The A1-Assembler

You can of course use my full featured SB-Assembler to create your own programs for the Apple 1. However you can also give my A1-Assembler a try. My A1-Assembler is a 4k program which actually runs on the Apple 1, or any of its replicas or emulators. If you have enough RAM memory you can create quite reasonable sized programs with it.
The assembler includes an editor which can be used to enter and edit your source code. After that a simple key press is all you need to assemble the program into directly runnable code. No need for data conversions or serial data transfers to get the program over to the Apple 1, you can immediately execute the program after it is assembled.

OK, I admit that the editor is rather primitive by today's standards. And although the assembler's features are copied from its bigger brother, the SB-Assembler, it lacks some of the functionality of the SB-Assembler which would make life a little easier.
Remember though that the Apple 1 itself is also a rather primitive computer. Simply consider the A1-Assembler like going out on a camping trip in a small tent while owning a nice luxurious house.

You can download a copy of the A1-Assembler from the downloads page. The download package contains the entire source file for the A1-Assembler, which can be assembled by the SB-Assembler. It also includes assembled versions to run from 3 different locations in various file formats. The 3 pre-assembled locations are $7xxx, $9xxx and $Exxx. However you can easily alter the source to create a file which will run from an entirely different address.
Two versions of the A1-Assembler exist, one for the 6502 and one for the 65C02. It appears that most Replicas are equipped with the 65C02 which has some extra instructions and addressing modes.
Finally you will also find some sample programs in the download package. You can upload these to your machine after which you can assemble and run them.

Please note that the 65C02 version of the A1-Assembler will only run on a 65C02 processor! You will run into all sorts of strange errors if you try to run the 65C02 version on a 6502 processor.

Compatibility with the SB-Assembler

Like I already said, the features of the A1-Assembler are copied from the SB-Assembler. This means that if you know how to work the SB-Assembler, you will have no problems knowing how to work the A1-Assembler. Only the editor would be new to you.
To put it in other words: A program written on the A1-Assembler can be transferred to the PC and can be assembled by the SB-Assembler without any changes. Well, to be honest this is not entirely true, you may have to change up to three things:

- The A1-Assembler can only assemble 6502 or 65C02 code, whereas the SB-Assembler has to be informed to use the 6502 or 65C02 cross overlay. So you'll have to add the line `.CR 6502` or `.CR 65C02` somewhere at the start of the source on the PC.
- If you don't specify the `.OR` address the A1-Assembler will use $0280 per default, whereas the SB-Assembler would use address $0000 as default. So you'll have to add the line `.OR $0280` somewhere at the start of the source on the PC if you want the code to be generated for address $0280 too.
- You'll have to tell the SB-Assembler in what file to store the generated code and what format to use. The A1-Assembler always stores the generated code in RAM memory. So the SB-Assembler needs the `.TF` directive somewhere at the start of the source on the PC.
  Also note that for the same reasons the SB-Assembler doesn't allow the use of the `.TA` directive.

Going the other way is also possible, starting the project with your favourite editor and assembling it with the SB-Assembler until you are ready to transfer the program back to the A1-Assembler. However you'll have to be careful not to use features which are unknown or limited in the A1-Assembler. For instance don't use Macros or Conditional assembly, because they won't work on the A1-Assembler.
One very important remark to make is that local labels in the SB-Assembler may be formed by virtually any combination of characters, whereas the A1-Assembler only accepts local labels which are numbered from 0 to 99. Furthermore local labels on the A1-Assembler may not be located more than 255 bytes away from its global label.

Let's Get Started

Here's a small example of what the A1-Assembler can do. It demonstrates some of its features and shows you how to use the A1-Assembler. Prior knowledge about the SB-Assembler can be helpful, but is not necessary to understand this example. Knowledge about 6502 assembly language is required however, I will explain how the A1-Assembler works, I won't teach you 6502 assembly though. If you want a nice reference to 6502 assembly you can have a look at Andrew Jacobs's site. There you'll find information of both the 6502 and the 65C02.
All of the A1-Assembler's features will be explained later. This demonstration may also be helpful to see the actual features in use after reading the detailed descriptions.

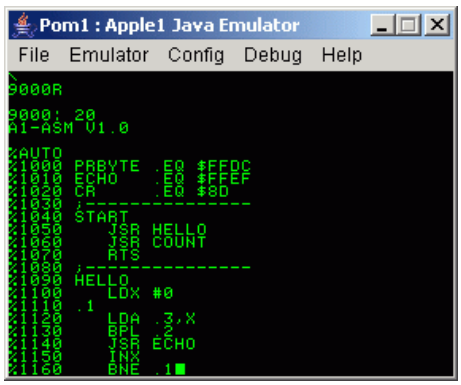
First of all you'll have to start the A1-Assembler somehow. Owners of the Achatz A-One are in luck, there the computer came with the A1-Assembler in ROM at address $9000. Real Apple 1 owners can use the E0 file to load into block $E000, which usually holds the Apple 1 Basic interpreter. Others may load the 70 file into RAM block $7000. No matter which file you use, you can always start the assembler by running it from the first byte of the block (9000R, or E000R, or 7000R, or any other self assembled variant).

Starting the A1-Assembler like that will initialize the memory organization and erase any source code from memory. A % symbol is used as prompt in the A1-Assembler. You type your commands behind this prompt.

memory. A % symbol is used as prompt in the A1-Assembler. You type your commands behind this prompt.
Now you can start entering the program below:

```
%AUTO
%1000 PRBYTE .EQ $FFDC
%1010 ECHO   .EQ $FFEF
%1020 CR     .EQ $8D
%1030 ;---------------
%1040 START
%1050    JSR HELLO
%1060    JSR COUNT
%1070    RTS
%1080 ;---------------
%1090 HELLO
%1100    LDX #0
%1110 .1
%1120    LDA .3,X
%1130    BPL .2
%1140    JSR ECHO
%1150    INX
%1160    BNE .1
%1170 .2
%1180    ORA #%1000.0000
%1190    JMP ECHO
%1200 .3
%1210    .AT -/HELLO WORLD/
%1220 ;---------------
%1230 COUNT
%1240    JSR .2
%1250    LDX #0
%1260 .1
%1270    TXA
%1280    JSR PRBYTE
%1290    LDA #" "
%1300    JSR ECHO
%1310    INX
%1320    CPX #10
%1330    BCC .1
%1340 .2
%1350    LDA #CR
%1360    JMP ECHO
%1370  (hit ESC here)
```

Wait a minute, wait a minute! I'm only human, I make typing errors every now and then. How can I correct them and continue afterwards?



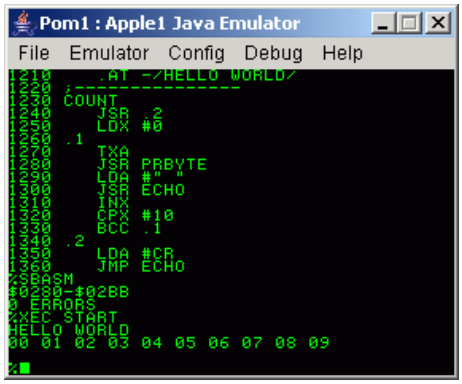
It's good of you to ask. If you notice the typing error immediately you can "back space" by pressing the "_" character, just like in the monitor. Note: The _ symbol will not be printed.
The first command of the listing above started the auto line numbering of the editor (Don't type the %, it's the prompt). The line numbers are entered for you by this auto line numbering feature. The only thing you type is what follows the line number on each line. After pressing Enter you can enter the next line until you have typed in the entire program. There, or at any time when you make a mistake, you can press ESC to cancel your current line and cancel the auto line numbering feature.
After that you can simply restart auto line numbering by entering the AUTO command again to retype your last program line, which was not saved because you pressed ESC. You can also enter the line number you want to correct directly, followed by the correct source text.
If you then want to restart auto line numbering from the next program line you simply enter the AUTO command. However if the line you just edited was not the last line of your source, you may have to restart auto line numbering from the next free line number. In that case simply type AUTO, followed by the desired line number.

After you have entered the entire program you may want to check your work. Simply type LIST to see the entire program scroll by. You can pause and resume the output by pressing almost any key. Hitting ESC will abort the listing. If you find any errors in your source simply retype the line, including its line number. The new line will overwrite the existing line in the source which has the same line number.
By using a line number which is in between two other line numbers you can insert lines you may have forgotten. Please note that the line numbers are only used to enter your source in the right order.

When all is entered correctly you can assemble your file, and if everything is OK you'll see the output similar to the listing below. When you see that there are no errors found in your source you may want to start your brand new program. Simply type XEC START to start your program. Please note that START refers to the label START in your source file.

```
%SBASM
$0280.$02BB
0 ERRORS
%XEC START
HELLO WORLD
00 01 02 03 04 05 06 07 08 09
```

%



If you are familiar with the SB-Assembler you are half way there of successfully using the A1-Assembler. All you have to do is learn the few simple editor commands and take notice of the differences/limitations compared to its bigger brother the SB-Assembler.
If you are not familiar with the SB-Assembler yet you'll have to read a bit more. Most of the explanations can be found on this very page. Other, more detailed explanations may be found in the user manual pages of the SB-Assembler. But remember that the A1-Assembler is not as versatile as the SB-Assembler.

Source Text

Your entire program must be captured in a source text. Each line in your source text starts with a line number and furthermore contains up to 4 more columns.

We need line numbers because of the primitive nature of the Apple 1's terminal and the editor of the A1-Assembler. I chose the use of self assigned line numbers as opposed to sequential line numbers. Both systems have their own pros and cons.
Sequential line numbers require less intervention from the programmer. They are always automatically generated and renumbered if new lines are inserted or when existing lines are deleted. Sequential line numbers require less memory in the source text because we don't have to store the line numbers in the source.
Self assigned line numbers on the other hand are sometimes easier for us humans. A routine starting at line number 9000 will remain at line number 9000, even when you insert or delete multiple lines below it. Thus it is often easier to find our way around the source text. Remember that the editor has no search capabilities to quickly find a specific place in your source text.

The line numbers are only there for the editor. The assembler couldn't care less about them. For instance you can't GOTO a specific line number within your program. Line numbers simply determine where lines are placed in your source text, no more, no less.
If the line number does not exist yet it is inserted in between lines with lower line numbers and lines with higher line numbers. If there are no higher line numbers in your source text the new line will be appended at the end of the source text.
If the line number already exists in your source text the old line will be replaced by the new line.
You can delete a single line by simply typing its line number with no text following it. A special command is available to delete multiple lines at a time.

Each line number is followed by at least one space. The first space after the line number is not part of the source text, it is only there to make the source text easier to read.

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 semi-colon 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!

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.

directives further down this page.
A mnemonic always consists of 3 characters. Please refer to a 6502 programming manual to find all possible mnemonics. The A1-Assembler uses standard 6502 mnemonics.
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 put optional comments at least 8 spaces behind the instruction, otherwise your comment may be mistaken for an operand. An alternative is to use a semi-colon as a comment delimiter.

```
1000      ROL
1010      ROL MEMORY
1020      ROL        THIS IS A COMMENT
1030      ROL ; OR COMMENT LIKE THIS
```

The forth column is optional and is always regarded as comments. As always it starts one space after the previous column. Unlike many other assemblers the A1-Assembler does not require you to use a semi-colon to start a comment in column 4. You may do so if you like, but you don't have to.

```
1000 LABEL LDX #0    CLEAR X
1010       LDY #0    ; CLEAR Y
```

Your source text is stored in memory starting from LOMEM. It may grow up to HIMEM. Allowing your source text to grow until HIMEM is not very wise though because that would leave no room for the symbol table which is created during assembly.
The MEMORY command can be used to view or alter these special boundaries.

The editor packs your program lines before it stores them into the source text to save some precious memory. First of all the line number is transferred into a 16-bit binary number. That way any line number will occupy only 2 bytes. This also means that the highest possible line number is 65535.
Multiple adjacent spaces and dashes are packed into 1 byte. This one byte is simply a counter, telling us how many adjacent spaces or dashes were used.
A maximum of 63 spaces or 63 dashes can be packed into one single byte. In the rare cases where you need more an extra counter byte is simply started when the previous one exceeds 63.
Why did I only pack spaces and dashes? To keep it simple I only had 4k to work with. And because spaces and dashes are the characters most likely to be used in clusters.

```
1000 ;--------------------------------
1010 ; PACKING EXAMPLE
1020 ;--------------------------------
1030 TEST   LDA #0     CLEAR COUNTER
1040        STA COUNT
```

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 A1-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 contain 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 cause 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.

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)
```

Commands

The A1-Assembler contains an editor which can be controlled by some commands. All commands consist of only one character. Below you'll find all commands completely named. However only the first character is important. Everything else you type until the end of line or first space is ignored.
For instance the AUTO command may be typed as a single A, as AUTO, or even as APPLE-1.

AUTO *linenum*,*increment*

The AUTO command starts the auto line numbering feature.
If no operands are given auto line numbering will start from the last entered line number + current increment. Or if no line has been entered before it will start from line number 1000, with an increment of 10.
You may use *linenum* to start the auto line numbering from the 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 ESC will cancel auto line numbering and the current unfinished line. Simply hit ESC when you're done entering your source code or when you make some typing errors which you can't correct with the back space key. Typing AUTO again will generate the same line number you have just cancelled to allow you to start from scratch with this line.

It goes without saying that you do not have to use AUTO line numbering if you just want to enter one or two lines somewhere in your program. Simply type the appropriate line number after the prompt, followed by your source text.

The value of *increment* is limited to 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 *source*,*destination*,*length*

This command can be used to copy a part of memory to another destination. All three parameters are mandatory, you can't skip any of them.
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 in page 0. The COPY command uses 6 bytes there as temporary storage ($E9 to $EF). Overwriting these values by a copy will very likely crash your system.
You should also be aware of the fact 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 (See `.TA` directive). After assembling your code you can move the code to the desired destination.

DELETE *begin*,*end*

Use this command to delete multiple lines at a time. Be careful though, undo is not possible. Once deleted the lines are gone forever!

Both the *begin* and *end* parameters are optional. However you'll have to enter at least one parameter for safety reasons.

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

HELP

This command may not do exactly what you would expect it to do. It only writes the text `WWW.SBPROJECTS.COM`. That's where you can go for this user's manual.

LIST *begin*,*end*

This command lists your source to the screen. If no parameters are given 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
```

You can pause the output of the listing by pressing almost any key. The ESC key aborts the listing.

The LIST command has one extra option. 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 can then be used by the SB-Assembler.

MEMORY *lomem*,*himem*

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

With no parameters this command will show you the current LOMEM, HIMEM and end of source address. Your source file starts at address LOMEM and may extend almost to address 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 HIMEM.
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 command to find out what part of memory to save to cassette in order to store your source text. LOMEM will be the start address and end of source will be the end address to write.

Generated code can be stored from address $0200 up to LOMEM, unless you have set the user safe area which can be set with the zero page addresses USR_OBJLO ($F8+$F9) and USR_OBJHI ($FA+$FB).

```
%MEMORY
$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! I didn't have the memory resources to make the settings fool proof. I don't consider you to be a fool, so I trust you to enter sensible values for LOMEM and HIMEM. Sensible values are from address $0200 up to the last available RAM (excluding the part of RAM which might hold the A1-Assembler itself).
Any other values will probably crash your computer sooner or later!

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

```
%MEMORY $1000,$8000
$1000.$8000
$1000
```

NEW

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

OLD

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 the NEW command!

RENUMBER *from*,*first*,*increment*

From time to time you may want to renumber your source, or part of your source. Usually you want to do that to tidy up a bit, or to make room for more than a few new source lines between two other lines.
For that purpose you can use the RENUMBER command.

The *from* parameter determines the line from which to start renumbering. If you omit it you will renumber your entire program.
first will be the first new line number to be used for the renumbered part of your source. If this line number is omitted the default AUTO line number will be used (1000).
Finally the *increment* parameter will determine the increment of the renumbered part of your source. If it is omitted the default increment of 10 will be used. The valid range for *increment* is from 1 to 255.

You can't set *from* higher than *first*, otherwise you may get duplicate line numbers which would definitely confuse the editor.

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
```

SBASM

This is what we came here for. 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.
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.

The name of this command may sound a bit silly, this is the A1-Assembler, not the SB-Assembler. Well to be honest, the A command was already in use, so I had to find a different command. Since I'm used to typing SBASM during my assembly work on the PC, I decided to use the same command for the A1-Assembler.

```
%SBASM
$0280.$036D
$0500.$0537
0 ERRORS
```

In the example above two blocks of memory were filled by the assembler. The first block running from address $0280 to $036D (inclusive), and the second block running from address $0500 to $0537.

VALUE *expression*,*expression*

This command can be used to view the value of labels, convert numbers from one radix to another, or even to do some simple calculations.

```
%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
```

WOZMON

This command simply transfers you to the Woz monitor of the Apple 1. You can re-enter the A1-Assembler without affecting your source text by running the assembler's starting address + 3. So if you originally started the A1-Assembler from address **9000R**, you can resume by entering **9003R** in the Woz monitor.

XEC *expression*

With this command you can start your new, or any other program in memory. This program can return control to the assembler by executing a final RTS. However this would not work if your program was run directly from the Woz monitor!

The *expression* determines the starting address. Usually you would start your program from its first address. If you assign a label to the first address of your program you may even use that label to start your program.
If you omit the *expression* XEC will run from the last given start address. This is useful when you are debugging your program. You can make some changes to the source, assemble the source and then execute the new code simply by typing XEC.

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.

.AS -/*string*/

This directive allows you 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 so called delimiter. A delimiter can be virtually any ASCII character, which should be the same at the beginning an at the end of the string. Usually the characters / \ " or '

character, which should be the same at the beginning an 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 A1-Assembler does not allow you to use more than one operand after .AS, unlike its bigger brother the SB-Assembler.

.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*. Therefore the *expression* may not contain forward referenced labels, otherwise the assembler would not know how many bytes to skip.
Skipped bytes are not altered! The only thing that happens is that the current program counter is incremented by *expression*.

You can use .BS for instance to declare RAM addresses easily (like i.e. 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 as *expression*, also quite silly values like $FFFF, the A1-Assembler couldn't care less.

.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
```

---

label **.EQ** ***expression***

Normally a label will get the value of the PC 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 can not 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 $0280.
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 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
%SBASM
$0080.$008C      use of ZP memory (.BS directive does not generate code)
$0300.$0301      first part of your program
$0400.$0402      second part of your program
0 ERRORS
```

.TA *expression*

You can't generate code in protected memory. Normally you can only generate code from address $0200 until LOMEM, the rest of memory is protected.
You may indicate a user safe area by setting the memory addresses USR_OBJLO ($F8+$F9) and USR_OBJHI ($FA+$FB) 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 command, or by saving it to tape and loading it at a different address.

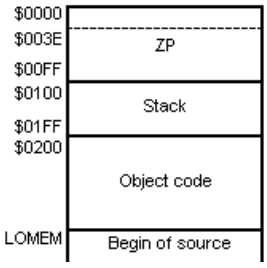
The *expression* may not contain forward referenced labels.

```
1000       .OR $E000
1010       .TA $0300
1020 START NOP
1030       NOP
%SBASM
$0300.$0301             this proves that the right target address is used
0 ERRORS
%VALUE START             here's some more proof
57344 -8192 $E000 %1110.0000.0000.0000
```

Memory Overview

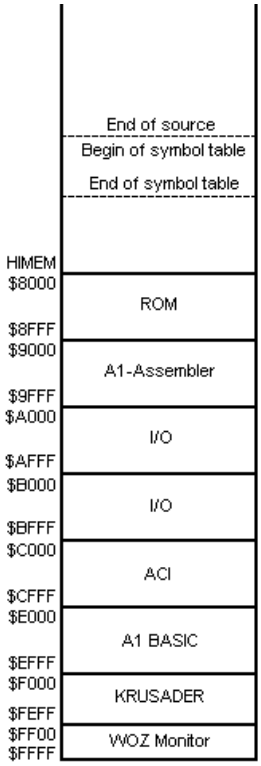
Actually, programming in assembly is all about memory. Where the code gets, what addresses are free to use, what addresses are used by the system and for what purpose, how big can the source file grow? All these questions become important if you want to get the most out of your system.

Memory use



Zero Page: The assembler uses the addresses $3E to $FF of the zero page. However this doesn't mean that you can't use these addresses in your own programs. For instance the range from $3E to $BD is the input buffer, which doesn't have to survive a user's program when you return to the assembler.
The most important registers, which should survive manipulations of a user's program upon return are $EF to $FF. Altering these locations may render your source file unreadable or may crash your system when the A1-Assembler regains control after executing the user program.

Stack page: The Woz monitor does not initialize the stack pointer. As a result the stack can be all over page $01. The A1-Assembler will initialize the stack pointer to



stack can be all over page $01. The A1-Assembler will initialize the stack pointer to $FF. I haven't actually verified it, but I don't think the assembler will need a deeper stack than 32 bytes (roughly speaking). This means that addresses $0100 to $01DF should be safe for your own programs.

Page $02 until LOMEM: This is the default safe space to store your generated code. Per default the A1-Assembler uses $0280 as the starting address. This is done because the Woz monitor uses addresses $0200 to $027F as input buffer. If you are in desperate need of memory you may start your program as low as $0220. That leaves a 32 byte input buffer during the Woz monitor, enough for most commands. However longer commands in the Woz monitor will overwrite the beginning of your program!

LOMEM until HIMEM: This area is used by the assembler to store your source text and the symbol table, which is generated during assembly.
LOMEM is set to $0600 per default, but can be changed with the MEMORY command. HIMEM is set to the highest available RAM, with a maximum value of $8000, or until we run into the code of the A1-Assembler itself.

Here's some information to give you a rough idea of what is stored in this area. Every source line will have at least 3 bytes, plus the source text itself. The first 3 bytes are a line length byte and a line number word. Thus the line number is stored as a 16-bit value. Every character in the rest of your source line occupies one byte in your source file, except multiple spaces and dashes. Multiple adjacent spaces or dashes are encoded into a single byte up to a maximum of 63 consecutive spaces or dashes. Multiple spaces are often used to align the assembler's columns, and multiple dashes are often used as separating comment lines. That's why I made the Apple 1 go through the trouble of packing these characters.
The symbol table uses only 6 bytes per assigned global label, regardless of the length of the label's name. And only 2 bytes per assigned local label. The symbol table is created during the first pass of the assembly process. It will survive until you edit your source text again. This means that you can recall the values of the labels after an assembly using the VALUE command until you start editing your source again.

A1-Assembler program: The A1-Assembler can be assembled to run in just about any location. It needs a free block of memory of 4k. Thus it may fit in block $E0 instead of Apple 1 Basic, or it can be stored in one of the unused blocks of memory in your memory map (e.g. $8xxx or $9xxx). It can also be assembled to run in the highest bank of RAM memory, block $7xxx.
But the best option of course is to run the program somewhere from ROM memory, that will leave the most free RAM for your source and program as possible.

What to do if you're short on memory? To put it simple: use as little as possible. But usually you don't have to overdo it. Don't spend hours to find a way to make your source text 2 bytes shorter. Instead start where the most can be gained.

This is obviously the length of the most often used labels. For instance if you have a label which is used 100 times in your program, it's text will be present 100 times in memory. So if you can make such a label 4 characters shorter you will earn 400 bytes.

Another tip is to make excessive use of local labels. Local labels occupy only 2 or 3 characters in your source and only 2 bytes in the symbol table.

Keep your comments as brief as possible, while maintaining readability. Abbreviate words like POINTER to PNTR, or BUFFER to BFFR. Don't over comment your programming if you're short on memory. For instance a line like `LDX #0  LOAD X WITH 0` is a perfect example of unnecessary commenting.

Don't bother about the number of spaces you use in your program, they are neccessary and even multiple spaces only occupy 1 byte of memory.

Normally the assembler can store generated code only between $0200 and LOMEM. If that's not enough you can raise LOMEM to a higher value, at the expense of the maximum length of your source file. Do this before you enter your source text because changing LOMEM and/or HIMEM will erase your program, with no normal way to undo it.

You can also set a user safe area where it is supposedly safe for the assembler to store the generated code. It's the programmer's responsibility to make sure it is really safe, the assembler simply accepts any area blindly.
You change the user safe area by changing the values USR_OBJLO ($F8+$F9) and USR_OBJHI ($FA+$FB). Per default these values are both 0, which disables it. But if you want to store some part of your program from $0100 to $01DF for instance you can simply fill USR_OBJLO with $00 and $01 (LSB first), and USR_OBJHI with $DF and $01.
You'll have to do these changes in the Woz monitor and use the A1-Assemblers' base address +3 to return to the assembler. The NEW command will not affect the user safe area settings. Starting the A1-Assembler from its first address does disable the user safe area again.

If you've entered a nice piece of work you do want to save it of course. The easiest way to do this is when your Apple 1 or replica contains a serial interface. Simply list your source using the LIST command and capture the output in your terminal program. Restoring the program later is only a matter of sending this file back to the Apple 1 through the same serial connection. Don't forget to clear any previous program, otherwise you might end

up with a mix of both new and old programs.

You can also save a block of memory in most emulators. By far the easiest way is to make a 64k snapshot of the current memory. But you can also save only the source text itself.
The MEMORY command, without parameters, will tell you where your source starts and where it ends. The first word is where it starts (LOMEM), the second word is HIMEM in which we are not interested. The next line also contains a word, which is in fact the address of the end of your source text (inclusive).

You can use this same trick when you want to store the source on an audio tape. Simply use MEMORY to find the first and last address to safe.

```
%MEMORY
$0600.$8000
$1234
%WOZ
\
C100R
0600.1234W
```

When you read your source text back LOMEM doesn't necessarily have to be the same as when you saved your source file. After changing LOMEM to the desired location simply adapt the start and end addresses for the Read command accordingly and everything will work again.

Please don't forget to re-enter the assembler on its usual starting address +3! Starting the assembler from its first address will erase your source text again.

Warning! Although the A1-Assembler tries its best to save you from crashing the system it can't protect you from a runaway user program! If you create a program which accidentally starts writing garbage data all over the place chances are that your precious source text gets corrupted. This means that you can not correct your malicious program any more. You would have to start all over again if you don't have a backup.
Therefore take the time to store your source code from time to time to prevent you from such disasters.

Error Messages

The A1-Assembler is not very talkative because of the limited memory resources of the Apple 1. Error messages are abbreviated to 3 characters to save memory. Here's a list of all possible error messages and their causes.

*** SYN ERR	Syntax error. The command you just entered did not exist.
*** LBL ERR	Label error. A label was required (.EQ directive). Illegal characters were used in a label declaration. A local label was used/declared when no global label was declared in the program.
*** RNG ERR	Range error. A constant number exceeded 16-bits. A local label was more than 255 bytes away from its global label. A branch destination was out of reach.
*** MIS ERR	A missing parameter or missing operand.
*** DIV ERR	Divide by 0 error. You tried to divide by 0 in an expression.
*** MEM ERR	Memory full, or illegal target address. Memory full can happen while entering new source lines, or while assembling when the symbol table exceeds above HIMEM. This error can also happen when your program tries to save generated code in unsafe memory locations.
*** DEF ERR	Definition error. An undefined label is used.
*** DIR ERR	Directive error. The given directive does not exist.
*** OPE ERR	Operand error. There is something wrong with the given operand for the directive or instruction.
*** MNE ERR	Mnemonic error. The given instruction mnemonic was not recognized.
*** EXT ERR	Extra definition error. The same label is declared twice in your program.
*** OLD ERR	The OLD command is not possible, for instance because you have already entered some new lines after the NEW command. Note: In order to save some code memory I decided to scrap this error in the 65C02 version of the A1-Assembler.

Tweaking The A1-Assembler

You can use one of the pre-assembled versions of the A1-Assembler as is of course. But since I included the source of the assembler you can also tweak it to your liking and re-assemble it.
All these tweaks involve adapting the files a1asm.asm and/or dec.asm. After changing the file you can re-assemble it by running `sbasm a1asm` from the DOS prompt.

Target Processor

Two versions of the A1-Assembler exist, one for the original 6502 and one for the 65C02 which is apparently used in most Replicas. You can select the desired version by selecting one of the two options below which you can find in the a1asm.asm file. Simply un-comment the one you want and comment the one you don't want, then reassemble the lot.

```
;--------------------------------------------------------------------------
; Select proper processor here

PROCESSOR       .EQ     $6502           Target processor and cross
;PROCESSOR       .EQ    $65C02           Target processor and cross
```

A1-Assembler's Starting Address

In order to set the A1-Assembler's starting address to your own preference you should alter the file a1asm.asm and re-assemble the lot.
You can choose pretty much any starting address you like. Simply change this line in a1asm.asm.

```
;--------------------------------------------------------------------------
; Select preferred target address here

TARGET          .EQ     $9000
```

And while you're at it, you can also change the output format of the generated code to your requirements. Simply uncomment or alter one of the following lines.

```
;--------------------------------------------------------------------------
; Select preferred target format here

;               .TF     A1ASM.TXT,AP1,16
;               .TF     A1ASM.BIN,BIN
;               .TF     A1ASM.HEX,INT
```

Constants
The Constants block in dec.asm holds some defaults which I chose. You're welcome to change any of these defaults if you want. I think the comments behind the default values don't need more explanation.
For instance the POM 1 emulator can't use the TAB or Ctrl keys unfortunately. If you want to use the TAB function you may change the value of label TAB to some other key (e.g. \ ).
You may also want to change the tab stop locations if you're not happy with the ones I have chosen.

```
;--------------------------------------------------------------------------
; Constants
;--------------------------------------------------------------------------

DEF_LOMEM       .EQ     $06             Default lomem page
DEF_AUTO        .EQ     1000            Default Auto line number start
DEF_INC         .EQ     10              Default Auto increment step
DEF_ORG         .EQ     $0280           Default .OR
DEF_OBJLOW      .EQ     $0200           Default lowest object address

PROMPT          .EQ     "%"             The assembler prompt character
CR              .EQ     $8D             CR character
ESC             .EQ     $9B             ESC character
BS              .EQ     $DF             Back space key
TAB             .EQ     $89             TAB character

TAB1            .EQ     8               1st tab stop
TAB2            .EQ     12              2nd tab stop
TAB3            .EQ     21              3rd tab stop
```

6502 Versus 65C02

Apparently most Replicas are equipped with the 65C02 processor. Therefore I decided to make two versions of the A1-Assembler, one for the native 6502 and one for the 65C02.
The 6502 version can run on both processor types. However it can not create 65C02 specific instructions or addressing modes, even if you run it on a 65C02.
The 65C02 version can only run on a 65C02 processor because I needed the space savings some of its new instructions had to offer. But keep in mind that programs you create using the specific 65C02 instructions or addressing modes will not run on machines using a genuine 6502!
Running the 65C02 version on a 6502 processor will cause all sorts of strange errors to happen. I didn't have the memory resources to prevent you from running the 65C02 version on a wrong processor.

You can tell the two version of the A1-Assembler apart by looking at the welcome text which is printed when you start the program.

```
A1-ASM V1.0     for the genuine 6502 version

A1-ASM C1.0     for the 65C02 version
```

One final word on the mnemonics of some of the new 65C02 instructions. Some assemblers use the mnemonics DEA and INA for the new decrement Accu and increment Accu instructions. Others simply use DEC and INC without

operands. I chose for the latter option.
On the other hand some assemblers use the DEC A syntax. This syntax will not be recognized by the A1-Assembler. BTW its bigger brother the SB-Assembler allows all possible syntaxes, DEC, DEC A and DEA all generate the same code there.

```
DEC             legal for A1-Assembler and SB-Assembler
DEC A           illegal for A1-Assembler, legal for SB-Assembler (*)
DEA             illegal for A1-Assembler, legal for SB-Assembler
```

DEC A will be interpreted as decrement the address pointed to by label A by the A1-Assembler.