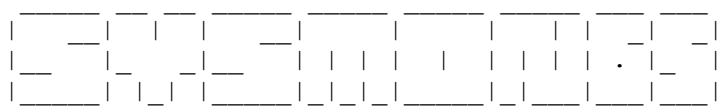
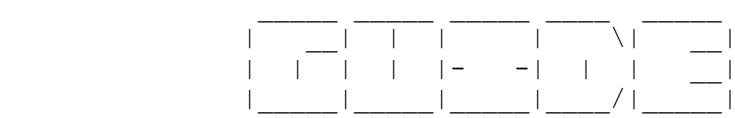
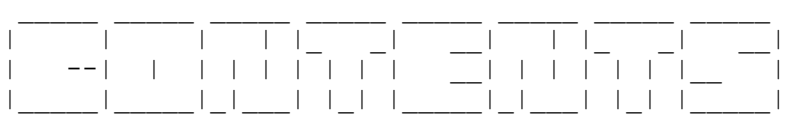


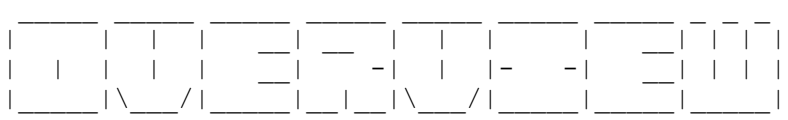


Version 1.07

Joe DiMeglio

Contributor: Peter G





Overview

SYSMON65 is a system monitor which has been developed for my custom built 65C02 single board computer (SBC). This monitor can easily be ported to other 65C02 based SBCs. SYSMON65 also supports 65C02 specific instructions, such as phx, plx, bra, and more. However, it may be converted back to 6502 specific instructions (if anyone were really interested in do so) for use with the KIM-1, Apple II series computers, Commodore 64, VIC-20, and other hardware implementations.

My SBC includes the venerable 65C02 processor, a 6551 for UART functionality, a 6522 chip for I/O ports, a 27C256 typical ROM and RAM chips, and address decoding logic chips.

After searching the internet and reviewing the few 6502 operating systems and monitor software that I could find, I discovered that I only liked parts of the functionality of each. So, I decided to develop my own monitor software. During early development, the memory footprint of the monitor software was not considered, but as development progressed this became a priority, I had to keep it “real tight”! {Except for the text graphics – although ive add a pack routine to compress those as much as possible}

My goal for *SYSMON65* is to be a software development tool for my SBC. A user may paste code into the terminal, compile it, do memory dumps, decode it, edit the code, and much more…

Parts of my code was inspired by the following:

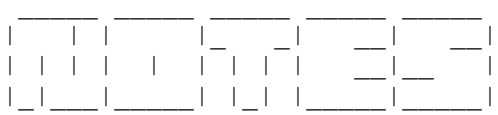
- The *A1 Assembler* by San Bergmans
  I like the front-end editor, but the assembler does not come with a dissembler. The assembler is a 2-pass assembler and feels very solid, but it was written primarily for an Apple computer. With San’s permission I used his front-end editor code, reviewed every line, and eventually re-wrote most of this.

- The KRUSADER by Ken Wessen
  This has a super-efficient disassembler (most likely created by MOS and used by Apple) and includes the specific 65C02 instructions. With respect, I did not like the front-end editor.

The look and feel of *SYSMON65* was heavily influenced by the famous line-oriented debugger Debug https://en.wikipedia.org/wiki/Debug_(command) found in DOS (those were the days!). Some *Debug* front-end functionality can be found in *SYSMON65*, including the [Backspace] key to fix mistypes, and command history functionality using the [Up arrow] key.

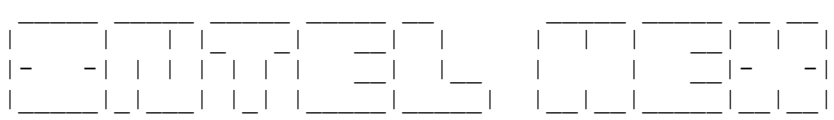
SYSMON65 includes a full 2-pass assembler with local and global labels, directives, and more. A 65C02 disassembler is included which includes step-by-step debugging (aka Tracing), memory dumps, ASCII dumps, fill, delete, block move, intel hex loader and more.

SYSMON65 has been tested on hardware running a N65C02 processor. It also includes code for an LCD 16x2 module. The software currently takes just over 7.2KB of memory space.



Notes

- *SYSMON65* has been designed for the *Rockwell 65C02* CPU.
- Terminal running ANSI screen codes with serial 19200 baud N81.
- 6551 ACIA routines are bug free, i.e., does not include the Xmit bug.
- I use *RealTerm* https://sourceforge.net/projects/realterm/
- Assembled using *Michael Kowalski* (minimum version 1.3.2)



Intel Hex Loader

Intel Hex Loader functionality has been built in.

The command line will look for a semicolon (:) as the first character and will automatically download the file. Simply paste the intel hex file contents to the command line.

With the MYWYM directive enabled, 4 LEDs via the VIA ports 1 to 4 will flash (think “knight rider”) with every block read.

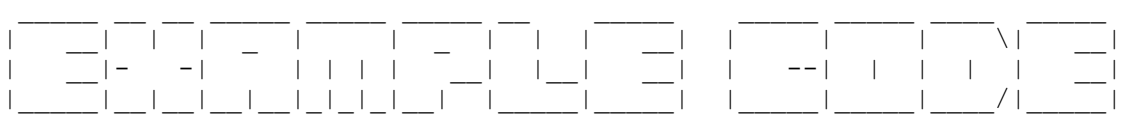
Example Intel Hex File

The listing below shows an example Intel Hex file. The colour coding below the listing defines the various fields of the Intel hex file:

```
:10010000214601360121470136007EFE09D2190140
:100110002146017E17C20001FF5F16002148011928
:10012000194E79234623965778239EDA3F01B2CAA7
:100130003F0156702B5E712B722B732146013421C7
:00000001FF
```

Start code | Byte count | Address | Record type | Data | Checksum

See: https://en.wikipedia.org/wiki/Intel_HEX



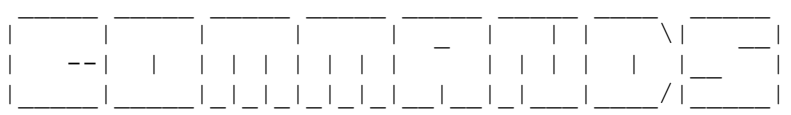
Example Code

SYSMON65 is designed to allow copy & paste from an external text application, e.g., Windows Notepad. Then simply compiling the pasted code with the “S” key.

Copy & paste the sample below (i.e., between the lines). Notice the variable declarations, local and global variables.

```
      AUTO

WRBYTE      .EQ $FFDC
ECHO        .EQ $FFEF
CR          .EQ $0D
SP          .EQ $20
;-----------------------------
START       JSR HELLO  ;output to screen
            JSR COUNT
            RTS
;-----------------------------
HELLO       LDX #0
.1          LDA .3,X
            BPL .2
            JSR ECHO
            INX
            BNE .1
.2          ORA #1000.0000
            JMP ECHO
.3
;-----------------------------
      .AT -/HELLO WORLD/
;-----------------------------
COUNT       JSR .2
            LDX #0
.1          TXA
            JSR WRBYTE
            LDA #" "
            JSR ECHO
            INX
            CPX #10
            BCC .1
.2          LDA #CR
            JMP ECHO
```



Command

The following section contains the valid commands of the monitor.

 _____ _____ _____ _____
_			_ _		
__	__	_____		_		_____

Auto (A)

Auto linenum,increment

This command will auto number each line. The assembler uses line numbers to allow you to identify which line you like to maintain, i.e., delete, insert, re-number etc. Pressing the [Escape] key on the last line will exit the assembler editor.

Note, the origin of the source is defined by the variable DEF_ORG (default is $1000) and the incremental steps are DEF_INC (default is 10). Also, the count is set by DEF_AUTO – which is 1000

In/Out Window

```
.1020 ECHO    .EQ $FFEF
.1030 CR      .EQ $0D
.1040 SP      .EQ $20
.1050
.1060 ;-------------------------------
.1070 START       JSR HELLO ;output to screen
.1080             JSR COUNT
.1090             RTS
.1100
.1110 ;-------------------------------
.1120 HELLO       LDX #0
.1130 .1          LDA .3,X
.1140             BPL .2
.1150             JSR ECHO
.1160             INX
.1170             BNE .1
.1180 .2          ORA #1000.0000
.1190             JMP ECHO
.1200
.1210 .3
.1220 ;-------------------------------
.1230         .AT -/HELLO WORLD/
.1240 ;-------------------------------
.1250
.1260 COUNT       JSR .2
.1270             LDX #0
.1280 .1          TXA
.1290             JSR WRBYTE
.1300             LDA #" "
.1310             JSR ECHO
.1320             INX
.1330             CPX #10
.1340             BCC .1
.1350 .2          LDA #CR
.1360             JMP ECHO
```

Figure 1

If no operand(s) are specified, then auto line numbering will commence from the last entered line number + current increment. If no line has been entered prior, then auto line numbering will commence from line number 1000, with an increment value of 10.

You may use *linenum* to start auto line numbering from any specified number.
You may use *increment* to change the default increment of 10.

```
AUTO           Start numbering from last entered line number + increment
AUTO 2000      Start numbering from 2000 with unchanged increment
AUTO 4000,5    Start numbering from 4000 with 5 as increment
AUTO ,10       Start numbering from last entered line number + 5 as new
increment
```

Pressing the [Escape] key will cancel auto line numbering and the current unfinished line. Simply press the [Escape] key when you have completed entering your source code or when you have made a syntax error and cannot correct with the [Backspace] key. Typing AUTO again will generate the same line number you had just cancelled to allow you to start again with this line.

You do not have to use AUTO line numbering if you only want to enter a few lines somewhere in your code. Simply type the appropriate line number after the prompt followed by your source text.

The value of *increment* is limited to within the range of 1 to 255. Higher values are truncated to the LSB value only, which could cause some unexpected increments. An *increment* of 0 will result in an increment of 1.

Copy (C)

COPY source,destination,length

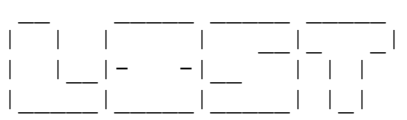
This command can be used to copy a part of memory to another destination. Please note, all three parameters are mandatory!

It is possible that the destination block will eventually overwrite the source block. This means that the original block can be partially destroyed after the copy. However, the copy will always be an exact copy of the original contents of the source block.

Warning! Be careful when the destination is specified in page 0. The COPY command utilises 6 bytes in page 0 as temporary storage. Overwriting these values will very likely crash your system. You should also be aware that the input buffer may partially overwrite your copied code if the destination is in the zero page.

There is absolutely no safeguard built into this command. You can make a copy anywhere in RAM, effectively destroying the data which is overwritten. This might even be your precious source text!

This command can be useful if you assembled a program with a different target address (Refer to the *.TA* directive). After assembling your code, you can move the code to the desired destination.



List (L)

LIST begin,end

This command lists your source to the screen. If no parameters are specified, then the entire program is listed. The *begin* and *end* parameters can be used in the usual manner to control the range to be listed.

```
LIST                    list entire program
LIST 1000               list only line 1000
LIST 1000,2000          list lines 1000 until 2000
LIST 1000,              list from line 1000 until the end of source
LIST ,2000              list from begin of source to line 2000
LIST D                  list dump the entire program
```

The [Escape] key aborts the listing.

The LIST command has an additional feature. Typing LIST D will dump the entire program to the output without line numbers. This option can be used to transfer your source file to the PC over the RS232 connection. The resulting file on the PC may then be saved.

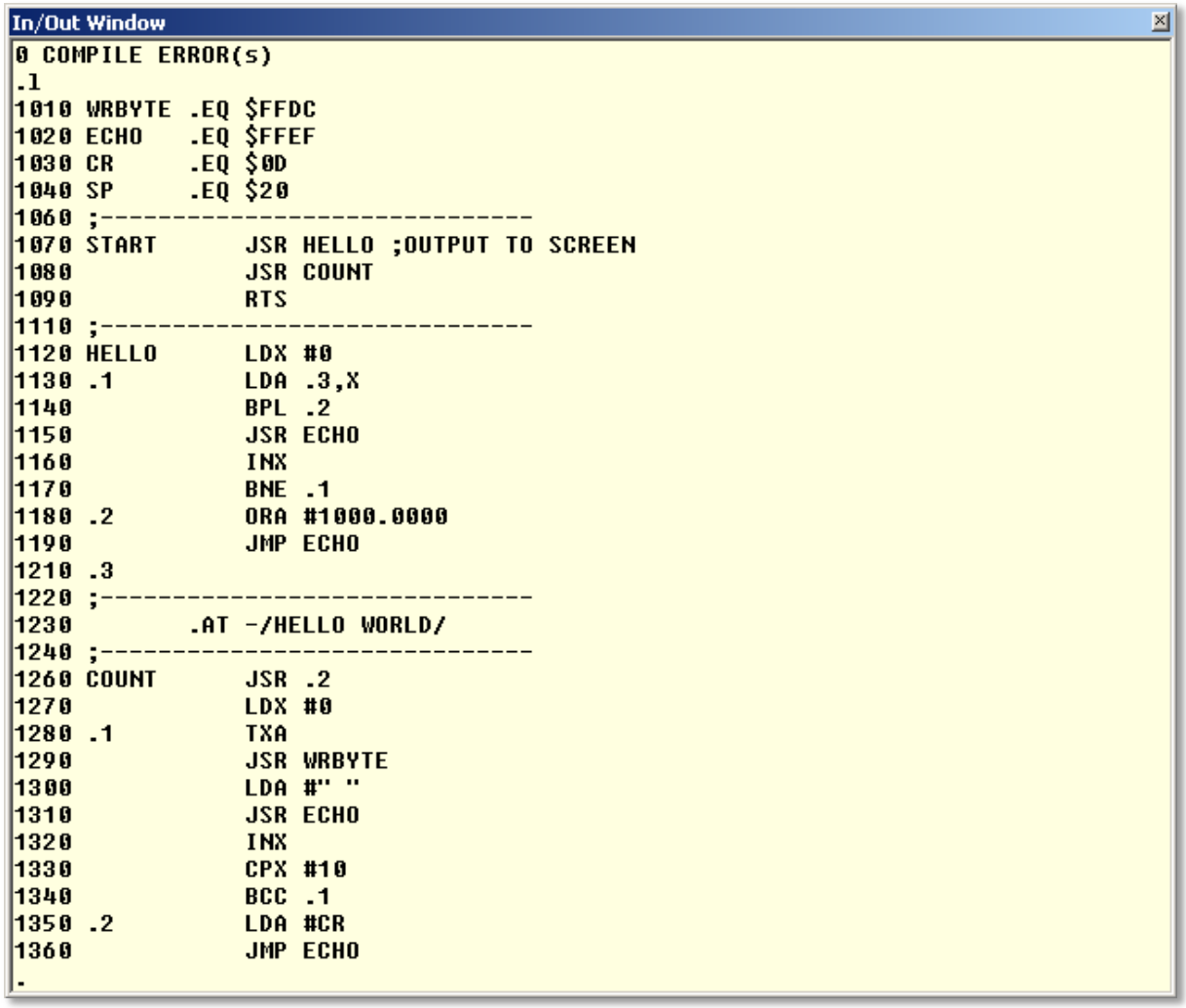

In/Out Window
0 COMPILE ERROR(s)
.1
1010 WRBYTE .EQ $FFDC
1020 ECHO   .EQ $FFEF
1030 CR   .EQ $0D
1040 SP   .EQ $20
1060 ;------------------------------
1070 START   JSR HELLO ;OUTPUT TO SCREEN
1080   JSR COUNT
1090   RTS
1110 ;------------------------------
1120 HELLO   LDX #0
1130 .1   LDA .3,X
1140   BPL .2
1150   JSR ECHO
1160   INX
1170   BNE .1
1180 .2   ORA #1000.0000
1190   JMP ECHO
1210 .3
1220 ;------------------------------
1230   .AT -/HELLO WORLD/
1240 ;------------------------------
1260 COUNT   JSR .2
1270   LDX #0
1280 .1   TXA
1290   JSR WRBYTE
1300   LDA #" "
1310   JSR ECHO
1320   INX
1330   CPX #10
1340   BCC .1
1350 .2   LDA #CR
1360   JMP ECHO
.

Figure 2

```
 _____ _____ __    __
|   __|     |  |  |  |
|   __|-   -|  |__|  |__
|__|  |_____|_____|_____|
```

Fill (F)

Fill begin, range, byte value

Fill a location from source to destination with specified value.

```
Fill                    illegal command
Fill 1000               illegal command
Fill $1000,$0200,$ff    fill from location $1000, 512($100) bytes with $ff
Fill $1000 $0200 $ff    fill from location $1000, 512($100) bytes with $ff
```

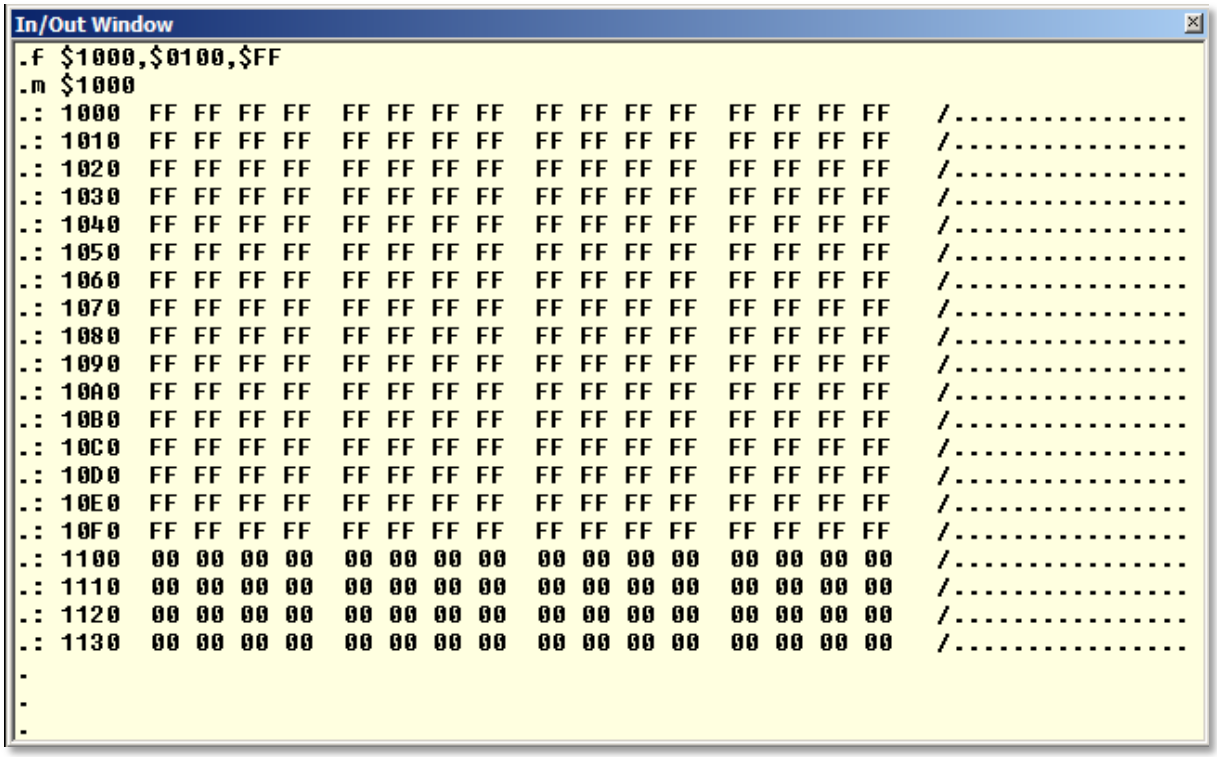
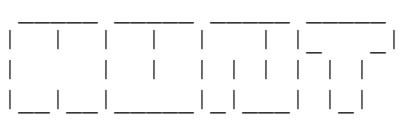


Figure 3



Hunt (H) - future

Hunt begin, end, value

Hunt for the specified value within the start to end range in memory.

{Status: Feature not complete!}

Memory (Y)

Y lomem, himem

This command can be used to examine or change the memory configuration.

With no parameters this command will show you the current **Lower Limit(**LOMEM), **Total RAM** and end of source address. Your source file starts at address Lower Limit and may extend almost to address **UPPER Spent**(HIMEM). The end of your source file is at the same time the beginning of the symbol table which is built during pass 1 of the assembler. The symbol table will hold all your label declarations and may grow from the end of the source text all the way up to UPPER Spent.

Each global label will occupy 6 bytes in the symbol table, while each local label will occupy 2 bytes. This should give you a rough idea about the required amount of memory for the symbol table.

You can use the MEMORY (Y) command to find out what part of memory to save to file/cassette in order to store your source text. Lower Limit will be the start address and end of source will be the end address to write.

Generated code can be stored from address $0200 up to LOMEM, unless you have set the user safe area which can be set with the zero page addresses USR_OBJLO and USR_OBJHI.

```
Y
0600.$8000      lomem,himem
0F14            end of source text
```

At start up LOMEM will be set to $0600 and HIMEM to the highest available RAM address (max $8000). You may change LOMEM and HIMEM to your own liking, and I mean that!

Sensible values are from address $0200 up to the last available RAM .Any other values will probably crash your computer sooner or later!

Changing LOMEM and/or HIMEM will delete your current source text!

```
Y $1000,$8000
$1000.$8000
$1000
```

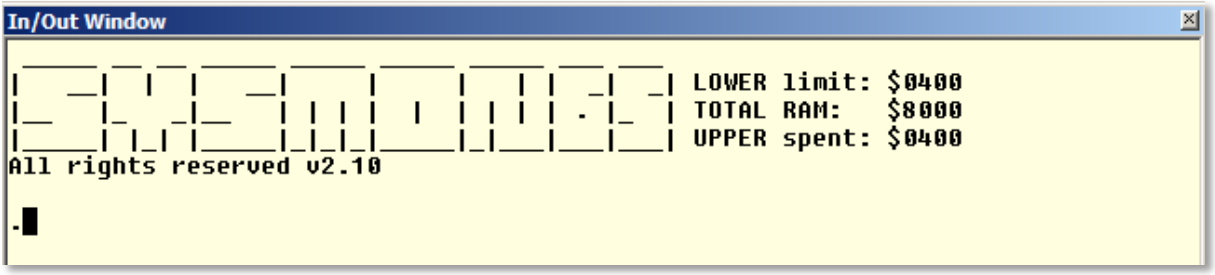

In/Out Window
LOWER limit: $0400
TOTAL RAM: $8000
UPPER spent: $0400
All rights reserved v2.10

Figure 4

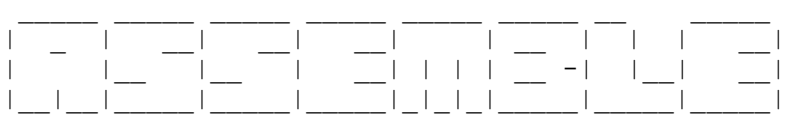
New (N)

With this command you simply delete your current source text so you can start from scratch.

```
N          ; deletes your source code
N -f       ; force restores memory address's as if cold booted
```

Old (O)

If you accidentally typed the NEW command you may restore your program. This will only work if you haven't entered any new source lines after executed the NEW command!



Assemble Code (S)

This command effectively starts the 2 pass assembler. If no errors are found this command will inform you about the memory locations which are used to store the generated code.

The first column starts after the first space behind the line number. This column may contain a label or may be blank. A global label always starts with a character from A to Z and may contain any number of characters from A to Z, 0 to 9, or dots. Global label definitions may be followed by a colon, which is customary in some assemblers. Local labels always start with a dot, followed by a decimal number from 0 to 99.

If the first column does not contain a label it must start with a space. Or a line can start with a semi-colon in the first column, which indicates that the rest of the line is a comment. Comments are ignored by the assembler and are only there for us humans.

```
1000 LABEL
1010 ECHO:
1020 .59
1030 LABEL.WITH.A.VERY.LONG.NAME
1040 ; THIS LINE CONTAINS A COMMENT
1050 ; COMMENT LINES ARE IGNORED BY
1060 ; THE ASSEMBLER
1070  NOP       NO LABEL ON THIS LINE
```

If the first character is a space the first column is considered empty, and thus contains no label (See line 1070). Please note that would make 2 spaces if you also count the space which always follows the line number!

Per default a label gets the value of the current program counter. Only global labels may get a different value if the source line contains an .EQ directive.

Please note that global labels may contain virtually any number of characters (from 1, up to the maximum line length). All these characters are significant!

However in order to preserve memory keep your labels as short as possible but keep them meaningful. Every character is one byte of your valuable memory, for every reference to that label!

If your source text contains errors the line numbers of the offending lines are listed, followed by a short description of the error which occurred. No code will be generated if errors occur during pass 1. Code generated in pass 2 will not be reliable if any errors occur during assembly.

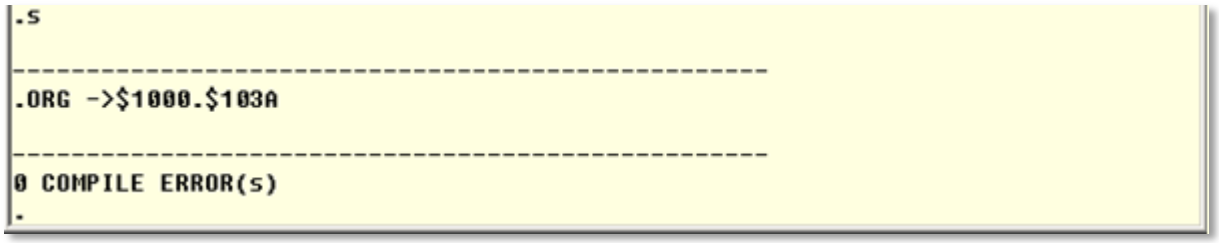


Figure 5

Compiling Errors will show you which lines have errored on.

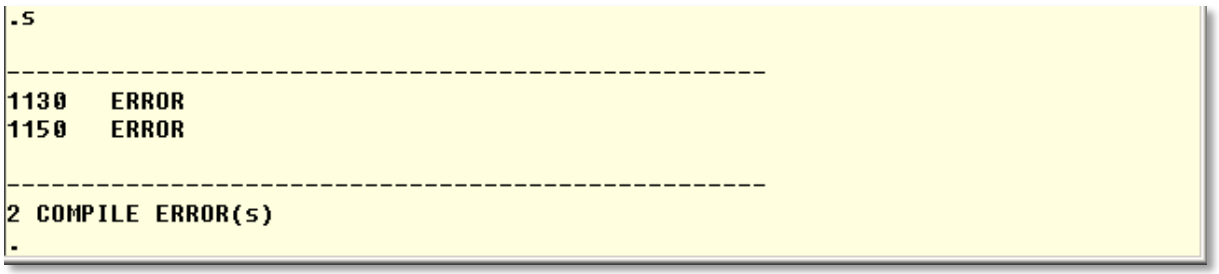


Figure 6

The second column starts at least one space behind the first column. It contains an assembler directive or a mnemonic.

An assembler directive always starts with a dot, followed by 2 characters. See the description of the available directives further down this page. A mnemonic always consists of 3 characters

The second column may also start with a semicolon, which means that the rest of the line contains comments only.

If the second column is left empty, the entire rest of the line must remain empty. This is not a problem for the assembler. It is perfectly legal to place only a single label on a separate source line.

```
1000 START ; THE PROGRAM STARTS HERE
1010       INX
1020 .1    RTS
1030 TEXT
1040       .AS -/HELLO/
```

The third column starts at least one space behind the second column. It contains the operand of the previous mnemonic or assembler directive, if one is required. If the previous

mnemonic or assembler directive did not need an operand this column is simply regarded as comment.

Some mnemonics have an optional operand. One such an example is the ROL instruction. Without operand it Rolls the contents of the Accumulator. With an operand it Rolls the contents of the address indicated by the operand.

In such cases you will have to use a semi-colon as a comment delimiter.

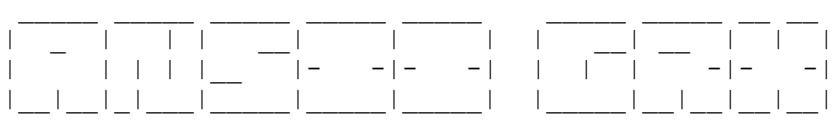
```
1000      ROL
1010      ROL MEMORY
1020      ROL         ;THIS IS A COMMENT
```

Break (B)

This command will execute BRK software interrupts. You can use the BRK as a software interrupt with the second byte following BRK the command.

Clear Screen (Z)

This command will clear the screen by ANSI screen codes sent to the terminal.



ANSI Library Routines

Some defined routines in the library – see general lib.65s

```
GOTOXY      ; GOTO x & y
WRCSI       ; CSI n x See https://en.wikipedia.org/wiki/ANSI_escape_codel ow level
CLS         ; ANSI - Clear Screen; ;.BYTE      ESC,"[2J",EOS
HOME        ; Bring Cursor Home
SAVECUR     ; Save Cursor Position
RESTCUR     ; Restore Cursor Position
```

Help (?)

This command will display the SYSMON65 help screen.

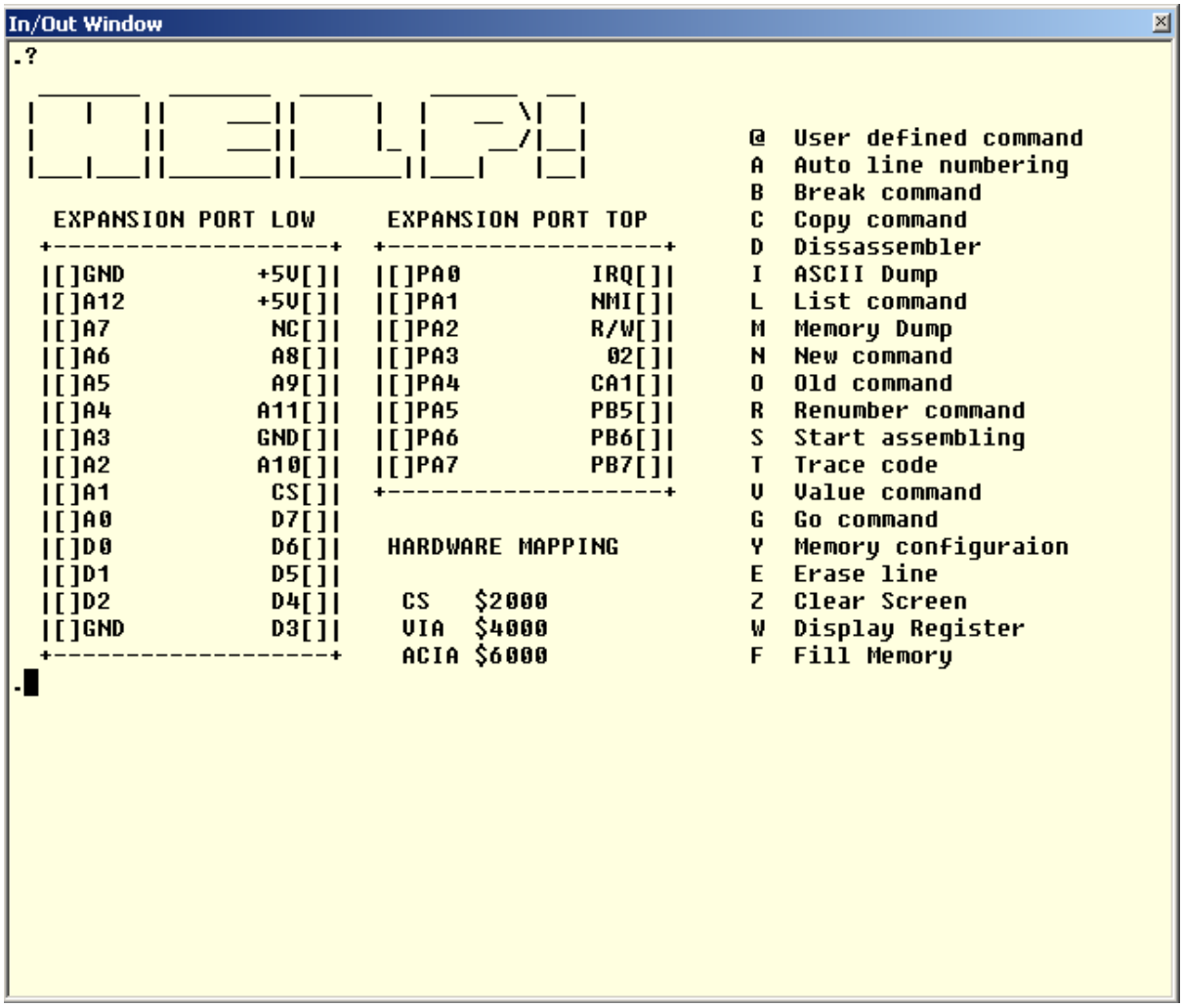
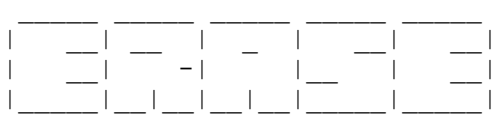


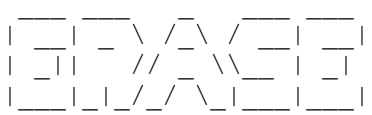
Figure 7

Go (G)

G *address or label*

This command will execute the code from the specified address or label.





Erase Line (E)

Erase begin,end

This command will delete multiple lines. Be careful as undo is not possible. Once deleted the lines are not recoverable!

Both the *begin* and *end* parameters are optional. But one of these parameters are mandatory for safety reasons.

```
ERASE 2000              delete only line 2000
ERASE 2000,2300         delete lines from line 2000 to 2300
ERASE 2000,             delete from line 2000 until the end of source
ERASE,2300              delete from begin of source to line 2300
```

Value (V)

VALUE expression,expression

This command can be used to view the value of labels, convert numbers from one radix to another, or do simple calculations. Label values are only valid after a successful assembly run.

```
VALUE $1234
4660 +4660 $1234 %0001.0010.0011.0100

VALUE -1
65535 -1 $FFFF %1111.1111.1111.1111

VALUE $1234+135
4795 +4795 $12BB %0001.0010.1011.1011

VALUE ECHO
65519 -17 $FFEF %1111.1111.1110.1111

VALUE $1234,1234,%0101.1010
4660 +4660 $1234 %0001.0010.0011.0100
1234 +1234 $04D2 %0000.0100.1101.0010
90 +90 $005A %0101.1010
```

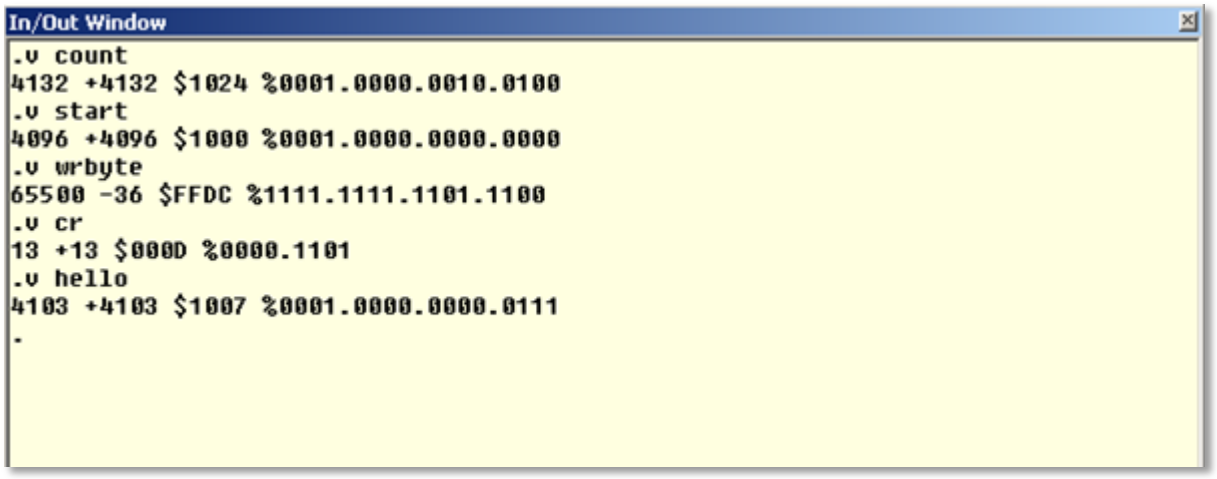
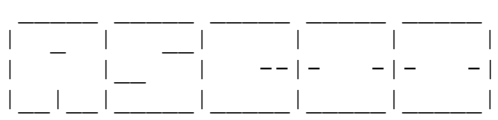


Figure 8



Ascii (I)

i *address*

This command is an ASCII dump of the specified address of RAM. Address values pre-fixed with a $ symbols is considered a hexadecimal value.

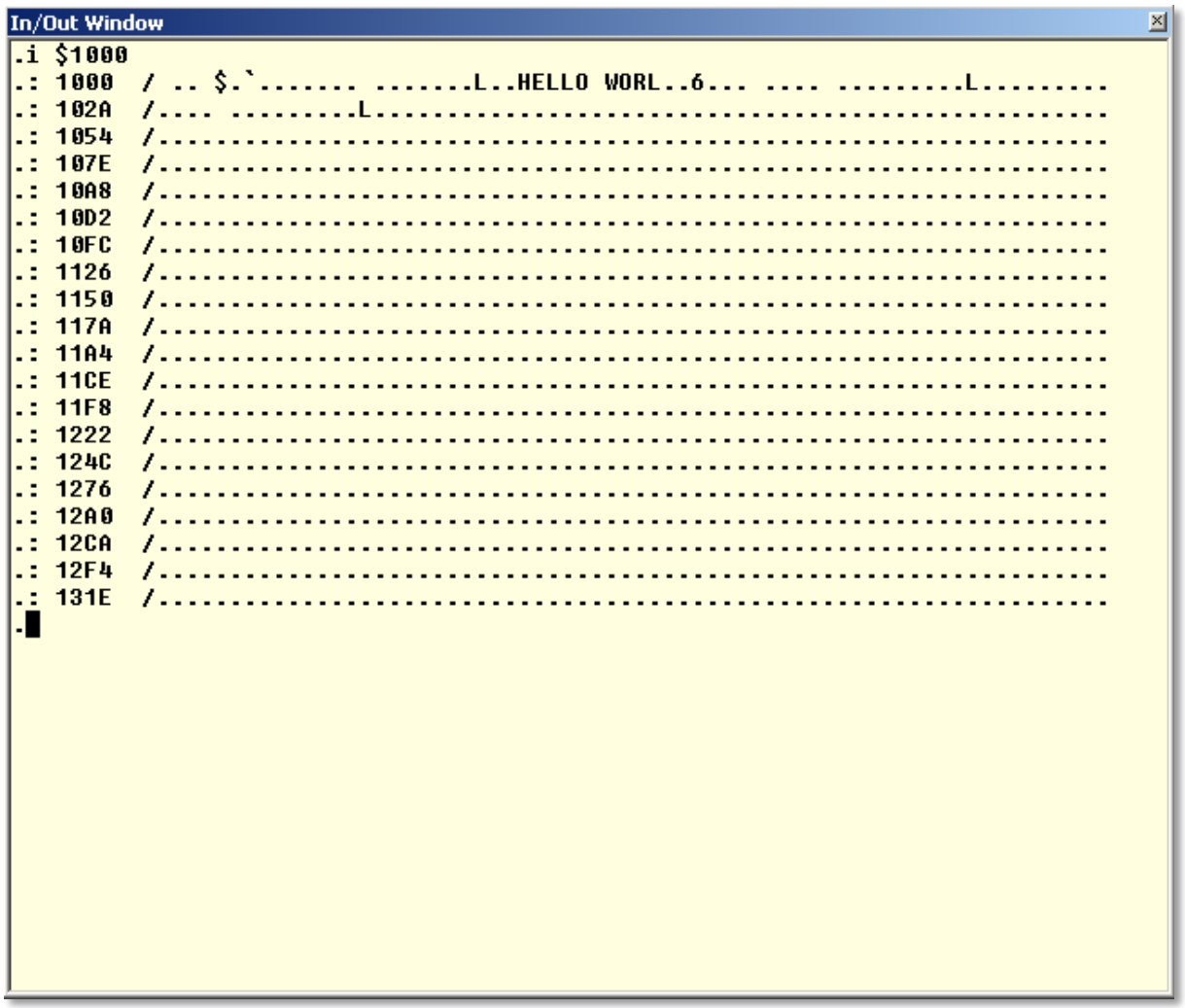
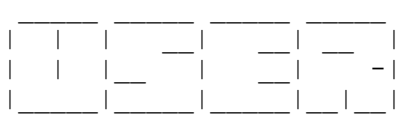


Figure 9



User Command (@)

@ command

This command allows users to extend the commands to the monitor. Changing the USERKEYDEF vector, will mean that you can then add commands to the keyboard input. I.e., @ command will jump to USERKEYDEF where the user then needs to parse the IN keyboard buffer for addition keys/commands.

For example @S would jump to USERKEYDEF {aka *JMP (USERKEYDEF)* } where your routine would parse the "IN" for the character S and then act accordingly if found. See routine KEYDEF for example of how current commands are parsed.

NB: "IN" is the location for the keyboard buffer. 128 bytes reserved in zero page.

See file constant.65s

Renumber (R)

RENUMBER from,first,increment

This command will re-number your entire (or part of your) source code. Usually this is done to tidy up snippets of code or make room for more source lines between existing source lines.

The *from* parameter determines the line from which to start renumbering. If you omit this then you will renumber your entire program.

The *first* parameter will be the first new line number to be used for the renumbered part of your source. If omitted, the default AUTO line number will be used (value of 1000).

The *increment* parameter will determine the increment of the renumbered part of your source. If omitted, the default increment of 10 will be used. The valid range for *increment* is between 1 and 255.

Parameter value *from* cannot be greater than parameter value *first*, else you may get duplicate line numbers.

After renumbering, the next auto line number will be the last renumbered line number + *increment*. The new increment will also be set according to the renumbered increment.

```
RENUMBER               renumbers entire source, same as RENUMBER 0,1000,10
RENUMBER 2000,3000     renumbers source from 2000 until end, increment 10
RENUMBER ,4000         renumbers entire source, new source starts at 4000
RENUMBER 1000,2000,5   renumbers from line 2000, new line 2000, increment 5
```

Disassembler (D)

D address (or label)

This command will dissemble code commencing from (start) address or label.

```
D               Continues disassembling from last address
D $2000         Disassemble from memory hex location $2000
D start         Disassemble from label start
```

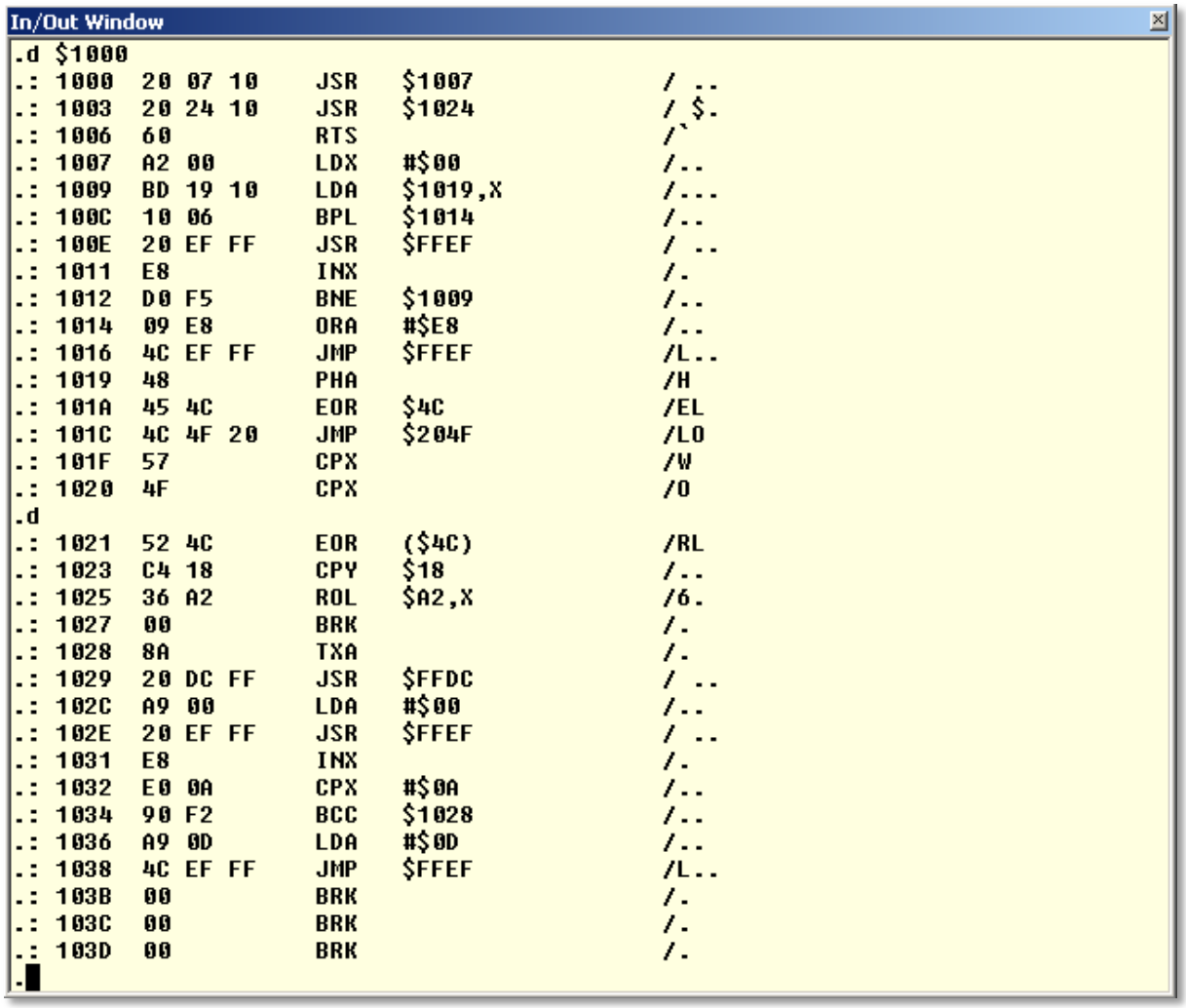


Figure 10

Mem Dump (M)

M address

This command will display a byte and ASCII dump of RAM. Subsequent commands (of M) will continue to page through the memory.

```
MEM                     Continues mem dump from last address
MEM $2000               Mem dump from memory hex location $2000
MEM start               Mem dump from label start
```

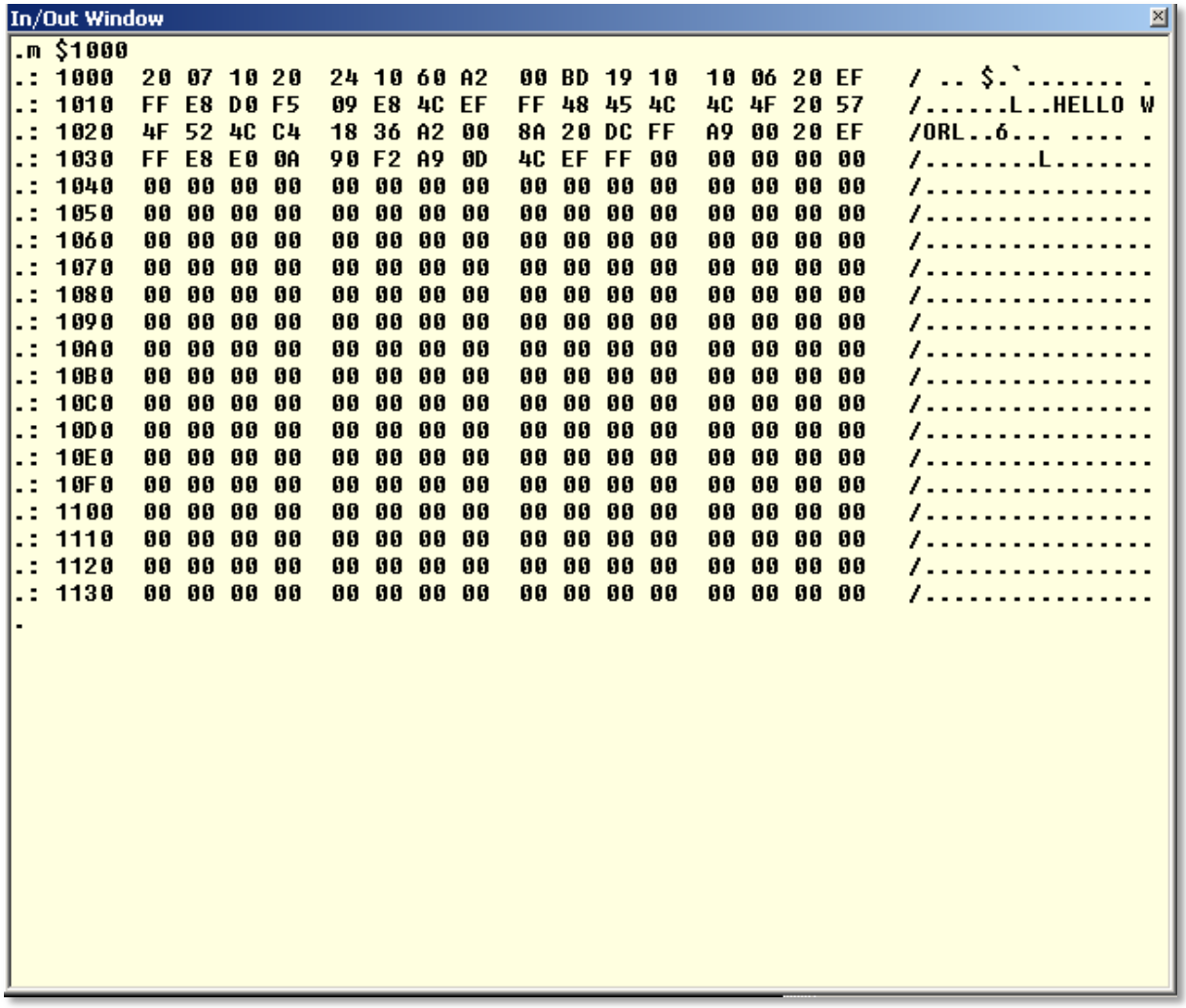


Figure 11

```
 _____ _____ _____ _____ _____
|_   _| __  |  _  |     |   __|
  | | |    -|     |   --|   __|
  |_| |__|__|__|__|_____|_____|
```

Trace (T)

T *address*

This command will allow the user to step through the code one line at time. Pressing t will continue to step through each line. The flags, PC, and registers will show the values of the last actioned command.

```
TRACE                   Continues tracing from last address
TRACE $2000             Trace from memory hex location $2000
TRACE start             Trace from label start
```

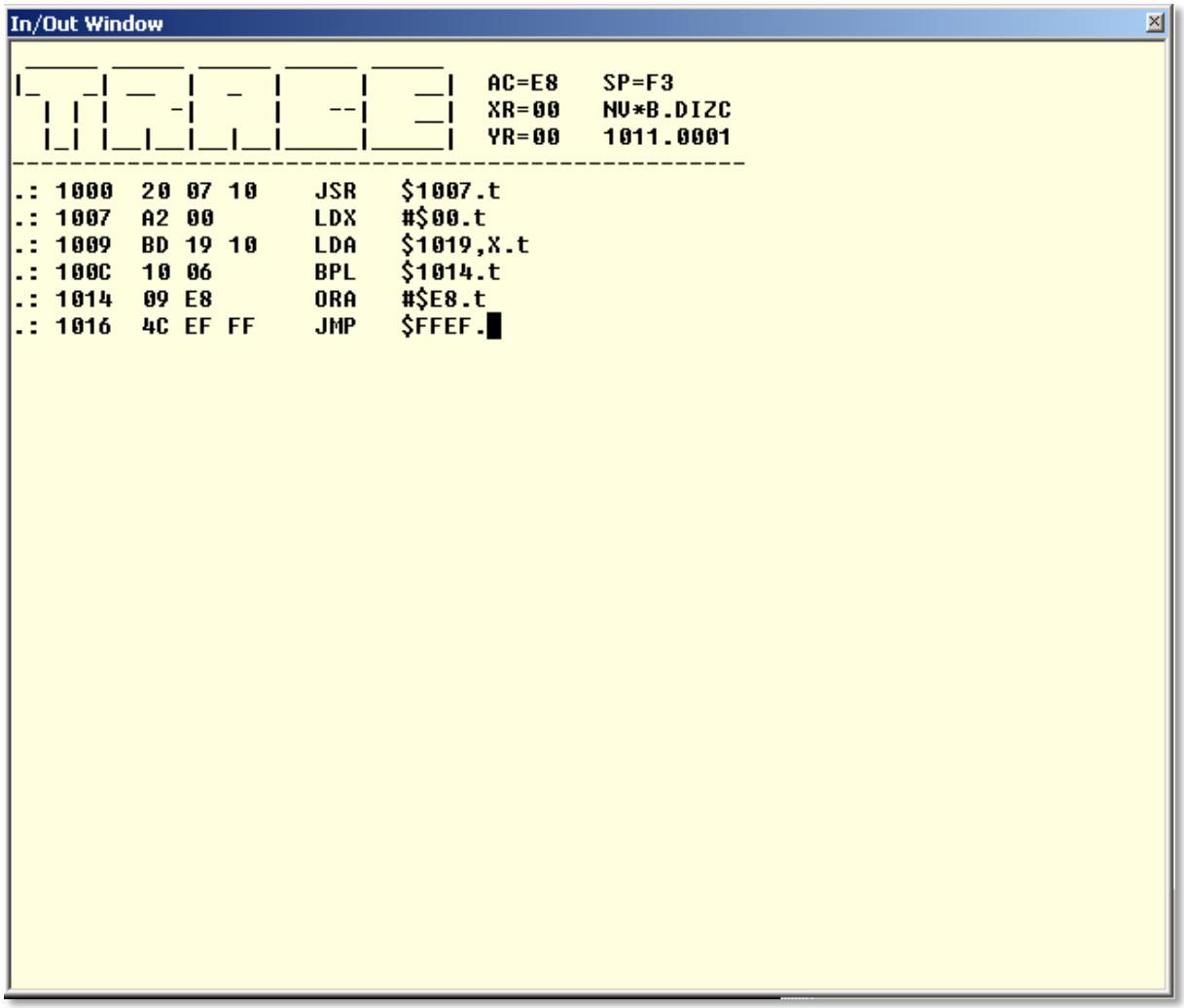
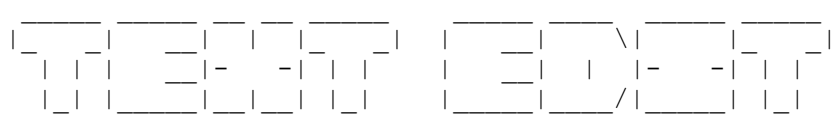


Figure 12



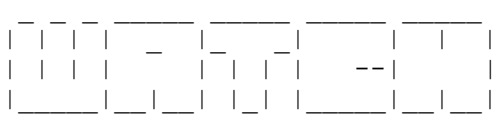
Text Edit (X) - future

X start address

Enter/edit text beginning at a specified address

```
X               Continues editing from last address
X start         enter/edit text beginning at a specified address
```

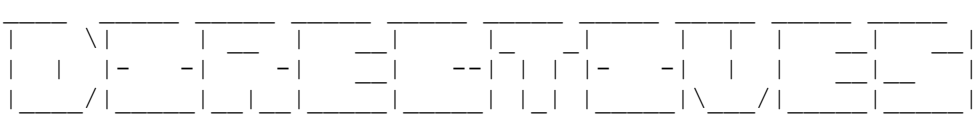
{future}



Watch (W) - future

Continuously read and then display contents of a specified address, loop until keystroke

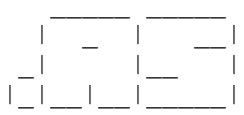
{future}



Directives

Directives are often called pseudo-opcodes. They are always to be found in column 2, where you would also find processor opcodes (mnemonics). A directive is a command to the assembler, for instance to generate data bytes or change the current program counter.

The following section contains valid directives of the monitor.



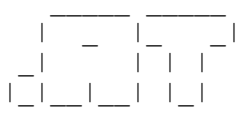
.AS -/string/

This directive allows the user to enter an entire string as data into your program. If the first character of the operand is — sign the entire string will be in negative ASCII (128 .. 256), the way the Apple 1 likes to get its ASCII characters. If the first character is not a — sign the string will be in positive ASCII (0 .. 127).

The string of characters must be surrounded by a delimiter. A delimiter can be virtually any ASCII character, which should be the same at the beginning and at the end of the string. Usually the characters / \ " or ' are used as delimiters, that is if you can type \ of course. The delimiter you use may not occur in the string, otherwise you'll get an error message.

```
1000    .AS /ABC/          generates 41 42 43
1010    .AS !123!          generates 31 32 33
1020    .AS -"ABC"         generates C1 C2 C3
1030    .AS -'1234567890'  generates B1 B2 ... B3 B0
```

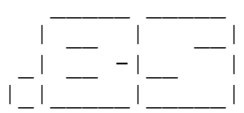
Please note that the assembler does not allow you to use more than one operand.



.AT -/string/

This directive is almost identical to the .AS directive. The only difference is the polarity of the last generated character, which is opposite from the rest of the string. This opposite polarity can be used by the software to signal the end of the string to be printed.

```
1000    .AT /ABC/           generates 41 42 C3
1010    .AT !123!           generates 31 32 B3
1020    .AT -"ABC"          generates C1 C2 43
1030    .AT -'1234567890'   generates B1 B2 ... B3 30
```



.BS expression

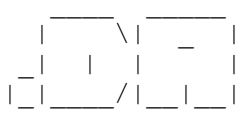
This directive skips the number of bytes indicated by the *expression*. The *expression* may not contain forward referenced labels as the assembler would not know how many bytes to skip.

Skipped bytes are not altered! The only action to occur is the current program counter to be incremented by *expression*.

You can use .BS to declare RAM addresses easily, e.g., Zero Page locations.

```
1000          .OR $0080
1010 POINTER  .BS 2     A 2 BYTE POINTER
1020 COUNT    .BS 1     A 1 BYTE COUNTER
1030 BUFFER   .BS 10    A 10 BYTE BUFFER
1040 FLAG     .BS 1     A 1 BYTE FLAG
```

You may use any value for *expression*, e.g. a non-useful value such as $FFFF is value.



.DA *expression*

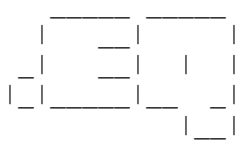
With this directive you can include data bytes and words into your program. You can include as many operands as you like (until the program line is full), all separated from the previous one by a comma. Any combination of word, LSB and MSB operands is possible.

For byte data the *expression* must be preceded by a < or a > symbol. The < symbol will use only the LSB of the 16-bit *expression*, whereas the > symbol will use the MSB. Word data is generated with LSB first (little endian). This is the way the 6502 likes it best.

```
1000    .DA $1234                 generates 2 bytes, 34 12
1010    .DA >$1234                generates 1 byte, 34
1020    .DA <$1234                generates 1 byte, 12
1030    .DA $1234,<$5678,>$9ABC   multiple operands, 34 12, 78, 9A
```

The data directive (.DA) and all immediate addressing mode instructions normally use the < symbol to identify the 8 least significant bits of the expression. If you need the most significant bits however you can substitute the <symbol by the >symbol.

```
    .DA $1234     16-Bit data result ($34 $12)
    .DA <$1234    8-Bit data result LSB ($34)
    .DA >$1234    8-Bit data result MSB ($12)
    LDA <$1234    Load Accu with LSB ($34)
    LDX >$1234    Load X with MSB ($12)
```



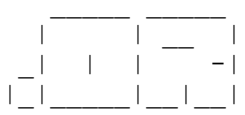
.EQ expression

Normally a label will get the value of the Program Counter at the beginning of the line on which the label is assigned. This behaviour can only be changed by this directive. Column 1 must contain a global label when the second column contains the .EQ directive. You can't use the .EQ directive on local labels.

The label in column 1 gets the value which is represented by *expression*. This *expression* may not contain forward referenced labels!

```
PRBYTE .EQ $FFDC
ECHO   .EQ $FFEF
CR     .EQ $8D
SPACE  .EQ " "
CHOUT  .EQ ECHO     CHOUT will get the value $FFEF
```

It doesn't matter what type of data is assigned to a label. It may be an address, a constant value, an ASCII value, or whatever. You can however only assign values to labels. This means that you cannot assign a string of characters to a label.



.OR expression

This directive sets the starting address of your program, or parts of it. It also sets the target address to the same value (See .TA directive). If this directive is omitted the default starting address will be $1000. See DEF_ORG in Constants.65s

You can set the starting address *expression* anywhere in memory. However you can not store code just about anywhere in memory. If you haven't set a user safe area you can only generate code to the range from $0200 (DEF_OBJLOW) to LOMEM, otherwise you'll get a memory error.

You may change the starting address of your program as often as you like. Every block of memory generated is reported by the assembler, which makes it easier for you to locate your code.

The *expression* may not contain forward referenced labels.

```
1000       .OR $0080    ;START ZP DEFINITION
1010 PNTR  .BS 2
1020 CNTR  .BS 1
1030 BFFR  .BS 10
1040       .OR $0300    ;START CODE HERE
1050       NOP
1060       NOP
1070       .OR $0400    ;MORE CODE HERE
1080       NOP
1090       NOP
1100       NOP
```

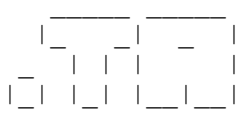
(.BS directive does not generate code)

```
.s

-------------------------------------------------
.ORG ->$0080.$008C
.ORG ->$0300.$0301
.ORG ->$0400.$0402

-------------------------------------------------
0 COMPILE ERROR(s)
.
```

Figure 13



.TA expression

You can't generate code in protected memory. Normally you can only generate code from address $0200 until LOMEM, the rest of memory is protected.

You may indicate a user safe area by setting the memory addresses USR_OBJLO and USR_OBJHI to declare another part of memory to be safe. However you're in charge there, you're the one who should be absolutely sure that it IS safe! Setting these two values doesn't automatically make the area safe, it only allows the assembler to store generated code there.

But what if you want to create a program which should run in a protected area, let's say from address $E000? Simple, you set the .OR to $E000, and change the target address to a safe area, e.g. $0300 (see example below).

The assembler will generate all addresses as if it was actually using address $E000. However the code is stored at address $0300. Obviously this will result in a program which does not work as is. You'll have to move the program to the intended destination before it can be run.

Moving the code to its final destination can be done with the COPY (C) command, or by saving it to file and loading it at a different address.

The *expression* may not contain forward referenced labels.

```
1000            .OR $E000
1010            .TA $0300
1020 START      NOP
1030            NOP

START
------------------------------------------------------------
ORG -> $0300.$0301     this proves that the right target address is used
------------------------------------------------------------
0 COMPLIE ERRORS

VALUE START              here's some more proof
57344 -8192 $E000 %1110.0000.0000.0000
```

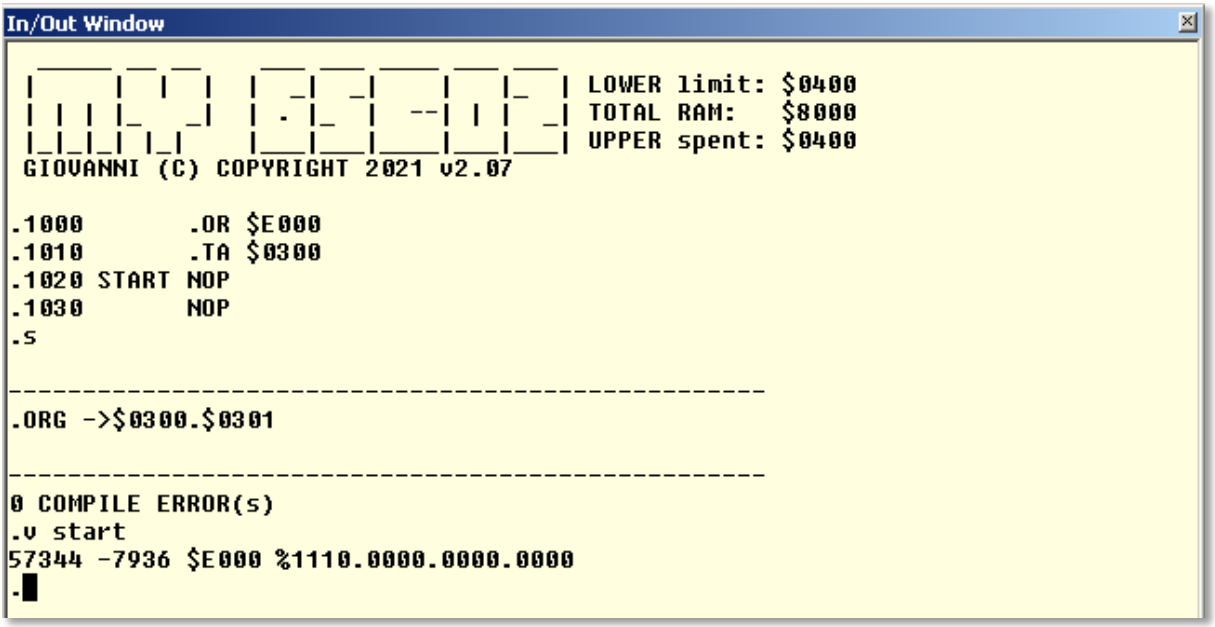
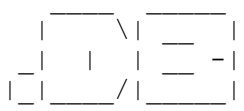

In/Out Window
LOWER limit: $0400
TOTAL RAM: $8000
UPPER spent: $0400
GIOVANNI (C) COPYRIGHT 2021 v2.07
.1000 .OR $E000
.1010 .TA $0300
.1020 START NOP
.1030 NOP
.s
.ORG ->$0300.$0301
0 COMPILE ERROR(s)
.v start
57344 -7936 $E000 %1110.0000.0000.0000

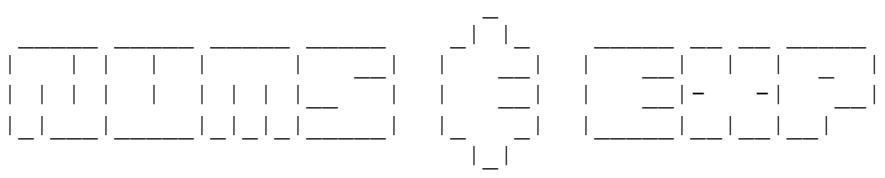
Figure 14



.DB expression

{Not implemented yet}

Single byte definition.



Numbers and Expressions

Many commands and operands accept numbers and expressions. An expression is simply a mathematical combination of several numbers.
Any number is limited to 16-bits only. Enter larger numbers than that and you'll be treated with a range error.
You may precede any number with a negative sign to make it negative (2's compliment).

Wherever the Assembler expects a number you can supply it in one of the following options:

Decimal numbers

Start with a digit from 0 to 9, and may only contain these numbers.

```
123
-500
```

Hexadecimal numbers

Start with a dollar symbol, and contains only normal digits 0 to 9 and extra digits A to F.

```
$10
$FFEF
-$100
```

Binary numbers

Start with a percent symbol and may contain only the digits 0 and 1. You may place dots anywhere in a binary number to make them easier to read. The assembler simply ignores the dots.

```
%1000.1101
%1111100101110101
%1111.1001.0111.0101    same value as above!
-%1000
```

Positive ASCII

Generates values between 0 and 127, depending on the character enclosed in single quotes.

```
'A'     TRANSLATES TO $41
'2'     TRANSLATES TO $32
```

Negative ASCII

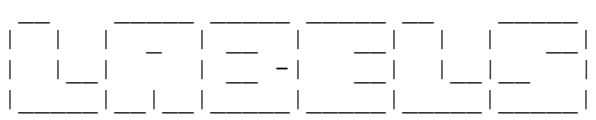
Generates values between 128 and 255, depending on the character enclosed in double quotes. Please note that this is the native Apple 1 mode to represent ASCII characters!

```
"A"     TRANSLATES TO $C1
"3"     TRANSLATES TO $B3
```

Current PC

A single dollar symbol, not followed by a legal hexadecimal digit, will result in the current program counter value. The value used was the program counter at the start of the current source line.

```
$
```



Labels

Simply the label's value is used. Only assembly pass 1 allows the use of labels which are not defined yet. In that case we speak of forward referenced labels.

An undefined label during pass 2 of the assembly will result in a definition error.

In case of forward referenced labels we can not know their actual value during pass 1 of the assembler. Therefore some instructions which can use shorter addressing modes will fall back on the worst case scenario and use long addressing mode instead.

Expressions can be used to combine 2 or more values to get a new final value. You can use one of the 4 basic operators in expressions:

+ Addition

- Subtraction

* Multiplication

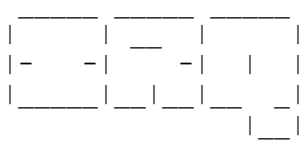
/ Division

All expressions are evaluated from left to right. No priority is given to multiplication and division over addition and subtraction unlike in normal math. Parentheses can not be used to change priority in expressions. Overflows in expressions are ignored and the result is always truncated to 16-bit integers.
You can mix any legal number form with any number of operations.

```
1234+$1200    RESULTS IN $16D2
$F000-123     RESULTS IN $EF85
%101*2        RESULTS IN $000A
$5678/4       RESULTS IN $159E
LABEL*2       RESULTS IN THE VALUE OF LABEL TIMES 2
```

All results are 16-bits long integers. No errors are reported if the result exceeds the limits of a 16-bit number, only the least significant 16-bits are used as result. This may sometimes give some strange results, especially if the expression contains multiple operations.

For example 7/8*100 results in 0. This is because 7/8 is 0.875, which is truncated to 0 caused by the integer division. You'll get a much better result by rewriting the expression to 100*7/8, which is still an integer.



Interrupts

Interrupts pass though RAM vectors to allowing you change the locations.

```
USIRQ       .RS   2          ;User IRQ vector
USBRK       .RS   2          ;User BRK vector
USMNI       .RS   2          ;User NMI vector
```

Break Vector

Change USBRK to point to your routine. Then return back to BRK_RETURN routine. Otherwise ensure you have below to mirror the initial 3 push when the IRQ/BRK was called.

```
PLP         ;pull off flags
PLA         ;PC low
PLA         ;PC high
```

IRQ Vector

Change USIRQ to point to your routine. Then return back to IRQ_RETURN routine. Otherwise ensure you have below to mirror the initial 3 push when the IRQ/BRK was called.

```
PLP         ;pull off flags
PLA         ;PC low
PLA         ;PC high
```

Reset Vector

This can be changed. Ensure that the new vector has a CRC that's EOR with $A5

```
USRRS       .RS   3          ;User RESET vector

LDA   #<YOUR_RESET           ;your reset user vector
STA   USRRST                 ;store in RAM
EOR   #$A5
STA   USRRST+2
LDA   #>YOUR_RESET
STA   USRRST+1               ;press reset
```

Note: RESET_RETURN will reset the IRQ,NMI and IRQ vectors.

 _____ _____ _____ _____ _____ __   _____ _____
			_				__	__
--							- -		__	__	-
_____	_____	_	_	_	__	__	_____	_____	_____	__	__

Code Complier Directives

MYWYM

This is setting specifically for my board and the addressing of chip. Currently my settings are

```
VIA     = $4000                 ;My board settings
ACIA    = $6000
CS      = $2000                 ;used for LCD Display select
```

Change to suite yours.

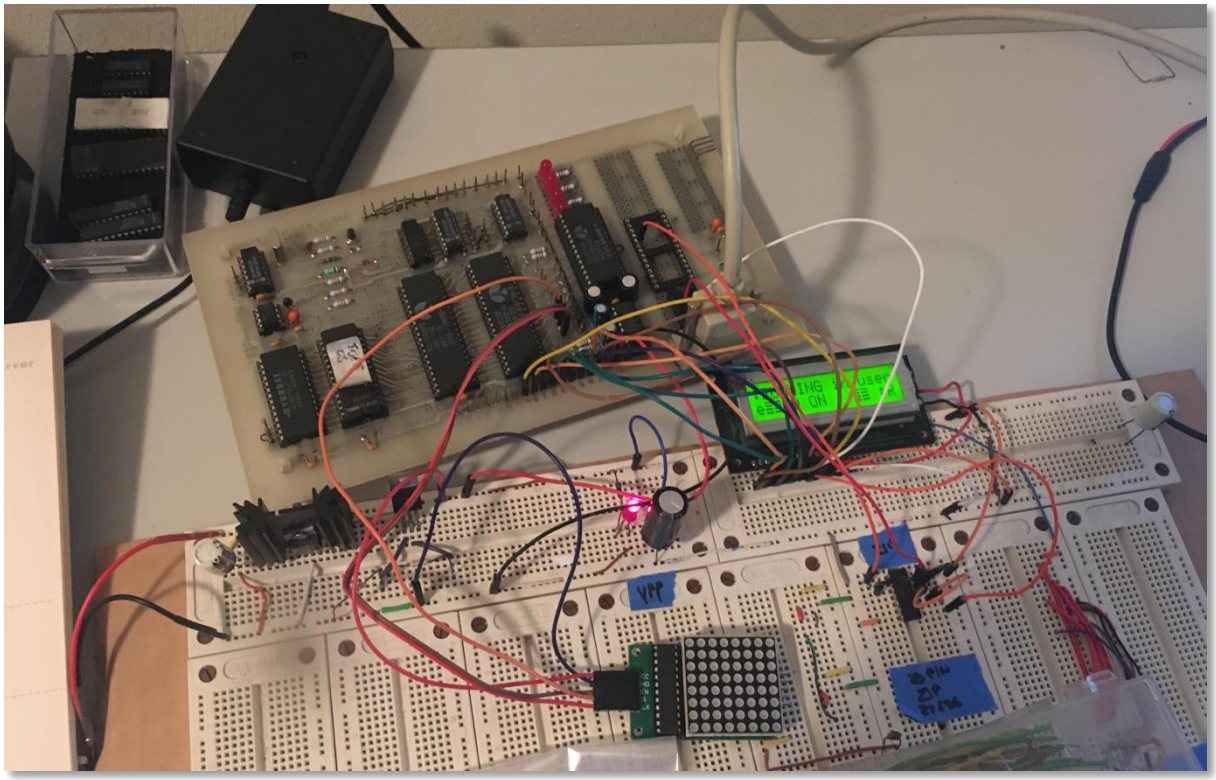


Figure 15

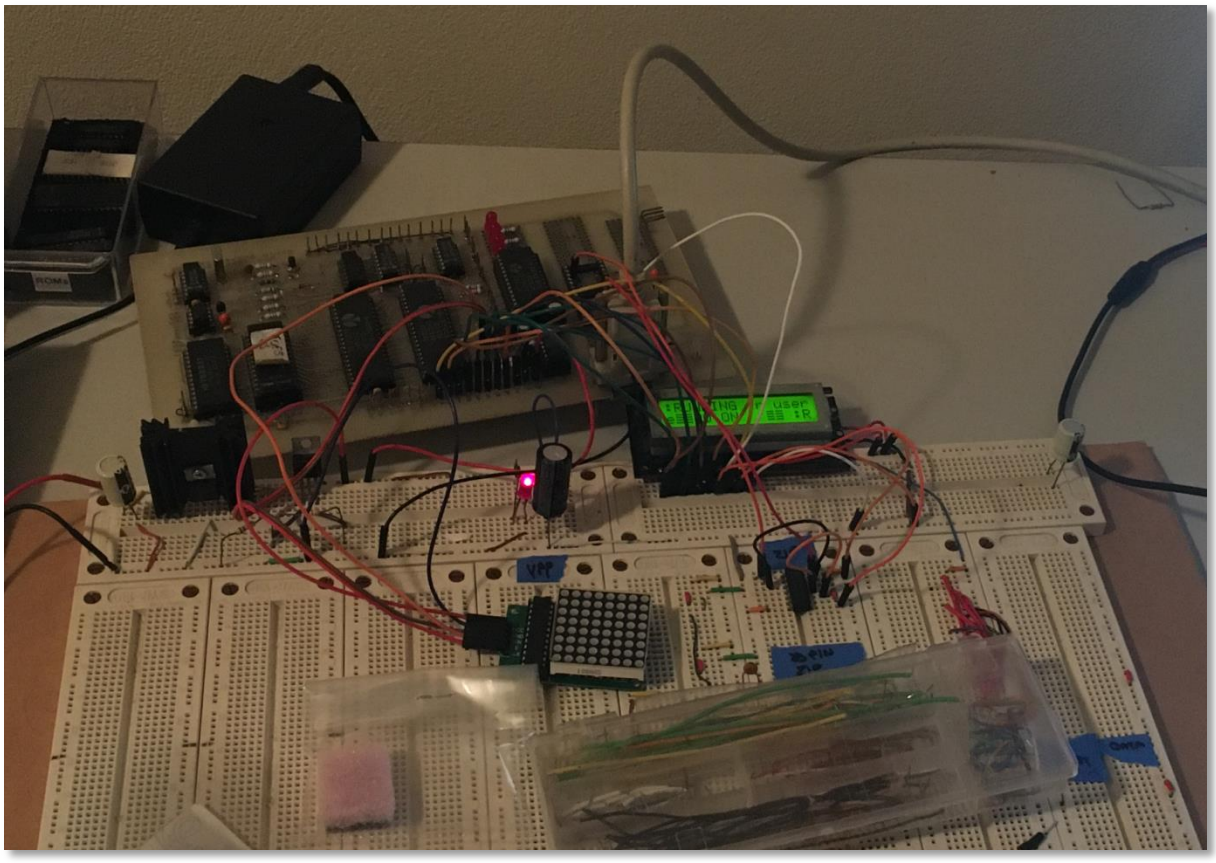


Figure 16

SYMON

These settings are for Symon Simulator https://github.com/sethm/symon

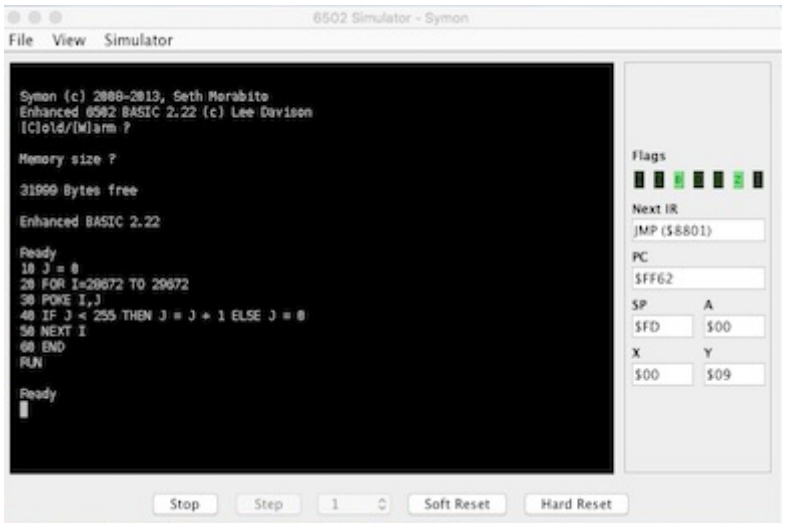


Figure 17

LCD Routines

Library Routines for a 16x2 LCD display - HD44780-compatible controllers. The library is found in General lib.65s. The routines are:

```
LCD_LINIT       ; Initialisation
LCD_SCROLL      ; LCD Scroll
LCD_CLR         ; Clear screen
LCD_HOME        ; LCD home
LCD_BUSY        ; Wait for LCD busy bit to clear
LCD_WRCHAR      ; Print character on LCD
```

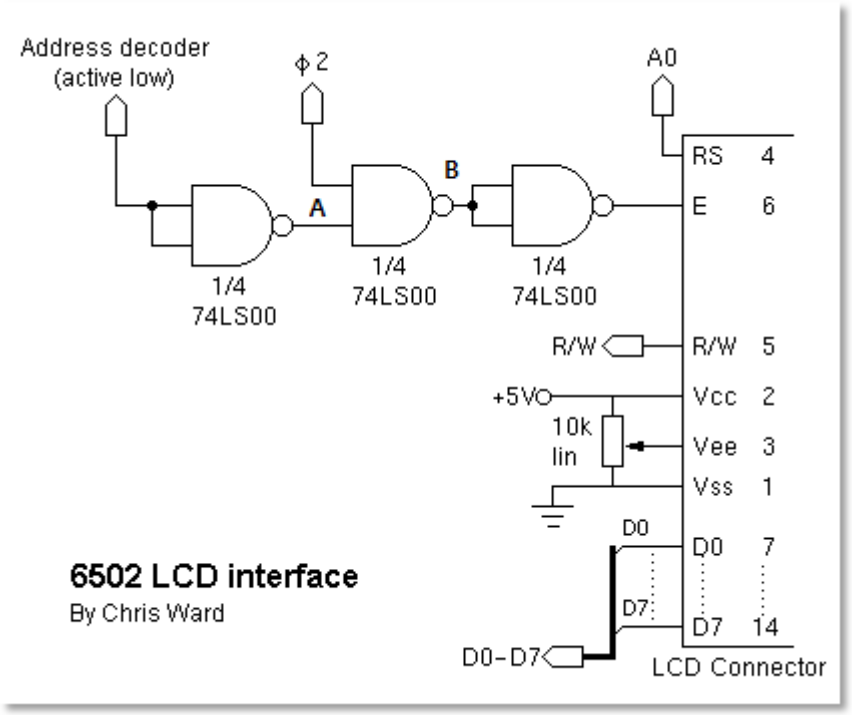
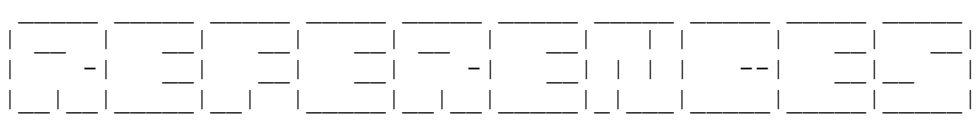


Figure 18

CS	A	02	B	E
0	1	0	1	0
1	0	0	1	0
0	1	1	0	1
1	0	1	1	0

E - enable (This loads the data into the HD44780) - on the falling edge.



References

Text graphics Generator:

http://patorjk.com/software/taag/#p=display&h=3&v=0&f=Rectangles&t=Type%20Something%20

LCD References

http://www.6502.org/mini-projects/optrexlcd/lcd.htm

Data Sheet

https://www.sparkfun.com/datasheets/LCD/HD44780.pdf

Intel Hex

https://en.wikipedia.org/wiki/Intel_HEX

Text Font