to size <word>. This is equivalent to the expression *=*+<word>

3.4 .DC <word> <byte>

(Define Constant storage) This fills an area of memory at the current address equal to size <word> with the byte value <byte>

3.5 <label> = <expression> or <label> .= <expression>

Assigns the specified label to a given value. The .= directive allows a label to be assigned different values during the assembly process. See section 3.17 for an example of using this.

3.6 .BYTE [+<byte>],<bytes|string|char>

Store byte values at the current address. If the first value is prefaced by a '+', then that value will be used as a constant that will be added to all the remaining bytes on that line.

```
Ex:
  .BYTE +$80,$10,20,"Testing",'a
```
will generate the following byte sequence:
```
  90 94 D4 E5 F3 F4 E9 EE E7 E1
```

```
  .SBYTE [+<byte>],<bytes|string|char>
```
This is the same as the .BYTE directive, but all the byte values will be converted to Atari screen codes instead of ATASCII values. This conversion is applied prior to the constant addition.

```
Ex:
  .SBYTE +$80,$10,20,"Testing",'a
```
will generate the follow byte sequence:
```
  90 94 D4 B4 F3 F4 E9 EE E7 E1
```

```
  .CBYTE [+<byte>],<bytes|string|char>
```
This is the same as the .BYTE directive, except that the final byte value on the line will be EOR'd with $80. This format is often used by print routines that use the high-bit of a character to indicate the end of a string.

```
Ex:
  .CBYTE +$80,$10,20,"Testing",'a
```
will generate the following byte sequence:
```
  90 94 D4 E5 F3 F4 E9 EE E7 61
```

3.7 .DBYTE <words>

Stores words in memory at the current memory address in MSB/LSB format.
```
Ex:
  .DBYTE $1234,-1,1
```
will generate:
```
  12 34 FF FF 00 01
```

```
.WORD <words>
```
Stores words in memory at the current memory address in native format (LSB/MSB).

```
Ex:
  .WORD $1234,-1,1
```
will generate:
```
  34 12 FF FF 01 00
```

3.8 .FLOAT <float>

Stores a 6 byte BCD floating point number in the format used in the Atari OS ROM.

```
Ex:
  .FLOAT 3.14156295,-2.718281828
```
will generate:
```
  40 03 14 15 62 95 C0 27 18 28 18 28
```

3.9 .IF <expression>,.ELSE,.ENDIF

These statements form the basis for ATasm's conditional assembly routines. They allow for code blocks to be assembled or skipped based on the value of an expression. The expression following the .IF directive will be evaluated and if true (or non-zero) the statements following the .IF up to the matching .ELSE or .ENDIF will be assembled. Otherwise, the code block will be skipped. The .ELSE block is optional and only needed if you want one block of code to be assembled when the expression is true and another to be assembled if the expression is false. The end of the conditional assembly block must be denoted with the .ENDIF directive.

3.10 .INCLUDE <filename>

Include additional files into the assembly. Using Mac/65, .INCLUDEs could only be nested one level deep. However, ATasm allows arbitrary nesting of .INCLUDE files. Quotes around the filename are optional. Notice that the '#Dn:' filespec is not applicable since ATasm is not accessing Atari disks (or disk images). Instead, the current working directory on the host machine will be searched for the file. The filename can include full path if desired.

3.11 .INCBIN <filename>

This includes the contents of binary file at the current memory position. This is useful for including character sets, maps and other large data sets without having to generate .BYTE entries.

3.12 .ERROR <string>

This will generate an assembler error, printing the message specified in the string parameter. The error will halt assembly.

3.13 .WARN <string>

This will generate an assembler warning, printing the message specified in the string parameter. The warning will be included in the warning count at the end of the assembly process.

3.14 .OPT [NO] <string>

This will set or clear specific compiler options. Currently, ATasm only implements the following options: ERR, OBJ, LIST and ILL. By default, both ERR and OBJ options are set, while the ILL and LIST options are off. If ERR is turned off then all warnings that would normally be sent to the screen will be suppressed. Notice that this behavior is subtly different than the original Mac/65 program which suppressed both warnings and errors. OBJ is used to control whether or not object code is stored in the binary image. Again, behavior is changed from the original environment. Setting .OPT NO OBJ could be useful if you wish to use label values in your source code as reference only, without actually generating code. The ILL opt toggles illegal opcodes availability. Illegal opcodes can be used inside areas of code surrounded by .OPT ILL, overriding the command-line parameter. The LIST opt can be used to override the command line -v argument and/or turn off the generation of screen output for certain sections of source files (for instance, long sections of data).

3.15 .LOCAL

This creates a new local label region. Within each local region, all labels beginning with '?' are assumed to be unique within that region. This allows libraries to be built without fear of label collision. Notice that although Mac/65 was limited to 62 local regions, ATasm has virtually unlimited regions (65536 regions). Local labels may be forward referenced like other labels, but they will not appear the symbol table dump at the end of the assembly processes.

3.16 .MACRO <macro name>, .ENDM

The .MACRO directive must be paired with an .ENDM directive. All macros must be defined before use. Once defined, a macro can be called with optional parameters, and are then functionally equivalent to a user-defined opcode. However, while opcodes are by their very nature fairly simple, macros can be quite complex. Notice that unlike Mac/65, macros may NOT have the same name as an existing label. Macro definitions cannot contain other macro definitions (although they can use existing macros). All labels within a macro are assumed to be local to that macro, but can be accessed from outside.

There are two types of macro parameters, expressions and strings. They can be referenced by using '%' for expression parameters and '%$' for string parameters followed by a number indicating what parameter to use. The parameter number can be a decimal number, or a label enclosed with parenthesis. So, %1 accesses the first parameter as an expression, and %$1 accesses the first parameter as a string.

Parameter %0 returns the total number of parameters passed to the macro, and %$0 returns the macro name.

When calling a macro, parameters can be separated either by commas or by spaces.

```
Ex:
       .MACRO VDLI
         .IF %0<>1
           .ERROR "VDLI: Wrong number of parameters"
```

```
    .ELSE
      ldy # <%1
      ldx # >%1
      lda #$C0
      sty $0200
      stx $0201
      sta $D40E
    .ENDIF
  .ENDM
```

This macro sets the display list interrupt to the address passed as its first parameter.

```
  .MACRO ADD_WORD
    .IF %0<2 OR %0>3
      .ERROR "ADD_WORD: Wrong number of parameters"
    .ELSE
      lda %1
      clc
      adc %2
      .IF %0=3
        sta %3
      .ELSE
        sta %2
      .ENDIF
        lda %1+1
        adc %2+1
      .IF %0=3
        sta %3+1
      .ELSE
        sta %2+1
      .ENDIF
    .ENDIF
  .ENDM
```

This macro has different results depending on its invocation. If called with two parameters:

```
ADD_WORD addr1,addr2
```

then the word value at addr1 is added to the word value at addr2. However, if called with three parameters:

```
ADD_WORD addr1,addr2,addr3
```

then the result of adding the word values in addr1 and addr2 is stored in addr3.

For more complicated macro examples, see the Mac/65 instruction manual or examine the included file 'iomac.m65' from the original Mac/65 install.

3.17 .REPT <word>, .ENDR

The .REPT directive must be paired with an .ENDR directive. All statements between the directive pair will be repeated <word> number of times.

```
Ex:
  .rept 4
      asl a
  .endr
```

generates:

```
          asl a
          asl a
          asl a
          asl a
```

and a more complicated example:

```
      table   .rept 192
            .word [*-table]/2*40
            .endr
```

generates a look-up table starting:

```
      00 00 28 00 50 00 78 00 A0 00 C8 00 F0 00 18 01 ...
```

which might be useful in a hi-res graphics mode plotting routine.

Another interesting example is inspired by a question from Tom Hunt:

```
        shapes
              r .= 0
              .rept 8
                .dbyte shape1+r*16
              r .= r+1
              .endr
        shape1
              r .= 1
              .rept 8
              .dbyte ~1111000000000000/r
              .dbyte ~1100000000000000/r
              .dbyte ~1000000000000000/r
              .dbyte ~0001000000000000/r
              .dbyte ~0000100000000000/r
              .dbyte ~0000010000000000/r
              .dbyte ~0000001000000000/r
              .dbyte ~0000000100000000/r
              r .= r * 2
              .endr
```

This will generate 8 instances of an arrow, with each instance shifted one bit to the right. It also creates a look-up table indexing into the top of each arrow.

3.18 .SET 6, <expression>

This directive will cause code to assemble to the current location plus the value of the given expression. This is useful for writing routines which can be copied from a cartridge area or bank-switched memory address into RAM.

Note that this is the only .SET directive from the original Mac/65 that is implemented. However, ATasm's implementation is slightly different. ATasm allows a full expression to be used as a parameter rather than simply an address. It also allows negative values as well as positive. Be aware that using forward defined variables inside of the .SET region to define the expression will cause a phase error.

3.19 .BANK

This directive was suggested by Chris Hutt to assist in building cartridge images that are greater than 64K in length. Basically, this directive will in essence start a new assembly in memory. However, addresses and labels available in one .BANK can still be referenced in the next .BANK. When saving the obj file, the banks are appended. This allows files larger than 64K to be assembled.

So the following produces a 32K file:

```
*=8000
.include "bank0code.asm"
*=bfff
.byte $ff ; ensure bank takes up exacly 16K

.bank
*=8000
.include "bank1code.asm"
*=bfff
.byte $ff
```

This directive is also handy for coding loaders that use the INITAD vector:

```
.bank
*=$4000
init
  .include "initcode.asm"

.bank
*=$2e2      ; when DOS loads an address into 2e2 (INITAD, it will
.word init  ; jsr immediately to that location, upon RTS
              ; it will continue to load...)
.bank
*=$6000
.include "restofthecode.asm"
```

Chapter 4: Incompatibilities with Mac/65

Perhaps most importantly, ATasm works with ASCII files, not ATASCII or Mac/65 tokenized save files. If you must use a tokenized file there are programs available on UMICH to convert tokenized files to ATASCI (or load the file in Mac/65 and LIST it to disk). Then use a filter program such as 'a2u' to convert the ATASCII to ASCII.

1) Comment lines only begin with ';' not with '*' -- sorry
2) Labels cannot contain embedded '.'s (but can contain '_'s)
3) The character '|' can be used in place of '!' as a binary OR
4) Macros can have arbitrary number of parameters and can be nested arbitrarily deep during invocation.
5) .INCLUDEs can be arbitrarily nested.
6) There are an unlimited number of .LOCAL regions (well, 65536 of them)
7) Macro names must be unique and cannot be the same as an existing label
8) Macro parameters can be separated by commas or by spaces
9) .END,.PAGE,.TAB,.TITLE, and most .SET directives are ignored
10) Extra directives .DC, .DS, .INCBIN, .WARN, .REPT/.ENDR have been added

11) .OPT ERR, .OPT NO ERR, .OPT OBJ, .OPT NO OBJ have different behavior than the original (see section 3.14), all other .OPT directives are ignored.
12) Operands are reserved words and cannot be used as labels or equates.

If you run across other incompatibilities or have a burning desire for a new feature, send them to me, and I will update this section (and possibly even update ATasm's behavior)

Chapter 5: A brief digression on writing ATasm

ATasm has been in development on and off for well over five years, evolving as needs dictated. Unfortunately, this evolution has resulted in rather patchy code in places. For instance, originally, the tokenizer was written as a free-form compiler(!). At the time, I felt that it would be more useful to allow the programmer full freedom when entering code. However, this decision means that it is then impossible to distinguish between labels, embedded compiler directives, and macros. This results in a few of ATasm's amusing quirks (no embedded '.'s in labels, unique label and macro names, and probably other darker characteristics).

When programming in 6502 assembly language, I actually tend not to heavily use macros, conditional assembly or many other of the features developed to make programming less burdensome. I think this is because I originally learned assembly on the old Atari Assembler cartridge, and never unlearned old habits. The upshot is, the macro facilities are not heavily tested. I have successfully compiled the sample files that come with the Mac/65 disk based assembler, but really crazy macros may not give the anticipated result. If you stumble across code that ATasm incorrectly handles, isolate the shortest example that you can and send it to me.

Blatant plug Also be sure to check out EnvisionPC, available available at http://atari.miribilist.com/envision/index.html. This is a very useful cross-platform graphics utility that runs native on the PC (Linux/Windows) or on MacOSX. It allows you to easily design all those nifty Atari character sets and maps. The original Atari based Envision always begged for a mouse interface... and now it exists. EnvisionPC is a full-featured character editor and map-maker base on the original APX Envision graphics utility. It supports ANTIC modes 2-7, 10 font banks loaded concurrently, and map sizes of up to 512x512 characters. Download it and start designing new games today.

Chapter 6: Bug reports, Feature Requests and Credits

This program would not be what it is today without the help of the following people.

Patches and code contributions: Mark Schmelzenbach, B. Watson, Dan Horak

Bug Reports and Feature requests: Cow Claygil, Doug Hodson, Dan Horak, Tom Hunt, Chris Hutt, Manuel Polik, Carsten Stroten, Thompsen, Greg Troutman, B. Watson... and many others.

Appendix A: Summary of 6502 Opcodes

```
ADC   ADd to accumulator with Carry.
```

```
AND   binary AND with accumulator.
ASL   Arithmetic Shift Left. Bit0=0 C=Bit7.
BCC   Branch on Carry Clear.
BCS   Branch on Carry Set.
BEQ   Branch on result EQual (zero).
BGE   Branch Greater than or Equal (alternate form of BCS)
BIT   test BITs in memory with accumulator.
BLT   Branch Less Than (alternate form of BCC)
BMI   Branch on result MInus.
BNE   Branch on result Not Equal (not zero).
BPL   Branch on result PLus.
BRK   forced BReaK.
BVC   Branch on oVerflow Clear.
BVS   Branch on oVerflow Set.
CLC   CLear Carry flag.
CLD   CLear Decimal mode.
CLI   CLear Interrupt disable bit.
CLV   CLear oVerflow flag.
CMP   CoMPare with accumulator.
CPX   ComPare with X register.
CPY   ComPare with Y register.
DEC   DECrement memory by one.
DEX   DEcrement X register by one.
DEY   DEcrement Y register by one.
EOR   binary Exclusive-OR with accumulator.
INC   INCrement memory by one.
INX   INcrement X register by one.
INY   INcrement Y register by one.
JMP   unconditional JuMP to new address.
JSR   unconditional Jump, Saving Return address.
LDA   LoaD Accumulator.
LDX   LoaD X register.
LDY   LoaD Y register.
LSR   Logical Shift Right. (Bit7=0 C=Bit0).
NOP   No OPeration.
ORA   binary OR with accumulator.
PHA   PusH Accumulator on stack.
PHP   PusH Processor status register on stack.
PLA   PulL Accumulator from stack.
PLP   PulL Processor status register from stack.
ROL   Rotate one bit Left (mem. or acc., C=Bit7 Bit0=C).
ROR   Rotate one bit Right (mem. or acc., C=Bit0 Bit7=C).
RTI   ReTurn from Interrupt.
RTS   ReTurn from Subroutine.
SBC   SuBtraCt from accumulator with borrow.
SEC   SDt Carry flag.
SED   SEt Decimal mode.
SEI   SEt Interrupt disable status.
STA   STore Accumulator in memory.
STX   STore X register in memory.
STY   STore Y register in memory.
TAX   Transfer Accumulator to X register.
TAY   Transfer Accumulator to Y register.
TSX   Transfer Stack pointer to X register.
TXA   Transfer X register to Accumulator.
TXS   Transfer X register to Stack pointer.
TYA   Transfer Y register to Accumulator.
```

Appendix B: 6502 Addressing modes

Absolute: The word following the opcode is the address of the operand.

```
Ex.    LDA  $0800
```

Absolute, indexed X: The word following the opcode is added to register X (as an unsigned word) to give the address of the operand.

```
Ex.    LDA  $FE90,X
```

Absolute, indexed Y: The word following the opcode is added to register Y (as an unsigned word) to give the address of the operand.

```
Ex.    LDA  $FE90,Y
```

Accumulator: The operand is the accumulator.

```
Ex.    LSR  A
```

or, an alternate form:

```
       LSR
```

Immediate mode: The operand is the byte following the opcode.

```
Ex.    LDA  #$07
```

Implied: The operands are indicated in the mnemonic.

```
Ex.    CLC
```

Indirect, absolute: The word following the opcode is the address of a word which is the address of the operand.

```
Ex.    JMP  ($0036)
```

Relative: The byte following the opcode is added (as a signed word) to the Program Counter to give the address of the operand.

```
Ex.    BCC  $03
Ex.    BCC  $0803 (alternate form)
```

Zero page absolute: The byte following the opcode is the address on page 0 of the operand.

```
Ex.    LDA  $1F
```

Zero page, indexed X: The byte following the opcode is added to register X to give the address on page 0 of the operand.

```
Ex.    LDA  $2A,X
```

Zero page, indexed Y: The byte following the opcode is added to register Y to give the address on page 0 of the operand.

```
Ex.    LDX  $2A,Y
```

Note: Although technically the opcodes LDA $20,Y and STA $20,Y are illegal, ATasm (and many other 8-bit assemblers) will convert this to an absolute indexed addressing mode.

Zero page, indexed, indirect: The byte following the opcode is added to register X to give the address on page 0 which contains the address of the operand.

```
Ex.    LDA  ($2A,X)
```

Zero page, indirect indexed: The byte following the opcode is an address on page 0. This word at this address is added to register Y (as an unsigned word) to give the address of the operand.

```
Ex.    LDA  ($2A),Y
```

Appendix C: Atari "Sally" 6502 Undocumented Opcodes

Original list (version 3.0, 5/17/1997) was created by Freddy Offenga (offen300@hio.tem.nhl.nl). Additional credits: Joakim Atterhal, Adam Vardy, Craig Taylor;

References and list sources:

1) "Illegal opcodes", WosFilm and Frankenstein, *Mega Magazine #2*, December 1991.
2) "Illegal opcodes v2", WosFilm and Fran-ken-stein, *Mega Maga-zine #6*, October 1993.
3) "Illegal Opcodes der 65xx-CPU", Frank Leiprecht, *ABBUC Sondermagazin 10*, *Top-Maga-zin,* October 1991.
4) "Ergnzung zu den Illegalen OP-Codes", Peter Wtzel, *Top-M-ag-azin*, January 1992.
5) "6502 Opcodes and Quasi-Opcodes", Craig Taylor, 1992.
6) "Extra Instructions Of The 65XX Series CPU", Adam Vardy, 27 Sept. 1996

This appendix was taken verbatim from a list I was sent some time back. The formatting has changed, as well as a few opcode names. Errors are probably due to carelessness on my part, and should not reflect upon the original compilers of this document. That being said, notice that these are undocumented opcodes. They may or may not work any given emulator, and their behavior may not work as advertised even on real hardware. Use these instructions at your own risk!

ANC

AND byte with accumulator. If result is negative then carry is set.

```
Status flags: N,Z,C
Addressing: Immediate
```

ARR

AND byte with accumulator then rotate one bit right in accumulator and finally check bits 5 and 6:

- If both bits are 1: set C, clear V.
- If both bits are 0: clear C and V.
- If only bit 5 is 1: set V, clear C.
- If only bit 6 is 1: set C and V.

```
Status flags: N,V,Z,C
Addressing: Immediate
```

ATX

AND byte with accumulator, then transfer accumulator to X register.

```
Status flags: N,Z
Addressing: Immediate
```

AXS

AND X register with accumulator and store result in X register, then subtract byte from X register (without borrow).

```
Status flags: N,Z,C
Addressing: Immediate
```

AX7

AND X register with accumulator then AND result with 7 and store in memory.

```
Status flags: -
Addressing: Absolute,Y ;(Indirect),Y
```

AXE

Exact operation unknown.

```
Addressing: Immediate
```

DCP

Subtract 1 from memory (without borrow).

```
Status flags: C
Addressing: Zero Page; Zero Page,X; Absolute; Absolute,X; Absolute,Y;
(Indirect,X); (Indirect),Y
```

ISB

Increase memory by one, then subtract memory from accumulator (with borrow).

```
Status flags: N,V,Z,C
Addressing: Zero Page; Zero Page,X; Absolute; Absolute,X; Absolute,Y;
(Indirect,X); (Indirect),Y
```

JAM

Stop program counter (lock up processor).

```
Status flags: -
Addressing: implied
```

LAS

AND memory with stack pointer, transfer result to accumulator, X register and stack pointer.

```
Status flags: N,Z
Addressing: Absolute,Y
```

LAX

Load accumulator and X register with memory.

```
Status flags: N,Z
Addressing: Zero Page; Zero Page,Y; Absolute; Absolute,Y;(Indirect,X);
(Indirect),Y
```

RLA

Rotate one bit left in memory, then AND accumulator with memory.

```
Status flags: N,Z,C
```

```
Addressing : Zero Page; Zero Page,X; Absolute; Absolute,X; Absolute,Y;
(Indirect,X); (Indirect),Y
```

RRA

Rotate one bit right in memory, then add memory to accumulator (with carry).

```
Status flags: N,V,Z,C
Addressing : Zero Page; Zero Page,X; Absolute; Absolute,X; Absolute,Y;
(Indirect,X); (Indirect),Y
```

SAX

AND X register with accumulator and store result in memory.

```
Status flags: N,Z
Addressing: Zero Page;Zero Page,Y;(Indirect,X);Absolute
```

SLO

Shift left one bit in memory, then OR accumulator with memory.

```
Status flags: N,Z,C
Addressing: Zero Page; Zero Page,X; Absolute; Absolute,X; Absolute,Y
(Indirect,X); (Indirect),Y;
```

SRE

Shift right one bit in memory, then EOR accumulator with memory.

```
Status flags: N,Z,C
Addressing Zero Page; Zero Page,X; Absolute; Absolute,X; Absolute,Y;
(Indirect,X);(Indirect),Y;
```

SXA

AND X register with the high byte of the target address of the argument +1. Store the result in memory.

Status flags: -

```
Addressing: Absolute,Y
```

SYA

AND Y register with the high byte of the target address of the argument +1. Store the result in memory.

Status flags: -

```
Addressing: Absolute,X
```

XAS

AND X register with accumulator and store result in stack pointer, then AND stack pointer with the high byte of the target address of the argument +1. Store result in memory.

Status flags: -

```
Addressing: Absolute,Y
```

Appendix D: Licensing

ATasm is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

ATasm is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details, which can be found in the file LICENSE in this archive.

If you have a burning desire to use this program under a different license, please contact me.