CASM

A Commodore 64 Assembler for the IBM PC

Version 1.3

Written by Peter Gonzalez, 1995

pbgonz@mail.wm.edu

NOTES FROM THE AUTHOR

Getting started

There are a few people who always read the manuals from cover to cover before getting started, but if you're the type who would be interested in a Commodore 64 assembler in the first place, I'm sure you are not among that number. So if you must, go test it out, run the demos, tinker until you get stuck, and I'll see you in a few minutes.

* * *

Great! Now let's get started.

Using This Manual

I've divided this document into lots of little sections so you can use it as a reference, but I've also kept it short so that you could easily read it all at once. Also, I've stuck mostly to examples to minimize the boring rhetoric. If you plan to do any serious programming with CASM, you should read the manual in its entirety before getting started, since it will probably save you a lot of time and silly mistakes. The language is pretty intuitive, but there are some nice features you don't know about.

I will assume that the reader is familiar with the 6502 assembly language as implemented on the popular Turbo Assembler which runs on the Commodore 64. This manual provides a description of the CASM language structure, other miscellaneous information, and explanations for the ancillary utilities CDASM, CDUMP, and CLINK. Sorry, no documentation is available for the source code, and the source code is only available directly from me.

Isn't the Commodore 64 Obsolete?

Although the Commodore 64 may be by today's standards "obsolete," it is certainly not useless. For the beginner, the simple 6502 assembly language provides a great introduction to computer architecture and hardware. For the hobbyist, its compact and simple design makes it easy to switch over to a battery supply and box up in a homemade case. And yet it has 64K of RAM, a fancy 3-voice sound synthesizer, and a full NTSC- or PAL-compatible video chip with programmable characters, single-bit precision scrolling, and 8 movable object blocks.

For the hacker, the hardware is replete with undocumented features to discover, and the entire CPU bus is available via the user expansion slot in the back. And, of course, for the gamer/nostalgist, there are literally thousands of free classic video games available on the Internet, along with two excellent emulators for the PC to save you the trip to the attic. Best of all, the price has dropped significantly since the advent of the Pentium.

My Motivations

I probably fall somewhere in the hacker / nostalgist range. I'll be honest. I needed the assembler to help me realize my childhood dream of writing a Commodore 64 video game. This is really embarrassing.

New Features with Version 1.3

Version 1.3 has some significant improvements over Version 1.1b which was released earlier this month. A binary bitmap feature has been added for DATA blocks which simplifies sprite and character set storage. Most importantly, it runs a lot faster. I switched the data structures over from doubly linked-lists to a fast hash / b-tree structure,[1] and the tokenizer was completely rewritten. And, of course, all of the annoying lock-ups were ironed out.

[1] Unfortunately, this caused the undesirable side-effect that the block listings generated by CASM's "/D" command-line option are sometimes out of order, and the quirk probably won't be fixed at least until the next release.

INTRODUCTION

Just what is CASM?

CASM is a cross-assembler which runs on an IBM PC and generates code for a 6502 processor. While CASM is tailored for writing Commodore 64 programs, it should also work for any 6502-based microcomputer, including the popular Apple II and Atari 800 systems.

The 6502 is a simple CPU, and the CASM language is likewise simple. This is not a bad thing, however, because size and efficiency are primary considerations when writing anything serious for a 1MHz / 64K system, and fancy assembler features tend to remove important things like clock cycles from the programmer's thought process. Although Turbo Assembler shares many of CASM's features, it chokes up on large projects and is not recommended for the impatient. (Of course these are limits of the hardware, and not the software.)

Many who (like myself) stood stalwartly by their '64 while their friends "sold out" to IBM will probably object to the idea of a Commodore assembler for the IBM. But before you get too worked up, play around with it a bit. Give it a chance. The CASM / emulator environment takes a little getting used to, but for big projects,[2] it is a *major* improvement. I've spent many long hours punching code into the ole '64, and I know.

Using a Cross-assembler

Since CASM is a cross-assembler, the problem of how to run the output programs immediately presents itself. There are three ways to do this. The easiest, and fastest, is to get one of the popular emulators[3] and run your programs on the PC. Once you get your code working, though, you'll want to see it run on the real thing. The simplest way to accomplish this is to upload the programs via modem to the Commodore 64. If you don't have a Commodore modem (or would like to wait less than an hour to see the results), most of the emulators will allow you to connect a 1541 drive to your PC's parallel port. This is much faster, but usually requires registering the shareware.

[2] In fact, CASM is conspicuously designed for large projects that would be unmanageable to code by hand.

[3] I personally recommend C64S, written by Miha Peternel and distributed by Seattle Lab. A shareware version of C64S is available on the Seattle Lab's World Wide Web page at: `http://www.seattlelab.com`

GENERAL SYNTAX

This section describes the basics of how the assembler reads text from a file. It explains the syntax for comments, numbers, and text strings, and defines some terms that will be used in the rest of the manual.

The Tokenizer Preprocesses .ASM Files

CASM is presented with an .ASM file to assemble. At the lowest level, CASM sees this file as a collection of tokens separated by "whitespace" (spaces, tabs, etc.). The rules for what make up a "token" are primarily an implementation detail, but it is important to note that the intelligent portion of the assembler — the portion which will be communicating error messages — cannot see the carriage returns, comments, or file boundaries. Another portion, the "tokenizer" part, is what parses the .ASM file and hands the tokens (and corresponding source filename / line number) up to the actual assembler. So when an error is reported on a certain file or line, the error is often in the *vicinity* of that line.

Line and Block Comments

Both line comments and block comments (informational notes which are invisible to the assembler) may be placed in the .ASM file. Any time a semicolon(`;`) followed *directly* by a hyphen(−) is encountered in the file, the tokenizer ignores everything until the next hyphen-semicolon pair. If the initial semicolon is not followed directly by a hyphen, everything until the next carriage return is skipped. For example:

```
;-
  Here is a block comment,
  which can span multiple lines.
-;
   ldx #$00
   sta $0400,x        ; this is a line comment
   dex

;------------------------------------------------;
;  Programmers often do artistic things like this.  ;
;------------------------------------------------;
```

Case Sensitivity

It should be noted that the assembler is case-insensitive. For example, CASM will not distinguish between the identifiers "`MainProgram`" and "`mainprogram`".

Number Formats

CASM recognizes 8-bit numbers ("bytes") and 16-bit numbers ("words"). These numbers may be presented in binary, decimal, hexadecimal, or as an ASCII / screen code character.

Binary

Binary numbers are preceded by a percent sign(`%`). A binary byte is composed of eight ones and zeros written with out spaces between the digits (e.g. "`%11001101`"). A binary word is a binary byte followed by a period(`.`) and then another eight ones and zeros (e.g. "`%11001101.01101101`").

Decimal

Decimal bytes can range from 0 to 255, written with out spaces between the digits. Decimal words can range from 0 to 65535. Words less than 256 should be padded with leading zeros to make four digits. (e.g. "`0014`" is a decimal word, but "`254`" is a decimal byte.)

Hexadecimal

Hexadecimal numbers are preceded by a dollar sign(`$`), and are composed of the digits `0-9` and `a-f` or `A-F`. (CASM is case-insensitive.) For example, "`$1A`" would be a hexadecimal byte, and "`$FFC2`" would be a hexadecimal word.

ASCII / Screen Code Characters

A byte may also be presented as a single ASCII or screen code character. The character will be interpreted as a byte with the value of its corresponding ASCII / screen code value (see the next section, *Text String Formats*). ASCII characters are placed in double quotes(`"`), and screen codes are placed in single quotes(`'`). Here are two examples:

```
lda #"A"
jsr $ffd2    ; write the ASCII letter "A" using kernel routine

lda #'a'
sta $0400    ; store the screen code for 'a' in the video memory
```

It is legal to enclose a single quote in single quotes (e.g. " `lda #'''` ") or a double quote in double quotes (e.g. " `lda #"""` ") when it represents a byte in a `SUB`; however, this is illegal with strings in a `DATA` block (e.g. " `'Here's a problem'` ").

Text String Formats

CASM distinguishes two types of text strings. ASCII[4] text is what is used by BASIC strings and the kernel routines. It supports control codes for clearing the screen, changing colors, etc. Screen codes are used in the video buffer, and each code corresponds directly to an element

[4] Actually, the Commodore has its own "PET-ASCII," which differs somewhat from standard ASCII. At the moment, CASM interprets double-quoted strings as standard ASCII text, which works for the most part. In practice, assembly language programs almost always use screen codes.

of the current character set. In CASM, ASCII text strings are enclosed in double quotes("), and screen codes are enclosed in single quotes(').

Since the IBM keyboard is not capable of creating all of the Commodore's graphics symbols, only a subset of the screen codes are possible in single quotes. (The others can be specified as numbers or with aliases.) The numerical digits 0-9 and alphabetical characters a-z and A-Z are converted to their expected screen codes, as are the following symbols:
`@[]!#$%&\()*+,-./:;<=>?`. The Table 2-1 lists the unusual IBM symbols which are mapped to Commodore symbols. Any other IBM characters are converted to their ASCII equivalents.

IBM	*C64*	*Hex*
\	British pound	1C
^	up arrow	1E
`	left arrow	1F
|	vertical line	5D
~	overscore	63
_	underscore	64

Table 1-1: Screen Code Equivalents

Identifiers

"Identifiers" are used for `SUB`, `DATA`, and `LAYOUT` section names, aliases, and line labels. They must start with an alphabetical character or underscore(_) and the following characters can be alphabetical characters, underscores, or numerical digits. Identifiers are limited in length to 32 characters.

THE ASSEMBLER

This section describes the higher-level aspects of the CASM language. .ASM files are made up of four general components:

- byte- and word-sized aliases
- SUB blocks
- DATA blocks
- a LAYOUT section

All are optional except the LAYOUT section, which is mandatory and which must come first in the file. These four components will be explained in order.

Aliases

Aliases are identifiers which can represent an 8-bit number (byte) or 16-bit number (word). When used in a SUB or DATA block, an alias behaves as if the number the alias represents were inserted in place of the identifier.

Aliases can only be defined outside of SUB and DATA blocks, and must appear *before* the blocks that use them. Thus, they are usually placed at the beginning, immediately after the LAYOUT section. A byte alias is defined with the keyword ALIASB followed by the identifier, an equals sign(=) and an 8-bit number (in hexadecimal, decimal, or binary) or ASCII character or screen code. Here are some examples:[5]

```
ALIASB Byte           = 123
ALIASW WriteChar      = $FFD2
ALIASB ASCIICode      = "a"
ALIASW BinaryNum      = %00001111.00001111
```

An alias is used in a SUB and DATA block as if it were the number. For example:

```
lda #ASCIICode        ; put "a" in the accumulator
jsr WriteChar         ; use kernel routine to write it
```

SUB Blocks

SUB blocks are probably the most commonly used blocks. They begin with the keyword SUB followed by an identifier, a list of assembly language instructions and line label definitions, and are terminated with the keyword ENDS. While SUB blocks are typically used to define subroutines, remember that the "return from subroutine" (RTS) instruction is *not* automatically appended.

[5] **Note:** The examples given in this section will not assemble without the proper LAYOUT sections.

The standard 6502 instruction set[6] is supported with the usual mnemonics, except that when operating on the accumulator, ASL, LSR, ROL, and ROR must be followed with the parameter "ACC". Also, the dummy instructions DAT and DATW direct the assembler to store their parameter, a byte or word respectively, at that location. For compatibility with Turbo Assembler, BYT is a synonym for DAT.

Line Labels

Line labels are defined by placing the identifier followed by a colon where an instruction would normally start. For example:

```
ALIASW JoystickPort1  = $DC00

SUB TestSub
         lda JoystickPort1
         lsr acc             ; test for up
         bcs NotUp
         dec $d001           ; move sprite up
NotUp:
         lsr acc             ; test for down
         bcs NotDown
         inc $d001           ; move sprite down
NotDown:
         dat $60             ; fancy way of writing "rts", since $60
                             ; is the assembled code for "rts"
ENDS
```

Line labels are local to the block in which they are defined so that label names may be reused in other blocks. To access a label in another block, preface the label with the name of the block in which the label is defined, followed by the pipe symbol(|). Block names may also be used as labels, and refer as such to the beginning of that block. For example:

```
SUB Test1
         lda #$00
label:   rts
ENDS

SUB Test2
         jsr Test1           ; jsr to the first line of Test1
         jmp Test1|label     ; jmp to the second line of Test1
ENDS
```

Address Arithmetic

Advanced programs often need to access high or low bytes of label addresses or offsets from labels. CASM supports four operators for calculating these: +, −, <, and >.

< or > can come immediately after any instruction that takes a word as a parameter. < takes the LSB and > takes the MSB. For example:

```
lda  <$1234     ; same as lda  $12
lda #>$1234     ; same as lda #$34
```

[6] Support for the nonstandard instructions (e.g. SLO) may be included in a later version, if someone would like to take the liberty of typing them up in a table like the one in Appendix B and sending them to me.

Here is a common usage:

```
ALIASW ScreenStart    = $0400      ; Start of the text screen memory

SUB ClearScreen
         lda #<ScreenStart
         sta $fb
         lda #>ScreenStart
         sta $fc

         lda #'a'
         ldy #$00
         sta ($fb),y
ENDS
```

Which would be assembled (with the proper `LAYOUT` section) to this:

```
         LDA #$00
         STA $FB
         LDA #$04
         STA $FC
         LDA $01
         LDY #$00
         STA ($FB),Y
```

The + and − operators are used to calculate offsets from a label, as well as to perform simple arithmetic. They may be used with either aliases or numeric constants, but if a label is present, it must come first. For example:

```
         lda #$23 + 1
         lda #'z' - 'a' + 1
         sta alias + 1
         sta <$1000 + $123 - %00000010
         jmp label3 - 2
```

Or, more practically, something like this:

```
         lda <label + alias - 1
         sta $fb
         lda >label + alias - 1
         sta $fc
```

DATA Blocks

`DATA` blocks are similar to `SUB` blocks, but are used to store everything besides program code. They are typically used for text messages, sprites, character sets, and tables. A `DATA` block begins with the keyword `DATA`, followed by an identifier, a list of bytes, words, text strings, "binary bitmaps", labels, and label references, and it is terminated with the keyword `ENDD`.

Bytes, Words, and Text Strings

The bytes and words can be in any base. *Note that words are stored in the regular "little-endian" form, with LSB before MSB.* Here is an example:

```
DATA TestData
         1                          ; decimal byte
         $02                        ; hexadecimal byte
         %00000011                  ; binary byte

         1284                       ; decimal word
         $0706                      ; hexadecimal word
         %00001001.00001000         ; binary word

         "Hello"                    ; ASCII string
         'Hello'                    ; Screen string
ENDD
```

From the above, CASM would produce:

```
$0000:   $01 $02 $03 $04 $05 $06 $07 $08
$0008:   $09 $48 $65 $6C $6C $6F $48 $05
$0010:   $0C $0C $0F
```

Binary Bitmaps

Binary bitmaps are a new feature with CASM Version 1.3. They simplify the storage of character sets, sprites, and other binary-encoded graphic images. Between two brackets (`[,]`), any multiple of eight bits may be entered. (The brackets are required to make it easier to track down missing or extraneous bits in large bitmaps.) An "off" bit is represented with a period(`.`), and an "on" bit with `x`, `X`, or an asterisk(`*`). (The three options are provided to allow for different editor fonts and user preferences.) The assembler then combines every eight successive bits into a binary byte and stores it.

For example,

```
DATA TestChar
         [.XXXXXX.]
         [X......X]
         [X.X..X.X]
         [X......X]
         [X.XXXX.X]
         [X..XX..X]
         [X......X]
         [.XXXXXX.]

; a label definition (see next section)
ColorField:

         [****....****....****....]   0
ENDD
```

This would be assembled to produce:

```
$0000:   $7E $81 $A5 $81 $BD $99 $81 $7E
$0008:   $F0 $F0 $F0 $00
```

Labels and Label References

Labels are defined within `DATA` blocks just as they are in `SUB` blocks: the label name is given followed by a colon(`:`), as in the above example. Label references are made simply by entering the name of the label. The actual address of the label is then stored as a word in little-

endian form. For example, if placed at $0801, this DATA block would generate the BASIC code to jump to the hexadecimal address $0810:

```
DATA SysStart

; This generates a BASIC stub to run the program.  It should go at the
; beginning of the LAYOUT block, and the load address should be $0801.
; The start address for the program should be $0810.

; The BASIC program is "1995 SYS(2064)"

         EndOfLine    ; pointer to where next BASIC line begins
         1995         ; line number
         $9E          ; BASIC "SYS" instruction
         "(2064)"
         0            ; End of line
EndOfLine:
         0 0          ; End of BASIC program
ENDD
```

With the appropriate LAYOUT section, the assembler would generate the following bytes. Note that the address of the label EndOfLine ($080D) is stored in the file where the label is referenced.

```
$0801:   $0D $08 $CB $07 $9E $28 $32 $30
$0809:   $36 $34 $29 $00 $00 $00
```

The LAYOUT Section

As may be apparent by now, the LAYOUT section appears first in the file, and determines the placement of the blocks in the final object code. It begins with the keyword LAYOUT, contains a listing of blocks, and ends with the ENDL keyword. The first block in the list must be preceded by the actual address where it will start (in decimal or hexadecimal), which will become the load address for the file. The remaining blocks may optionally have start addresses, any extra intermediate space will be filled with zeros. No block may appear twice in the LAYOUT section.

Here is a small example:

```
LAYOUT
$0800:   Data2        ; $0800 becomes the load address for program
         Code1
$0808:   Data3        ; optional address $0808
ENDL

SUB Code1
         dat 1
         dat 2
         dat 3
ENDS

DATA Data2
         4 5 6
ENDD

DATA Data3
         7 8 9
ENDD
```

And here is the assembler output (note the zero-fill up to $0808):

```
$0800:   $04 $05 $06 $01 $02 $03 $00 $00
$0808:   $07 $08 $09
```

Appendix A contains the complete listing of a real program written with CASM.

OTHER FEATURES

Creating Libraries

The tokenizer also understands a directive which enables the programmer to create libraries of routines to be shared by separate projects. When the tokenizer encounters the at-sign(@), it reads the following filename and then opens the corresponding .LIB file and begins reading tokens from it. For example, consider the following code:

```
LAYOUT
$0801:          SYSStart     ; BASIC "SYS(2064)" start
                MainPrg
$2000:          SpriteData
ENDL

@SOUND                       ; Use the SOUND library

DATA SYSStart
   .
   .
   .
```

When the tokenizer reaches the "@SOUND" directive, it will attempt to open a file called SOUND.LIB. To the assembler, it will appear as if the contents of SOUND.LIB had been pasted into the .ASM file at that point. Hence, .LIB files cannot contain LAYOUT sections. (This is the justification for the .LIB file extension.)

.LIB files may include other .LIB files with the @ directive; however, no special provision has been made for recursion, so never allow a .LIB file to directly or indirectly reference itself. With the DOS SHARE.EXE loaded, this will give you an "Unable to open file" error. Without SHARE.EXE, CASM will most likely run out of memory or lock up.

Most beginning programmers don't use libraries, but as you find yourself repeatedly writing certain routines over and over again, you'll probably start thinking about putting together a library. The library feature is particularly attractive because CASM only assembles the routines you reference in the LAYOUT section. In contrast, many other assemblers just copy the entire library, which is fast but wasteful. In addition, CASM's approach also allows the programmer to choose the order in which the routines are placed.

In practice, CASM's library feature is most useful for creating large collections of aliases, such as VIC chip or SID chip addresses.

THE UTILITIES

CASM comes with three supporting utilities:

- CDASM.EXE - a disassembler which can convert executable programs back into CASM source code
- CDUMP.EXE - a dump utility which can display data in a variety of formats
- CLINK.EXE - a multi-module linker, which can be used to modularize large programs

This section explains the command-line options for CASM and these utilities.

CASM <SourceFile> [/D] [/N] [/Q]

CASM.EXE assembles **<SourceFile>** which contains the source code, and generates a corresponding .PRG file containing the executable code. If the file extension for <SourceFile> is omitted, it is assumed to be .ASM.

The following command-line options are available:

Option	*Description*
/D	Requests an accompanying disassembly showing the blocks that were assembled. This is useful for debugging, large projects, since it shows where the SUB and DATA blocks begin and end.
/N	Prevents the load address from being stored at the beginning of the file. Commodore 64 disk drives store the load address of a program as the first two bytes of the file. Other 6502-based systems may not adhere to this format.
/Q	Runs CASM in "quiet" mode, where less text is displayed on the screen. This is used when CLINK.EXE runs CASM.EXE.

Table 1-2: CASM.EXE Command-line Options

CDASM <SourceFile> [/A] [/B] [/L] [/N] [/R]

CDASM.EXE disassembles a **<SourceFile>**, writing the output to the screen. (If no file extension is given, .PRG is assumed.) The screen output may be redirected using the regular DOS redirection operators. For example, to disassemble the file GAME.PRG with labels and store the output in the file GAME.ASM, one might type the following:

```
CDASM GAME.PRG /L > GAME.ASM
```

The following command-line options are available:

Option	*Description*
/A	Shows the hexadecimal address of every line as a comment to the left of the disassembly.
/B	Shows original bytes that were disassembled. The bytes are listed in hexadecimal as comments to the right of the disassembly.
/L	Creates line labels for addresses within the file. This is probably the most useful feature of the disassembler. Labels are not generated for addresses that fall in the middle of an instruction.
/N	Indicates no load address stored at beginning of file. Normally the first two bytes of the file would contain the load address of the file, but with /N they are treated as normal code.
/R	Inserts a LAYOUT section, so the output may be re-assembled with CASM.

Table 1-3: CDASM.EXE Command-line Options

CDUMP <SourceFile> [/A] [/B#] [/D] [/N] [/X[,YY]]

CDUMP.EXE displays the contents of a file to the screen as a listing of bytes. By default, the output is shown in rows of 8 hexadecimal numbers. The following command-line options are available to change the format of the output:

Option	*Description*
/A	Shows addresses for each line to the left of the dump.
/B#	Shows the dump as binary bitmaps rather than hexadecimal numbers. "#" tells which symbol is to be used for "on" bits, and can be "*", "x", or "X" By default, eight bytes are shown per line.
/D	Shows the numbers in decimal instead of hexadecimal.
/N	Indicates no load address stored in file.
/X,YY	This can be used with the /B# option. "X" is a digit between 1 and 9 which tells how many bytes are displayed per line. The optional "YY" specifies can be any number from 1 to 99, and tells how many lines should be in each block. Blocks will be separated by blank lines.

Table 1-4: CDUMP.EXE Command-line Options

CLINK <ListFile> [/U|/C]

CLINK.EXE pastes together a set of files according to the specifications given in **<ListFile>** and stores the result in a .PRG file with the same name as the list file. (If no file-name extension is given for the list file, .LST is assumed.)

Option	*Description*
/U	This parameter reverses the process, generating the small files specified in the .LST file from the .PRG file. This is can be used to break an old project apart into modules.
/C	The "/C" parameter runs CASM on the .ASM files whose dates/time are earlier than that of the corresponding .PRG files. This can be used to re-assemble only those files which have been modified. It is similar to what programs like MAKE do.

Table 1-5: CLINK.EXE Command-line Options

Here is an example list file called TEST.LST:

```
$1000                       ; Load address of TEST.PRG
FILE1.PRG       $1000       ; Start addresses of FILE1.PRG
FILE2.PRG       $1500       ; FILE2.PRG starts $500 bytes later
```

The first address tells the load address of the final .PRG to be generated. Then follows a listing of .PRG modules and the address were they should be placed. The FILE1.PRG is less than $500 bytes in size, the rest are zero-filled as necessary. The above .LST file would place FILE1.PRG at the beginning of TEST.PRG, and then FILE2.PRG $500 bytes later.

Appendix A: A Sample Program

Here follows a sample program written with CASM. While the code is a bit sloppy in places, the output is entertaining when run.[7]

```
;-------------------------------------------------------------------------;
; SPINNER - Pope
;
;    This is a quick little program that spins the BASIC character set
;  using the interrupt.  It is included with CASM Version 1.3, available
;  on the World Wide Web at http://www.cs.wm.edu/~pbgonz/progc64.html,
;  or via Email to Lucas Pope <lupope@vt.edu>
;-------------------------------------------------------------------------;


;-------------------------------------------------------------------------;
LAYOUT
;-------------------------------------------------------------------------;
;
; The BASIC SYS start runs sSetup at $0810, which hooks the user interrupt
; to jump to sSpinner.  Then control is returned to BASIC, and the
; interrupt takes over.

$0801:   dSysStart                ; BASIC start
$0810:   sSetup                   ; intialization code
    sSpinner                      ; interrupt handler
    dProgVars                     ; program variables

ENDL

ALIASB CharBackColor  = $00       ; color of character back
ALIASB CharFrontColor = $0e       ; color of character front
ALIASB TurnPause      = 05        ; pause while letters are turning
ALIASB FlatPause      = 70        ; pause while letters are flat

ALIASB CountDown      = $fd       ; used as a timer

ALIASB Temp1          = $fb       ; temporary variable 1
ALIASB Temp2          = $fe       ; temporary variable 2

ALIASB Next           = $fc       ; stores the turn sequence of
                                  ;  the chars

ALIASW kReturnFromInt = $ea81     ; some Kernel code that returns from
                                  ;  an interrupt

;-------------------------------------------------------------------------;
DATA dProgVars
;-------------------------------------------------------------------------;
OldInterrupt:
   0 0
ENDD

;-------------------------------------------------------------------------;
DATA dSysStart
;-------------------------------------------------------------------------;
;
; The BASIC program looks like this:
;
; 1995 SYS(2064)
;
         EndOfLine    ; This will store the address of the label
         1995         ; This is a decimal word storing the line #
```

[7] Peter Gonzalez neither authored this example program nor claims responsibility for its poor readability.

```
          $9E          ; Basic "SYS" instruction
          "(2064)"     ; hex address $0810
          0            ; end of BASIC line
EndOfLine:
          0 0          ; end of BASIC program
ENDD


;-------------------------------------------------------------------------;
SUB sSetup
;-------------------------------------------------------------------------;
; sSetup performs the program initialization, hooks the user interrupt, and
; then returns control to BASIC.

Start:    sei
          lda $0001
          and #$fb
          sta $0001
          ldx #$00
          ldy #$08
Copy1:    lda $d000,x         ; copying the ROM char mem to 3000 and 2800
          sta $3000,x
          sta $2800,x
          lda $d100,x
          sta $2900,x
          sta $3100,x
          dex
          bne Copy1

          lda $0001
          ora #$04
          sta $0001

          ldx #$06
          lda #$08            ; creates the blinking cursor picture, which
                              ;  will be a vertical line
Cursor:   sta $2d00,x
          dex
          bpl Cursor


          ldx #$00
Flip:     lda $3000,x         ; flipping chars at 3000 and putting them
          lsr acc             ;  at 3800
          rol Temp1
          lsr acc
          rol Temp1
          lsr acc
          rol Temp1
          lsr acc
          rol Temp1
          lsr acc
          rol Temp1
          lsr acc
          rol Temp1
          lsr acc
          rol Temp1
          lsr acc
          rol Temp1
          lsr acc
          rol Temp1
          lda Temp1
          sta $3800,x
          lda $3100,x
          lsr acc
          rol Temp1
          lsr acc
          rol Temp1
          lsr acc
          rol Temp1
          lsr acc
```

```
          rol Temp1
          lsr acc
          rol Temp1
          lsr acc
          rol Temp1
          lsr acc
          rol Temp1
          lsr acc
          rol Temp1
          lda Temp1
          sta $3900,x
          dex
          bne Flip

          ; Make sure the routine is not already running:

          lda dProgVars|OldInterrupt
          cmp #$00
          beq NotLoadedYet
          lda dProgVars|OldInterrupt + 1
          cmp #$00
          beq NotLoadedYet

          cli
          rts                 ; return to basic


NotLoadedYet:

          lda $0314           ; Save the old interrupt:
          sta dProgVars|OldInterrupt
          lda $0315
          sta dProgVars|OldInterrupt + 1

          lda #<sSpinner|Start      ; Set user interrupt to vector through
          sta $0314                 ;  the sSpinner subroutine:
          lda #>sSpinner|Start
          sta $0315

          lda #$00
          sta Next
          lda #$1b
          sta $d018
          lda #FlatPause
          sta CountDown

          cli
          rts                 ; return to basic
ENDS


;-------------------------------------------------------------------------;
SUB sSpinner
;-------------------------------------------------------------------------;
;
; This is just a lot of bit flipping.  I sacrificed some readability to
; have the demo program do something worth seeing.
;

Start:    sei
          dec CountDown
          bne GoOn            ; jump to normal interrupt if CountDown <> 0
          lda #TurnPause
          sta CountDown

          inc Next

          lda Next            ; determines which part of the turn the
          cmp #$04            ;  characters are currently in
```

```
          beq ToBlack
          bcc Go5
          cmp #$05
          beq Go1
          cmp #$06
          beq Go2
          cmp #$07
          beq Flip
          cmp #$0b
          beq ToBlue
          bcc Go5
          cmp #$0c
          beq Go3
          cmp #$0d
          beq Go4

          lda #$00
          sta Next
          jmp norm
Go1:      jmp eflip1                ; jmp reaches for the branches above
Go2:      jmp eflip2
Go3:      jmp enorm1
Go4:      jmp enorm2
Go5:      jmp squish
GoOn:     jmp (dProgVars|OldInterrupt)

ToBlack:  ldx #$f9
          lda #CharBackColor
          sta $0286                 ; changes the color to CharBackColor
Loop1:    sta $d800,x
          sta $d8fa,x
          sta $d9f4,x
          sta $daee,x
          dex
          cpx #$ff
          bne Loop1
          jmp kReturnFromInt

ToBlue:   ldx #$f9
          lda #CharFrontColor
          sta $0286                 ; changes the color to CharFrontColor
Loop2:    sta $d800,x
          sta $d8fa,x
          sta $d9f4,x
          sta $daee,x
          dex
          cpx #$ff
          bne Loop2
          jmp kReturnFromInt



norm:     lda #FlatPause
          sta CountDown
          ldx #$00
dooper:   lda $3000,x             ; copies norm chars into visible
          sta $2800,x
          lda $3100,x
          sta $2900,x
          dex
          bne dooper
          jmp kReturnFromInt

Flip:     lda #FlatPause
          sta CountDown
          ldx #$00
dooper2:  lda $3800,x             ; copies flipped chars into visible
          sta $2800,x
          lda $3900,x
```

```
          sta $2900,x
          dex
          bne dooper2
          jmp kReturnFromInt

                                    ; squishes the visible characters
squish:   ldx #$00
sqshmore:
          lda $2800,x
          tay
          and #$08
          sta Temp2
          tya
          asl acc
          and #$0f
          sta Temp1
          tya
          lsr acc
          and #$f0
          ora Temp1
          ora Temp2
          sta $2800,x
          lda $2900,x
          tay
          and #$08
          sta Temp2
          tya
          asl acc
          and #$0f
          sta Temp1
          tya
          lsr acc
          and #$f0
          ora Temp1
          ora Temp2
          sta $2900,x
          dex
          bne sqshmore
          jmp kReturnFromInt

enorm1:   ldx #$00                  ; copies normal chars into view, squishing
LoopA:    lda $3000,x               ;  them at the same time (squishes less
          tay                       ;  every time) to make them appear to expand
          and #$08
          sta Temp2
          tya
          asl acc
          asl acc
          and #$0f
          sta Temp1
          tya
          lsr acc
          lsr acc
          and #$f0
          ora Temp1
          ora Temp2
          sta $2800,x
          lda $3100,x
          tay
          and #$08
          sta Temp2
          tya
          asl acc
          asl acc
          and #$0f
          sta Temp1
          tya
          lsr acc
          lsr acc
```

```
          and #$f0
          ora Temp1
          ora Temp2
          sta $2900,x
          dex
          bne LoopA
          jmp kReturnFromInt

enorm2:   ldx #$00                        ; same as above only expands less
LoopB:    lda $3000,x
          tay
          and #$08
          sta Temp2
          tya
          asl acc
          and #$0f
          sta Temp1
          tya
          lsr acc
          and #$f0
          ora Temp1
          ora Temp2
          sta $2800,x
          lda $3100,x
          tay
          and #$08
          sta Temp2
          tya
          asl acc
          and #$0f
          sta Temp1
          tya
          lsr acc
          and #$f0
          ora Temp1
          ora Temp2
          sta $2900,x
          dex
          bne LoopB
          jmp kReturnFromInt


eflip1:   ldx #$00                  ; copies flipped characters into view
LoopC:    lda $3800,x               ;  "squishing" them at the same time
          tay
          and #$08
          sta Temp2
          tya
          asl acc
          asl acc
          and #$0f
          sta Temp1
          tya
          lsr acc
          lsr acc
          and #$f0
          ora Temp1
          ora Temp2
          sta $2800,x
          lda $3900,x
          tay
          and #$08
          sta Temp2
          tya
          asl acc
          asl acc
          and #$0f
          sta Temp1
          tya
```

```
          lsr acc
          lsr acc
          and #$f0
          ora Temp1
          ora Temp2
          sta $2900,x
          dex
          bne LoopC
          jmp kReturnFromInt

eflip2:   ldx #$00                ; same as above only less
LoopD:    lda $3800,x
          tay
          and #$08
          sta Temp2
          tya
          asl acc
          and #$0f
          sta Temp1
          tya
          lsr acc
          and #$f0
          ora Temp1
          ora Temp2
          sta $2800,x
          lda $3900,x
          tay
          and #$08
          sta Temp2
          tya
          asl acc
          and #$0f
          sta Temp1
          tya
          lsr acc
          and #$f0
          ora Temp1
          ora Temp2
          sta $2900,x
          dex
          bne LoopD
          jmp kReturnFromInt
ENDS
```

Appendix B: Recognized 6502 Instructions

The following table lists the 6502 instructions supported by CASM.

Code	Instr.	Parm.	Code	Instr.	Parm.	Code	Instr.	Parm.	Code	Instr.
00	BRK		01	ORA	($00,X)	02	---		03	---
04	---		05	ORA	$00	06	ASL	$00	07	---
08	PHP		09	ORA	#$00	0A	ASL	ACC	0B	---
0C	---		0D	ORA	$0000	0E	ASL	$0000	0F	---
10	BPL	$NEAR	11	ORA	($00),Y	12	---		13	---
14	---		15	ORA	$00,X	16	ASL	$00,X	17	---
18	CLC		19	ORA	$0000,Y	1A	---		1B	---
1C	---		1D	ORA	$0000,X	1E	ASL	$0000,X	1F	---
20	JSR	$0000	21	AND	($00,X)	22	---		23	---
24	BIT	$00	25	AND	$00	26	ROL	$00	27	---
28	PLP		29	AND	#$00	2A	ROL	ACC	2B	---
2C	BIT	$0000	2D	AND	$0000	2E	ROL	$0000	2F	---
30	BMI	$NEAR	31	AND	($00),Y	32	---		33	---
34	---		35	AND	$00,X	36	ROL	$00,X	37	---
38	SEC		39	AND	$0000,Y	3A	---		3B	---
3C	---		3D	AND	$0000,X	3E	ROL	$0000,X	3F	---
40	RTI		41	EOR	($00,X)	42	---		43	---
44	---		45	EOR	$00	46	LSR	$00	47	---
48	PHA		49	EOR	#$00	4A	LSR	ACC	4B	---
4C	JMP	$0000	4D	EOR	$0000	4E	LSR	$0000	4F	---
50	BVC	$NEAR	51	EOR	($00),Y	52	---		53	---
54	---		55	EOR	$00,X	56	LSR	$00,X	57	---
58	CLI		59	EOR	$0000,Y	5A	---		5B	---
5C	---		5D	EOR	$0000,X	5E	LSR	$0000,X	5F	---
60	RTS		61	ADC	($00,X)	62	---		63	---
64	---		65	ADC	$00	66	ROR	$00	67	---
68	PLA		69	ADC	#$00	6A	ROR	ACC	6B	---
6C	JMP	($0000)	6D	ADC	$0000	6E	ROR	$0000	6F	---
70	BVS	$NEAR	71	ADC	($00),Y	72	---		73	---
74	---		75	ADC	$00,X	76	ROR	$00,X	77	---
78	SEI		79	ADC	$0000,Y	7A	---		7B	---
7C	---		7D	ADC	$0000,X	7E	ROR	$0000,X	7F	---

80	---	81	STA ($00,X)	82	---	83	---
84	STY $00	85	STA $00	86	STX $00	87	---
88	DEY	89	---	8A	TXA	8B	---
8C	STY $0000	8D	STA $0000	8E	STX $0000	8F	---
90	BCC $NEAR	91	STA ($00),Y	92	---	93	---
94	STY $00,X	95	STA $00,X	96	STX $00,Y	97	---
98	TYA	99	STA $0000,Y	9A	TXS	9B	---
9C	---	9D	STA $0000,X	9E	---	9F	---
A0	LDY #$00	A1	LDA ($00,X)	A2	LDX #$00	A3	---
A4	LDY $00	A5	LDA $00	A6	LDX $00	A7	---
A8	TAY	A9	LDA #$00	AA	TAX	AB	---
AC	LDY $0000	AD	LDA $0000	AE	LDX $0000	AF	---
B0	BCS $NEAR	B1	LDA ($00),Y	B2	---	B3	---
B4	LDY $00,X	B5	LDA $00,X	B6	LDX $00,Y	B7	---
B8	CLV	B9	LDA $0000,Y	BA	TSX	BB	---
BC	LDY $0000,X	BD	LDA $0000,X	BE	LDX $0000,Y	BF	---
C0	CPY #$00	C1	CMP ($00,X)	C2	---	C3	---
C4	CPY $00	C5	CMP $00	C6	DEC $00	C7	---
C8	INY	C9	CMP #$00	CA	DEX	CB	---
CC	CPY $0000	CD	CMP $0000	CE	DEC $0000	CF	---
D0	BNE $NEAR	D1	CMP ($00),Y	D2	---	D3	---
D4	---	D5	CMP $00,X	D6	DEC $00,X	D7	---
D8	CLD	D9	CMP $0000,Y	DA	---	DB	---
DC	---	DD	CMP $0000,X	DE	DEC $0000,X	DF	---
E0	CPX #$00	E1	SBC ($00,X)	E2	---	E3	---
E4	CPX $00	E5	SBC $00	E6	INC $00	E7	---
E8	INX	E9	SBC #$00	EA	NOP	EB	---
EC	CPX $0000	ED	SBC $0000	EE	INC $0000	EF	---
F0	BEQ $NEAR	F1	SBC ($00),Y	F2	---	F3	---
F4	---	F5	SBC $00,X	F6	INC $00,X	F7	---
F8	SED	F9	SBC $0000,Y	FA	---	FB	---
FC	---	FD	SBC $0000,X	FE	INC $0000,X	FF	---

Appendix C: Distribution and Modification

CASM and its supporting files CDASM, CLINK, and CDUMP are freeware. This means that they may be copied and distributed freely, provided that they are not modified and no fee is charged. The source code is available only directly through the author, and may not be distributed in any other way. You may contact me via e-mail at:

```
Peter Gonzalez <pbgonz@mail.wm.edu>
```

or download any new versions from my World Wide Web page at:

```
http://www.cs.wm.edu/~pbgonz/progc64.html
```

You are also encouraged to send me e-mail if you are missing some files, would like to report a bug, or have any comments/feedback. In fact, user feedback is probably the only force which will motivate me to distribute further improvements. ☺

Always give people credit for their own work, and support freeware!

NOTES

NOTES

NOTES