ATasm v1.30

A mostly Mac/65 compatible 6502 cross-assembler

ATasm is a 6502 command-line cross-assembler that is compatible with the original Mac/65 macroassembler released by OSS software. Code development can now be performed using "modern" editors and compiles with lightning speed.

ATasm Features

- ATasm produces Atari native binary load object files or can optionally target .XFD/.ATR disk images and the machine state files produced by the Atari800Win emulator (version 2.5c or greater),the Atari800 emulator (version 0.9.8g or greater) or the Atari++ emulator (version 1.24 or greater)
- Conditional code generation, and code block repetition
- Rich macro support, compatible with existing Mac/65 code libraries
- Atari specific assembler directives (.SBYTE, .FLOAT, etc.) and undocumented opcodes.

ATasm runs native on IBM PCs in Windows and compiles cleanly under Linux, MacOS/X or any platform with the GNU C or Clang compiler.

All source code and the Windows binary are included in the package.

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 4, 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 email 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 several zp, y -> absolute address operators (sbc/adc/and/eor/ora/cmp); Allowed compilation 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 emitting 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.

version 1.07 – introduced .BANKNUM operator; Allow .SET 6 to forward reference labels; Allow leading underscores in label names; Fixed an error with command-line defines; Allow character

quoting of spaces and semicolons; Allow comments to start without a preceding space. Fixed local label references inside of macros or macro parameters.

Version 1.08 – Initial support of list files with -g command-line parameter; Allow .INCBIN to honor include paths; Missing state files no longer segfault; Double-forward defines now throw an error rather than silently generate bad code.

Version 1.09 - Contributions by Peter Hinz; Fixed a problem with filename creation when saving output to an ATR image; Fixed CVE-2019-19785: Stack-based buffer overflow in the to_comma() function; Fixed CVE-2019-19786: Stack-based buffer overflow in the parse_expr() function; Fixed CVE-2019-19787: Stack-based buffer overflow in the get_signed_expression() function; Compiling windows executable with Clang.

Version 1.10 - Fixes absolute address calculation in the case of lda abs,x ,if the address was undefined but used an offset like (here+1) the code would place the address into page 0. Fixed a buffer overflow in aprintf().

Version 1.11 - Fixes buffer overrun in put_float(). Changes the output format of the list command slightly to allow source level debugging in the Altirra emulator. The list file has to start with "mads ". Adds support to pass Altirra specific ;##TRACE and ;##ASSERT commands from the source code to the listing file (to be read and interpreted by Altirra).

Version 1.12 - When listing an assembly via the -g command line parameter, labels not defined on the same line as an operand where not output at all. This version outputs the label in its original name (in all caps).

Version 1.13 - Added CC65 header and assembler include file generation switches. Switch -hc dumps all equates and labels into a C-style header file. Useful when interfacing with CC65. Each #define is prefixed with the basename of the assembler file. atasm -hc test.asm will create "test.asm.h" with #define TEST_… Switch -ha dumps all equates and labels into an atasm style include file. Both -hc and -ha switches can have their output filenames specified by the switch or if left out auto-generated by atasm. -hcmy-project.h -hamy-project.inc will generate "my-project.h" and "my-project.inc"

Version 1.14 - Improved the source file and line # tracking for equate and label definitions. The -hc and -ha switches now dump detailed info on which file and on which line the definition occurred. -hc and -ha dumps sort their output by the address and not alphabetically. Added support for labels with '.' in them. cio.cmd and cio.len are now valid labels. Fixed some Linux compile warnings.

Version 1.15 - Added -hv switch to dump all equates, labels, macro defs and included source files to the 'plugin.json' file located at the root folder from where atasm starts searching for source files. This is to be used by the Atasm-Altirra-Bridge VSCode plugin to allow you to quickly jump to your code.

Added modulo/remainder operator. %% or .MOD. You can now say '.byte 15%%10' and it will store the value of 5.

Version 1.16 - Added the << and >> shift operators to the expression parser. Code like this now assembles: lda #1<<4
Fixed a bug in .LST output generator. Filenames were not tracked correctly which caused source lines to be allocated to the wrong source file. Thank you to Lars Langhans for reporting this.

Version 1.17 - Added the -hv

$$clm$$

option. This allows you to export the list of defined (c)onstants, (l)abels and (m)acos, together with all the filenames included into your project to the 'asm-symbols.json' file. This is used by the atasm-altirra-bridge VSCode plugin (https://bit.ly/3ATTHVR) to quickly navigate to parts of your code.

Version 1.18 - Extended the -hv

$$clm$$

option with an L option.
l=dumps global symbols (excludes local symbols, those defined with a ? at the beginning).
L=dumps ALL symbols (global and local)
You can use .OPT NO SYMDUMP to turn off the dumping for constants.

```
.OPT NO SYMDUMP

.INCLUDE "ANTIC.asm"

.OPT SYMDUMP
```

```
First .opt turns off constant tracking. This means all constants created from
```

then until the ".opt SYMDUMP" will NOT make it into the "asm-symbols.json" file.

Version 1.19 - Modified the * program counter command to also name a memory region.

```
\* \= $2000 "BOOT"

Will name the region starting at $2000 as "Boot", which will be

dumped after the compile for the vscode extension. Same can be

done via: .REGION "BOOT" directly after the \* command.

Added \-eval command line option to only do the compile and NOT

write anything to disc.

Slight warning and error format change to make it external parsab
```

Version 1.20 - Added the .ELSEIF directive to build easier .IF .ELSEIF .ELSE .ENDIF control blocks.

Version 1.21 - Fixed the .REF implementation. It will now detect a forward reference to a label correctly. This is useful in building libraries and excluding code from them if the functions or data is not being referenced. i.e. If there is no JSR FUNC1 then the code in the .IF block is not generated.

```
.IF .DEF func1

        func1 ...

.ENDIF
```

Version 1.22 - Added long jump commands JEQ, JNE, JPL, JMI, JCC, JCS, JVC, JVS. These macro commands are similar to the 6502 branch instructions BEQ, BNE, BPL, BMI, BCC, BCS, BVC, BVS, but can target the entire 64KB address space via a jump. If the distance is short and the target is known during the first assembler pass then the jump is converted into a branch. The assembler spits out code change suggestions if it finds jumps that could be optimized.

Version 1.24 - Added basic trigonometry functions as dot commands: .sin, .cos. These can be used in loops to generate trig tables.

Version 1.25 - Added -a / -mac65 option to autobank the code with every program counter directive. This option will create a memory bank for every program counter assignment. The banks are not sorted or combined.

```
 Added the .INIT xyz and .RUN xyz directives to set the INITAT and RUNAT locations
```

Version 1.26 - fixed the parsing of the -hvclm and -hvcLm command line parameters
-l and -s options now produce the same output. -l also includes equates (=)

Version 1.27 - Thank you to Itay Chamiel for finding an OLD bug in atasm. Basically forward references that required an extra pass to resolve did not work. The issue has now been fixed. Broken since V1.08

Version 1.28 - Fixed a parse error in a forward declared variable.
The problem is that a forward declaration was given the value of $FFFF. Issue comes in if the variable is also given that value via x = $FFFF. At that point atasm does not know if its a forward declaration or a fixed assignment.

Version 1.29 - Added .IFDEF and .IFNDEF directives. Short version of: .IF .DEF and .IF .NOT .DEF
Parsing binary numbers in MADS format %11110000 works now. But is only supported outside a macro.
Added multi-line comments using the /* ..... */ format.
Added .PROC/.ENDP for procedure definitions. This defines a local scope and a label. Useful for grouping code.

Version 1.30 - Added .GUARD/.NOTIFY directive.
Check the .GUARD section for usage details.

Chapter 1: ATasm

1.1 Installation

The normal binary distribution will include the following files:

readme.md: this file
atasm.exe: The Windows executable, compiled with visual studio in 64-bit
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
              -fvalue: set raw binary fill byte to value.
                           This value should be a number between 0-255.
                           Decimal and hexadecimal numbers are accepted.
              -m[fname]: defines template emulator state file fname.
                           If the parameter fname is not provided, ATasm will atter
                           to use the default state file "statefile.a8"
              -lfname: exports a symbol table to fname.
              -gfname: dumps debug list to file [fname]
              -xfname: saves object file to .XFD/.ATR disk image fname
              -ofname: saves object file to fname
              -Dsymbol=value: pre-defines [symbol] to [value]
              -Idirectory: search directory for .INCLUDE files
              -mae: treats local labels like the MAE assembler. See section 3.15 for more info
              -hc[fname]: dumps constants/equates and labels to CC65 header file.
              -ha[fname]: dumps constants/equates and labels to ATasm .include file.
              -hv[clmL]: dumps constants/equates, labels and macro definition info
                           for VSCode plugin, c=constants, l=global labels,
                           m=macros, L=all labels
              -eval: only compile (no binary output)
              -a OR -mac65: autobank: Put each segment in its own bank. For Mac/65 compatibili
```

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:

1. The label 'BASIC' will have a value of 1
2. The label 'OSB' will have a value of 0
3. 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 the options 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.

```
atasm -hv sample.m65
```
This will generate an "asm-symbols.json" file with JSON information describing the location of constants/equates, labels and macros in the source code. Used by the VSCode `atasm-altirra-bridge` plugin to populate the a symbol explorer. This allows you to jump to specific code points quickly.

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>] [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 '_' or '.'
- Labels may not have the same name as a 6502 opcode.
- Labels may be terminated with a ':'.
- Comments must be preceded by a ';'.
- Multi-line comments can be started with /* and end with */

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 '$'
Example: `$1, $04, $ff, $1A`
As used:

```
lda #>$601
.BYTE $02,$04,$08,$10
```

Binary constants begin with '~' or '%'
Example: `~101, ~11`
As used:

```
lda #~11110000
.BYTE ~00011000,~00111100,~01111110
```

Character constants begin with a single quote (')
Example: 'a, 'A
As used:

```
lda #'a+$10
.BYTE 'a,'B
```

Strings are enclosed in double quotes (")
Example: "Test"

As used: `.BYTE "This is a tes",'t+$80`

Decimal constants have no special prefix
Example: 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 parentheses 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.
.BANKNUM <label> Returns the current bank number of the label.
```

Group 3: Logical Not

```
.NOT  Returns true if an expression is zero
```

Group 4: Multiplication, division and modulo

```
/        division
*        multiplication
%%       modulo
```

Group 5: Addition and subtraction

```
+        addition
-        subtraction
```

Group 6: Binary operators

```
<<       binary shift left
>>       binary shift right
&        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 affect 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> ["NAME"]

This sets the origin address for assembly and optionally names the memory region.

```
Example:
* = $600
* = $2000 "boot"
* = $3000 "sprites"
* = $4000 "music"
```

This defines 4 memory areas and gives 3 of them a name. The names are dumped after the assemble.

3.3 .DS <word> or *=*+<word>

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

```
Example:
label .DS 10
```

This allocates 10 bytes of storage and assigned the "label" to point to the first byte.

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>

```
Example:
.DC 10 $FF
```

Will generate the following byte sequence: FF FF FF FF FF FF FF FF FF FF

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.

```
Example:
        .BYTE +$80,$10,20,"Testing",'a
```

Will generate the following byte sequence: 90 94 D4 E5 F3 F4 E9 EE E7 E1

.SBYTE [+],<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.

```
Example:
  .SBYTE +$80,$10,20,"Testing",'a
```

Will generate the follow byte sequence: 90 94 D4 B4 F3 F4 E9 EE E7 E1

.CBYTE [+],<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.

```
Example:
.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.

```
Example:
  .DBYTE $1234,-1,1
```

Will generate: 12 34 FF FF 00 01

.WORD

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

```
Example:
  .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.

```
Example:
  .FLOAT 3.14156295,-2.718281828
```

Will generate: 40 03 14 15 62 95 C0 27 18 28 18 28

3.9 .IF <expression>,.ELSEIF <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 .ELSEIF, .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.

```
Example:

.IF TARGET=1
....
.ELSEIF TARGET=2
....
.ELSE
....
.ENDIF
```

You can combine various directives to make symbol checks:

```
.IF .DEF SOUND
  .... some code for when SOUND is defined
.ELSE
  .... some code if SOUND is not defined
.ENDIF

.IFDEF SOUND
        ....
.ENDIF
```

You can also use a .NOT to check if a symbol is not defined.

```
.IF .NOT .DEF SOUND
        ... code for when SOUND is not defined
.ENDIF

.IFNDEF SOUND
        ... same as above, just a bit shorter
.ENDIF
```

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 a full path if desired.

3.11 .INCBIN <filename>

This includes the contents of a 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 from 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).

There is an option that can be used in combination with the -hv command line parameter. If the -hv command line argument is used then ATasm will dump constants, labels, macros and included file info to a file ("asm-symbols.json"). This data is used by a Visual Studio Code plugin (https://bit.ly/3ATTHVR) ("Atasm-Altirra-Bridge") to allow you to quickly find info and navigate to the definition. When .including hardware definition data lots of constants are defined, most of them are not used by your program and would just clutter the symbol dump. You can use a .opt directive to turn the tracing of constant definitions on and off. By default constant tracing is ON.

i.e.

```
.OPT NO SYMDUMP
.INCLUDE "ANTIC.asm"
.OPT SYMDUMP
FIND_ME = $2000
```

In the above example all constants defined in the ANTIC.asm file will not be dumped to the "asm-symbols.json" file. But "FIND_ME" will be dumped.

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 in the symbol table dump at the end of the assembly processes.

If the -mae option is specified on the command line, ATasm operates in a special mode in which the .LOCAL directive is ignored. Instead, every span between two regular (i.e. non-local) labels is automatically defined as a local region. As with regular local regions, a local variable may only be referenced from within that region. In this mode the assembler internally creates unique names for local labels by appending the local onto the end of the previous global label. For example, a local label '?LP' following a regular label 'DELAY' will be named 'DELAY?LP'. In fact, you can code a direct reference to the label DELAY?LP if you need to access the local from outside its region. Note that in

contrast to regular mode, these 'composite' label names will appear in the -hvl dump output as if they were global.

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 parentheses. 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.

```
Example:
.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.

```
Example:
.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 lookup 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 lookup 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 [<word>,<word>]

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 exactly 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"
```

The .BANK directive can take two optional parameters indicating what bank should be used, and what bank should be reported by the .BANKNUM operator. If you wish to return to a previously used bank 0 (and also append the code in the .obj file), use the following:

```
.bank 0,0
```

If you wish to split the .obj file, but have the .BANKNUM operator report as bank 0, use the following:

```
.bank ,0
```

Note: When the -a / -mac65 (autobank) option is used, the .BANK directive is disabled, and will cause an assembly error. This is because manual banking doesn't mix with automatic per-segment banking. Basically, when -a/-mac65 is in use, each *= acts as though it had a .BANK (with no arguments) just before it. The advantage of autobank is that it makes Atasm emit the segments in the order they occur in the source, which makes Atasm more compatible with Mac/65.

3.20 .ALIGN boundary

This directive aligns the current location to a specified boundary. The boundary value has to be a power of 2. In effect this is a shortcut for something like page alignment:

```
*=(*+$FF) & $FF00       ; Make sure that the next byte is placed
                        ; on the next page boundary
.ALIGN $100
```

3.21 .NAME

This directive can be used to give a memory region a name. This is very useful in the Visual Studio Code extension that interfaces with ATasm.

```
* = $2000 "Booting"
OR
.NAME "Booting"
```

After the assembly stage the assembler will output a memory map.
i.e.

```
Memory Map
----------
$2000-$20AB Booting
$2100-$21FF     Sprite data
```

3.22 JEQ, JNE, JPL, JMI, JCC, JCS, JVC, JVS (long branches)

These macro commands are similar to the 6502 branch instructions BEQ, BNE, BPL, BMI, BCC, BCS, BVC, BVS, but can target the entire 64KB address space via a jump.

```
jne dest   ->  beq #3
           ->  jmp dest
```

If the distance is short and the target is known during the first assembler pass then the jump is converted into a branch.

The assembler spits out code change suggestions if it finds jumps that could be optimized to branches.

i.e.

```
Possible long jump optimizations
================================
tests/long.asm @ 19     jeq known --> BEQ known ; distance is -13
tests/long.asm @ 20     jne known --> BNE known ; distance is -15
tests/long.asm @ 21     jpl known --> BPL known ; distance is -17
tests/long.asm @ 30     jeq forward --> BEQ forward
        Now:    BNE $03 ; distance is 83
                JMP forward
```

3.23 Trigonometry value generators

The basic sin, cos functions are very useful when generating tables for fast 3D graphics or other trigonometry based operations. Atasm has the ability to generate SIN and COS values.

The value generation is implemented as dot functions. I.e. .SIN, .COS where the parameters determine the angle, by how many steps there are in a circle and by how much the sin/cos value should be scaled.

A circle has 360 degrees (2x PI radians) but we normally don't want to work in 360 degrees. The Steps parameter sets the number of degrees a circle has. The Angle parameter indicates how far around the circle the value is, relative to the steps. SIN and COS output a value range of [-1 ... 1], the Scale parameter is used to change that to your required range.

By default .SIN and .COS generate 16-signed values. On optional parameter and the beginning of the command lets you select the high or low byte of the 16-bit value.

The general format of a trigonometric function is:

```
.SIN [Optional high/low byte/word], Angle, Steps, Scale
```

Where the first parameter selects what size output will be generated:

```
< Low byte of 16-bit value
> High byte of 16-bit value
```

Some examples:

1. Create 4 sin values showing the full range. 256 steps, scale by 4096 ($1000)

```
.SIN 0, 256, $1000 ; Should give 0 = $0000
.SIN 64, 256, $1000 ; Should give 4096 = $1000
.SIN 128, 256, $1000 ; Should give 0 = $0000
.SIN 192, 256, $1000 ; Should give -4096 = $F000
This will produce the following bytes (note 16-bit LSB MSB)
00 00 00 10 00 00 00 F0
```

2. Create a 16-bit SIN table. 256 steps, scale by 4096 ($1000)

```
*=$2000 ; Locate data @ $2000
SCALE=4096
angle .= 0
.rept 256
.sin angle, 256, SCALE
                angle .= angle + 1
.endr
```

This will produce the following bytes (note 16-bit LSB MSB)
Source: tests/sin.asm

```
18 2000  00 00 .sin angle,256,SCALE ;(0,256,4096)=        0 0x0000
18 2002  64 00 .sin angle,256,SCALE ;(1,256,4096)=        100 0x0064
18 2004  C8 00 .sin angle,256,SCALE ;(2,256,4096)=        200 0x00C8
18 2006  2D 01 .sin angle,256,SCALE ;(3,256,4096)=        301 0x012D
18 2008  91 01 .sin angle,256,SCALE ;(4,256,4096)=        401 0x0191
18 200A  F5 01 .sin angle,256,SCALE ;(5,256,4096)=        501 0x01F5
18 200C  59 02 .sin angle,256,SCALE ;(6,256,4096)=        601 0x0259
. . .
18 21F8  6F FE .sin angle,256,SCALE ;(252,256,4096)=        -401 0xFE6F
18 21FA  D3 FE .sin angle,256,SCALE ;(253,256,4096)=        -301 0xFED3
18 21FC  38 FF .sin angle,256,SCALE ;(254,256,4096)=        -200 0xFF38
18 21FE  9C FF .sin angle,256,SCALE ;(255,256,4096)=        -100 0xFF9C
```

3. Create a 16-bit SIN and COS table. 256 steps, scale by 4096 ($1000)

COS is offset from SIN by 90 degrees, so starts at 90/360 * 256 = 64.

```
*=$2000 ; Locate data @ $2000
SCALE=4096
angle .= 0
.rept 256+64
.sin angle,256,SCALE
angle .= angle + 1
.endr
```

This will produce the following bytes (note 16-bit LSB MSB)

```
18 2000  00 00  .sin angle,256,SCALE ;(0,256,4096)=        0 0x0000
18 2002  64 00  .sin angle,256,SCALE ;(1,256,4096)=        100 0x0064
18 2004  C8 00  .sin angle,256,SCALE ;(2,256,4096)=        200 0x00C8
18 2006  2D 01  .sin angle,256,SCALE ;(3,256,4096)=        301 0x012D
18 2008  91 01  .sin angle,256,SCALE ;(4,256,4096)=        401 0x0191
. . .
18 2276  E1 0F  .sin angle,256,SCALE ;(315,256,4096)=        4065 0x0FE1
18 2278  EC 0F  .sin angle,256,SCALE ;(316,256,4096)=        4076 0x0FEC
18 227A  F4 0F  .sin angle,256,SCALE ;(317,256,4096)=        4084 0x0FF4
18 227C  FB 0F  .sin angle,256,SCALE ;(318,256,4096)=        4091 0x0FFB
18 227E  FE 0F  .sin angle,256,SCALE ;(319,256,4096)=        4094 0x0FFE
```

4. Here is a sample for a combined SIN/COS table with Low (LSB) and High (MSB) split into their own tables.

```
*=$2000 ; Locate data @ $2000
SCALE=4096
angle .= 0
.rept 256+64
        .sin <,angle,256,SCALE
        angle .= angle + 1
.endr
angle .= 0
.rept 256+64
        .sin >,angle,256,SCALE
        angle .= angle + 1
.endr
```

3.24 RUN and INIT address specifiers

There are two ways to get your code to INIT and BOOT in Atasm. Below is the same code showing the two different methods:

Option 1: Using banks

```
        .BANK   ; Put the following code into a memory bank
        * = $8000
INIT:
        LDA #0
        STA 710
        RTS

        .BANK   ; Start a new bank for INITAT.  During load this will be executed first
        * = $2e2
        .WORD INIT

        *=  $8000       ; Put the rest of the code in another bank
CYCLE:
        LDA 20
        STA 709
        JMP CYCLE

        .BANK           ; Tell Atari to run this code once the file is loaded
        * = $2e0
        .WORD CYCLE
```

Option 2: Using .RUN and .INIT

```
        .BANK
        * =  $8000
INIT:
        LDA #0
        STA 710
        RTS

        .INIT INIT

        .BANK
        * =  $8000
CYCLE:
        LDA 20
        STA 709
        JMP CYCLE

        .RUN CYCLE
```

3.25 .PROC/.ENDP Procedure definition

A procedure/function in assembler would look something like this:

```
        .local
TheName:
        ; some code
?loop: ; Local definition of a label

        ; Exit from procedure/function
        rts
```

The .PROC/.ENDP directive can be used to define the start and logical end points of a procedure. i.e.

```
.PROC TheName
        ; some code
?loop: ; Local definition of a label

        ; Exit from procedure/function
        rts
.ENDP
```

`TheName` would be defined as a label and .ENDP would indicate the logical end of the procedure.

Note that .PROC/.ENDP is just syntactic sugar but it does make your code more readable and it does help the 'atasm-altirra-bridge' VSCode plugin to better interact with your code.

3.25 .GUARD/.NOTIFY report on assembler conditions

When assembling your code you want to make sure that the memory area you are putting variables into, or allocating to your code, does not overflow. This can easily be done with some directive magic.

```
.IF [*>=$100]
  .ERROR "Allocated too many variabled"
.ENDIF

.IF [*<$5000]
  .WARN "Still have space left"
.ENDIF
```

But the syntax is clumsy. Hence the introduction of the *.GUARD* and *.NOTIFY* directives.

.GUARD condition, "Error message", parameter expressions, ...

The *.GUARD* can be used to stop the assembly process if a specific condition fails (evaluates to 0). You want all guard condition to be true for a successful assembly.

i.e. You allocate variables into the top of Page 0 and over allocate. That would mean that variables are now in Page 1. You can test for that with the following *.GUARD* directive.

```
        * = $80
var1 .ds 1
var2 .ds 1
        ...
alice   .ds 1 ; <-- this is the last byte in Page 0. Any thing else will overflow into Page 1
bob     .ds 2

        .GUARD [* <= $100], "Memory {{%1}}-{{*}} overflow by {{%2}} bytes", var1, [*-$100]
```

.GUARD takes a list of parameters of the form:
.GUARD condition, "message", [param1, param2, ...]

- The first parameter is a condition that needs to **false** before the rest of the guard is processed.
- The second parameter is a message string.
  Parts of the message can be replaced by parameters that follow the message.
  - `{{*}}` will be replaced with the current program counter (or the location where the next byte will be put).
  - `{{%1}}` is the value of the 1st parameter after the message.
  - `{{%n}}` where n is a number from 1 to n. *n* is unlimited, but don't be silly 😀
- Each of the message parameters will be evaluated into a number and that number can be used in the error message.

NOTE: It is a advisable to put your conditions and expressions into square brackets: *[* and *]*. This makes sure that the evalutation order is correct.

```
.GUARD [*<$5000], "Code is too big!"
.GUARD [*<$5000], "Code is too big, now at {{*}}!"
.GUARD [*<$5000], "Code is too big, {{%1}}bytes over the limit!", [* - $5000 + 1]
```

.NOTIFY condition, "message", [param1, param2, ...]

.NOTIFY takes a list of parameters of the form:
.NOTIFY condition, "message", [param1, param2, ...]

- The first parameter is a condition that needs to be **true** before the rest of the directive is processed.
- The second parameter is a message string.
  Parts of the message can be replaced by parameters.
  - `{{*}}` will be replaced with the current program counter (or the location where the next byte will be put).

    - `{{%1}}` is the value of the 1st parameter after the message.
    - `{{%n}}` where n is a number from 1 to ....
- Each of the message parameters will be evaluated into a number and that number can be used in the message.

NOTE: It is a advisable to put your conditions and expressions into square brackets: *[* and *]*. This makes sure that the evalutation order is correct.

```
.NOTIFY 0, "This will never print"
.NOTIFY 1, "This will always print"
.NOTIFY [*>$5000], "This will print when the PC is over $5000, now {{*}}"
.NOTIFY [[VAR1-VAR2] > $100], "Variables are too far apart: {{%1}}", [VAR1-VAR2]
```

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 to convert tokenized files to ATASCII (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.

- Comment lines only begin with ';' not with '*'
- Comment blocks are supported via /* ... */
- The character '|' can be used in place of '!' as a binary OR
- Macros can have an arbitrary number of parameters and can be nested arbitrarily deep during invocation.
- .INCLUDEs can be arbitrarily nested.
- There are an unlimited number of .LOCAL regions (well, 65536 of them)
- Macro names must be unique and cannot be the same as an existing label
- Macro parameters can be separated by commas or by spaces
- .END,.PAGE,.TAB,.TITLE, and most .SET directives are ignored
- Extra directives .DC, .DS, .INCBIN, .WARN, .REPT/.ENDR, .IFDEF, .IFNDEF have been added
- .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.
- Operands are reserved words and cannot be used as labels or equates.
- Operator precedence of unary > and < are given proper precedence. For instance, Mac/65 will treat #>1000+2000 as #>(1000+2000), not #(>1000)+2000 as ATasm does. This appears to be a bug in Mac/65.

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 over two decades, 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). Well at least the '.'s in labels has been fixed.

When programming in 6502 assembly language, I actually tend not to heavily use macros, conditional assembly or many of the other 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.

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
- Peter Hinz

Bug Reports and Feature requests:

- Cow Claygil
- Peter Dell
- Peter Fredrick
- Peter Hinz

- Doug Hodson
- Dan Horak
- Tom Hunt
- Chris Hutt
- Manuel Polik
- Carsten Stroten
- Thompsen
- Greg Troutman
- B. Watson
- Itay Chamiel
- ... 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 - Set 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.

```
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.

```
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.

```
LDA  $FE90,Y
```

Accumulator: The operand is the accumulator.

```
LSR  A
LSR (an alternate form)
```

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

```
LDA  #$07
```

Implied: The operands are indicated in the mnemonic.

```
CLC
```

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

```
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.

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

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

```
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.

```
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.

```
LDX  $2A,Y
```

Note: Although technically the opcodes LDA $20, YandSTA20,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.

```
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.

```
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-Magazin, October 1991.

4. "Ergaenzung zu den Illegalen OP-Codes", Peter Wtzel, Top-Magazin, 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