Everyone Likes Some Assembling (ELSA): the native 65C816 assembler

An overview

Version 0.93

Preface

ELSA is a clone of the MAE assembler by John Harris. This was the assembler I had used since around 1996, when I discovered it and switched to it from MAC/65.

I sometimes still use MAE, but as years were passing, MAE was becoming a bit too limited to my needs. Since that assembler is apparently no longer developed, and since its author, John Harris, has refused to publish its source code, I was forced to start writing my own assembler. It of course has the flavour of MAE, the assembler I was accustomed to. But I have not used any part of MAE's code: the code is entirely my own, the ELSA assembler only mimics most of the MAE's syntax, diverging whenever I thought I had a better idea.

Also, ELSA is only an assembler compiler. Unlike MAE, it does not contain an editor or a disassembler/debugger. For that last, I am currently working on my own disassembler for 65C816, and for the editor I still use MAE.

ELSA is entirely written in 65C816 native code and makes use of the RAM past the first 64k: it stores the symbol table there, thus making it virtually unlimited. In the base 64k the program occupies around 25k.

One important similarity between ELSA and MAE is that ELSA, like MAE, was written entirely on Atari. First versions were written and compiled under MAE, later versions compiled with themselves, still however being written in MAE's excellent editor.

For years it had been my private program, not really intended for public release. But it apparently has grown so that it can be shown to other people. So it will be with the hope that it turns out to be useful to someone.

KMK/DLT
Warszawa, February-November 2020

Table of contents

I. Operation

As said in the Preface, ELSA is MAE's clone, so first of all it is good to get some acquaintance with that assembler. The MAE User's Manual you can find on the Net will supply you with the basic information on this topic:

http://www.mixinc.net/atari/mae.htm

Unlike MAE (which stands for Macro-Assembler-Editor), ELSA is an assembler compiler only, contains no editor nor debugger.

Also, unlike MAE, ELSA aborts the assembling on first error, emits a bell signal (ASCII 253), and quits to DOS. This allows you to leave the computer alone doing a larger assembly builds instead of being forced to watch the screen constantly for error messages or miss them having been scrolled up out of the display.

Unlike MAE, which compiles from the memory to the memory or from memory to an object file, ELSA compiles from the source file to the object file only. Therefore it is good to have a fast hard drive as storage.

Other requirements are:

1) 65C816 CPU operating at least at 1.77 MHz;
2) 65C816 compatible OS ROM, like DracOS (also known as Rapidus OS);
3) a DOS with OSS CLI, preferably SpartaDOS.

Recommended:

4) SpartaDOS X;
5) at least 64k of the 65C816 High RAM, also known as the linear memory (the flat RAM past the address $00FFFF);
6) a 20 MHz CPU.

II. Command line arguments

The syntax is:

ELSA [options] source_file_name.ext [options]

The options are:

Switch	Function
/C	Case-insensitive labels: with this option the labels "ADR" and "adr" are identical.
/Dlabel=value	Assign "value" to the label named "label" and insert this label into the symbol table before the first assembly pass. Example: /DSTART=$2000
/L	Generate assembly listing during the second pass. The listing will appear on the screen, being formatted for 80-column displays. This switch has a priority over .LS and .LC directives possibly inserted into the source code (i.e. .LC will not be able to switch off the listing if it was enabled with -L).
/Mtarget	Define default target CPU. The available targets are: 6502, 65c02, 65sc02, 65c802 and 65c816. When no target is specified, 65C816 is assumed. How the targets are defined and what are the effects of selecting a particular target CPU, it is explained when the corresponding assembly directives are discussed. Example: /M65C802.
/Ofname.ext	Define the object file name. This switch has a priority over .OUT directives placed within the source code: then /O is specified, any .OUT will be ignored and a warning message will be printed on the screen. When the object file name is defined neither in the command line nor in the source code, the object code will not be saved anywhere. *Remark: note that it is not very safe to manually type both the source file name and the object file name each time a program needs to be assembled; better define the object file in your source using the .OUT directive, leaving the /O command line switch for the use inside BAT scripts and the like.*
/Q	Quiet assembly, i.e. suppress warnings.
/P	Warn about branches crossing a page boundary.
/R	After the second pass display the information on the usage of the labels.
/S	Dump entire symbol table to the screen. The following information is presented: label value, label type (Lt), program section (Ps), Warning record (Wr), label usage (Lu), name length (Nl), label name. *This information is of internal use only and may change meaning in the future.*
/U	Report all unreferenced internal addresses after the second pass.
/V	Report all unused labels after the second pass. This is reported by default in the final message as "n LABELS DEFINED (m NEVER USED)", adding the switch just causes the unreferenced labels to be explicitly listed. Unlike /U, this lists all unused labels regardless of their function, i.e. whether they are meaning addresses or values or whatever.

Instead of the "/" sign, the minus sign may be used, e.g. -M65C802 is perfectly valid.

III. General assembler syntax

As said above, ELSA is a clone of MAE. In the area of the syntax, MAE is in turn generally following the style of MAC/65, so that switching from the latter to the former makes no trouble. The MAE's oddity is that it only respects the first three characters of the name of a directive, so for example writing in the source .WORD or .WO makes no difference. ELSA keeps many of these quirks for (my) convenience, but the short forms are in fact explicit aliases for their longer equivalents.

IV. Labels

Label name

A label may be up to 240 characters long, which means that there is no practical size limit. The label's first character must be a letter, apart from that the decimal digits, the @ character, the dot (".") and the underscore character ("_") are allowed in the body of a label. The question mark, for the reason explained below, is only allowed as either the first or as the last character of a label (therefore such an expression as BOOT? = $09 is perfectly valid).

All label names beginning with double underscore character ("__") are reserved and should not be used in programs because of possible conflicts with pseudolabels (explained somewhat below) and labels declared by the assembler for internal purposes.

Label names are case-sensitive by default ("FOO" is different than "foo" etc.), if you want case insensitive searches, please specify /C in the command line.

General syntax

A label to be defined must start in the 0 (i.e. the leftmost) column of the text. Its name may be terminated with a colon (":"), this character, when found at the end of a label during its definition, is skipped.

Local labels, MAE-style

In the area of labels, the most notable feature of ELSA's predecessor, MAE, is the system of local labels marked with "?" character at the beginning. Such a label will serve as a local one in the area between two consecutive global labels. To reference such a local label, just prefix its name with the "?" character (e.g. LDA ?SIZE). When a reference to a local label is required from the outside of its global scope, the respective global label should be used followed by "?" and by the local label the reference is being made to (e.g. LDA IOCB?ICAX1,X). ELSA follows this system as a simple and elegant solution of the problem of label locality.

Any other label is a global label (unless stated otherwise).

Local namespaces

The directive .LOCAL *namespace* defines higher level of locality, not to be confused with the aforementioned system modelled after MAE (this may be used without using the .LOCAL keyword). All labels, no matter if "global" or "MAE-style local", when defined between two .LOCAL directives, belong to the local namespace defined by the first of them only. This allows strict separation of local namespaces from the main program and from each other, so that even the same include files, defining the same global labels, may be used multiple times in different parts of the program.

An obvious example is an init segment, which gets overwritten after use: within it you may use the same library procedures and system calls as within the rest of the program, but you do not want to reference accidentally from within the main program something which was only temporarily defined for the init segment.

When a reference between different namespaces is required, the label referenced should be preceded with the name of its namespace and a colon (":", e.g. JMP INIT0:START). The global namespace has no name, so when a reference to a global label is required from within a local namespace, the label being referenced should be preceded with a colon only (e.g. LDA :KBCODES).

References to a MAE-style local label defined within a local namespace from the outside of that namespace are not allowed.

Definition order

ELSA is a two-pass assembler, so it is best to define a label before its first use whenever possible: this is especially important in more complex arithmetic expressions, where all components must be defined during second assembly pass, or an error will occur. Labels for addresses may be defined after the reference, but when the address is on zero page, using it before definition will cause phasing error to occur during second pass. To avoid that, while referencing such a label, use the unary operator < (e.g. LDA <LABEL) to tell the assembler that the label to be defined will reference a zero-page location and an 8-bit address may be used.

V. Expressions

Like MAE, ELSA does not pay attention to arithmetic operator precedence, the expressions are evaluated straight from left to right, and there are no parentheses. Some day I will have to fix this, probably. The results coming from the integer evaluator are 32-bit unsigned integers. Whenever the result does not fit on 32 bits, it gets cut down to this size and a warning is generated.

The equal sign ("=") placed after a label means that the value of the following expression will be assigned to the label. Otherwise, when no equal sign follows, the label will be assigned the current value of the PC.

The asterisk ("*"), as in most other assemblers, means the current value of the PC. But, unlike in MAC/65, it is a read-only symbol and you cannot assign it a new

value; so the expression "*=*+value", commonly used in MAC/65 to reserve memory space of the "value" length, will not work - you have to use the .DS directive instead.

Value types

ELSA generally knows two types of values: addresses and values. The difference between them is mostly of internal significance only, and the conversions most of the time occur automatically. However, if an arithmetic operation tries to combine values with addresses or addresses with addresses in a suspicious way, the result will be of the value type and the assembler will generate a warning. The types can also be enforced by the programmer, whenever the automatic conversions do not yield satisfactory results: this may be done using the cast operators (explained further on).

VI. Constants

Prefix	Function
(none)	Decimal constant. Example: LDA 32000
$	Hexadecimal constant: LDA $7D00
%	Binary constant: LDA %0111110100000000
'	Character constant: LDA #'A
"..."	Character string or floating point constant: .BYTE "HELLO!"

Note that an ASCII constant consisting of a single character is marked by a single apostrophe situated in front of it. This also applies to directives such as .BYTE, thus

```
.BYTE 'H,'E,'L,'L,'O,'!
```

is a perfectly valid equivalent to the example shown in the table.

VII. Unary operators

Operators which can be applied to individual operands in expression:

Operator	Function
+	Do nothing.
-	Arithmetic negation: applies (XOR -1) + 1 (two's complement) to what follows.
!	Bitwise negation: applies XOR -1 (one's complement, „flip bits”) to what follows.

Operators which are applied to the result of entire expression after its evaluation:

Operator	Function
<	Extract the bits 0-7 of the given value: <$12345678 is $78.

>	Extract the bits 8-15 of the given value: >$12345678 is $56.
^	Extract the bits 16-23 of the given value: ^$12345678 is $34.
\	Extract the bits 24-32 of the given value: \$12345678 is $12.
($)	Cast the result to a value type. This also suppresses the warning on „fishy maths”.
(&)	Cast the result to an address type. The warning is likewise suppressed.

The casts are executed as the very last operations on the calculation's result, even after <, >, ^ and \.

Addressing modes:

Operator	Function
#	Force the immediate addressing mode (e.g. LDA #VALUE)
<	Force the zero-page addressing mode (e.g. LDA <VALUE)
|	Force the absolute (16-bit) addressing mode (e.g. LDA |VALUE)
!	Same as the | (e.g. LDA !VALUE).
>	Force the long absolute (24-bit) addressing mode (e.g. LDA >VALUE).

These latter ones will be applied first to arguments to mnemonics, then the assembler will proceed normally with the expression evaluation. So STA !0 (address 0 with forced 16-bit addressing) will produce $8D $00 $00, and STA !!0 will produce $8D $FF $FF (the first ! forces 16-bit addressing mode, the subsequent one negates the result of the argument evaluation).

VIII. Binary operators

Basic 32-bit integer arithmetics is what you expect:

Operator	Function
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo

The logical shifts (logical, because all the integers are unsigned) may be used as faster replacements for multiplication and division, where applicable:

Operator	Function
<<	Logical shift left. E.g. $00001234<<4 will produce $00012340
>>	Logical shift right. E.g. $FEDCAB98>>4 will produce $0FEDCAB9

There are slight differences between MAE and ELSA in the syntax of comparison

operators:

MAE	ELSA	Function
=	=	Equal
#	<>	Different
>	>	Greater
<	<	Lesser
(none)	>=	Greater or equal
(none)	<=	Lesser or equal

Attention should be paid to the fact that ELSA evaluates expressions from left to right. So, to avoid confusing effects in conditionals, it is best to do comparisons so that the right side of a comparator is a single constant or label. For example:

```
.IF FOO=BAR+1
```

will always(!) be TRUE; to make it work correctly write:

```
.IF BAR+1=FOO
```

Also, comparing to MAE, there are novelties in the logical operators:

MAE	ELSA	Function
&	&	binary AND
|	|	binary OR
^	^	binary XOR (EOR)
(none)	&&	logical AND
(none)	||	logical OR

If you supply addresses as both input numbers for most artithmetical or logical operations, the program will perform the required maths, but expect a warning in the process. Subtracting an address from another address, however, as being perfectly legal, is performed without complaints.

IX. Directives

The directives are keywords which are steering the process of assembling. In ELSA, as in MAC/65 and MAE, most of these keywords are preceded with a dot. It makes them more visible in the source code and also facilitates its parsing.

The directives must be located past the column 0 of your source file, i.e. there must be at least one space (or TAB) between them and the left margin. Only one directive is allowed per program line, unless stated otherwise.

Symbols used in the table below:

x - expression; w - expression, word value; b - expression, byte value; lb - label name. All these "expressions" must evaluate in the first assembly pass.

Target CPU control

Directive	Alias	Synopsis	Examples
.6502	.02	Set 6502 as the current target. Implies .RB. The target CPU is defined as a subset of 65C02, any instruction that does not belong to that subset will generate a warning.	.6502
.65C02	.c02	Set 65C02 as the current target. Implies .RB. The target CPU is defined as a subset of 65SC02, any instruction that does not belong to that subset will generate a warning.	.65C02
.65C802	.802	Set 65C802 as the current target. The target CPU is practically the 65C816, just the instructions related to 24-bit addressing (operational, but pretty much useless on 64k address space) will generate warnings.	.65C802
.65C816	.816 .65816	Set the 65C816 as the current target. This is the default, unless overridden in the command line or in the source code.	.65C816
.65SC02		Set 65SC02 (slightly modified 65C02 produced by Rockwell and WDC) as the current target. Implies .RB. The target CPU is defined as a subset of 65C802, any instruction that does not belong to that subset will generate a warning. Rockwell's BBR/BBS instructions and such (which are not continued in 65C802) are not supported.	.65SC02
.AB		Accumulator Byte: tell the assembler, that the current accumulator size is Byte. This directive has effect only if the target CPU is 65C802 or 65C816. For other targets the assembler will ignore the directive and generate a warning.	.AB
.AW		Accumulator Word: tell the assembler, that the current accumulator size is Word. This directive has effect only if the target CPU is 65C802 or 65C816. For other targets the assembler will ignore the directive and generate a warning.	.AW
.CPU b		If the current target is 65C802 or 65C816, tell the assembler if the current CPU mode selected is the emulation mode or the native mode. The parameter’s value of 8 means the emulation mode, and a value of 16 means the native mode. Other values will cause the	.CPU 16

		assembler to throw an error. .CPU 8 also implies .RB - and in this mode directives .AW, .IW and .RW will generate warnings and have no effect. If the current target CPU is not 65C802 or 65C816, .CPU 16 will generate a warning.	
.IB		Index registers Byte: tell the assembler, that the current X and Y register size is Byte. This directive has effect only if the target CPU is 65C802 or 65C816. For other targets the assembler will ignore the directive and generate a warning.	.IB
.IW		Index registers Word: tell the assembler, that the current X and Y register size is Word. This directive has effect only if the target CPU is 65C802 or 65C816. For other targets the assembler will ignore the directive and generate a warning.	.IW
.RB		Tell the assembler, that the current size of registers AXY is Byte. This directive has effect only if the target CPU is 65C802 or 65C816. For other targets the assembler will ignore the directive and generate a warning.	.RB
.RW		Tell the assembler, that the current size of registers AXY is Word. This directive has effect only if the target CPU is 65C802 or 65C816. For other targets the assembler will ignore the directive and generate a warning.	.RW

Conditionals and general assembly control

Directive	Alias	Synopsis	Examples
.ALIGN w		Align the current PC to the boundary specified by the argument. The argument's value has to be a power of two and be greater than a 0. A value of 1 is allowed, too, but it obviously does nothing.	.ALIGN $0100
.ELSE	.EL	This inverts the result of the expression evaluation made by the .IF directive.	(see .IF)
.END		Ends the assembling and closes the object code file, if any was opened.	.END
.ENDIF	***	This ends the conditional block started with .IF.	(see .IF)
.ERROR		Just like .PRINT, but after printing out the required text it also aborts the assembling with an error message. Obviously it makes sense within a conditional block only (.IF /	.ERROR "LM=",LM

		.ENDIF).	
.IF x		The beginning of the conditional block. If "x" is evaluated as true, the lines immediately following the .IF directive will be interpreted, or ignored otherwise. The conditional blocks can be nested up to 256 levels deep. When this limit is exceeded, the assembler will generate an error message.	.IF __M6502__=1 .PRINT "NMOS" .ELSE .PRINT "CMOS" .ENDIF .IF AB&&BC ...
.IFDEF lb		Returns TRUE if the label "lb" is defined, i.e. already present in the symbol table.	.IFDEF USE_CIO .INCLUDE CIO.S .ENDIF
.IFNDEF lb		As above, just returns FALSE when the label 'x' is defined.	.IFNDEF RTCLOCK RTCLOCK=18 .ENDIF
.OPT		Specify additional assembly options: * F- use this when including binary files with .BIN from a file system which does not provide reliable information on the length of files (such as AtariDOS file system). F+ is the default. * H- suppress (or enable with H+) writing headers to the object code. H+ is the default. * P+ - enable warnings on cross-page branches. P- is the default, unless /P was specified in the command line. If it was, P+ or P- used in the code have no effect. * W- disable warnings. W+ is the default, however the command line switch /Q has a priority here and with it the .OPT W+ will not enable warnings anyway.	.OPT H-,F- .OPT H+,F+
.PRINT	.PR	Prints the given text during the assembling. ASCII strings must be included within double quotation marks, multiple arguments must be separated with commas. When nothing is given, .PRINT will just output an EOL character.	.PRINT "PC:",*

Repetitions

Directive	Alias	Synopsis	Examples
.ENDR		Marks the end of the block started with .REPT.	(see .REPT)
.REPT w		Marks the beginning of the block of lines in your source file, which have to be repeated "w" times during assembling. The end of that block should be marked with .ENDR. Within the block, the pseudo-label __REPT__ contains the number of the current iteration,	LDA MATH .REPT 8 ASL ROL MATH+1 BCS SKIP# INC NULS

		and the operator # when appended to a label, makes it unique for each iteration. When "w" is zero, the pair REPT/ENDR will do nothing, and the assembler will generate a warning. The .REPT blocks cannot be nested.	SKIP# .ENDR

Fixed data definition control

Directive	Alias	Synopsis	Examples
.BYTE	.BY	Inject the given byte values to the output file. The consecutive "bytes" separated with commas can be: decimal numbers, hexadecimal numbers preceded with the $ character, single ASCII characters preceded with the apostrophe, labels, arithmetic expressions, or text strings included in the double quotation marks. When the first numeric value is preceded with the + or - sign, this value will be added to or subtracted from, respectively, the rest of the values generated by this directive.	.BYTE 0,$FF,'A,"Hey" .BYTE <VAL,>VAL .BYTE SIZE*2+1 .BYTE +$80,"HELLO" .BYTE -$20,"caps"
.CBYTE	.CB	As .BYTE, except that the last byte generated by the single .CBYTE directive will be "inverted" (i.e. EORed with $80).	.CBYTE "LOAD" .CBYTE +$20,"CAPS"
.DBYTE		As .WORD, but with the inverted order of the bytes (i.e. MSB first).	.DBYTE $07FF,13
.DC w b		Define Constant-filled block. The consecutive 16-bit "w" number of bytes will be filled with the 8-bit value of "b". A value of 0 for "w" will generate a warning.	.DC 345 $FF
.FLOAT	.FL	Store the arguments, separated by commas, in the Floating Point 6-byte BCD format for use with the OS ROM's Floating Point package. This directive accepts two types of arguments: FP constants, which are just converted to the BCD, and integer expressions, which are evaluated normally, then the result is converted to the BCD format. An FP constant must be included in double quotation marks; when they are missing, this means that the argument is an integer expression to be evaluated first. The results coming from the integer evaluator are interpreted as signed values (range -2147483648 < 0 < 2147483647), so e.g. .FLOAT -2+1 produces correct "-1" in BCD, and not what the integer evaluator would bring normally, i.e. 4294967295.	.FLOAT "3.14","1E+10" .FLOAT 256*3,-2+1
.HEX	.HE	Generate a series of hexadecimal numbers. It	.HEX AE 19 00 44 FF

		is similar to .BYTE, but in the particular case of hex number may be more handy, because does not stipulate them to be preceded with the $ sign and the separator is space.	
.LONG	.LO	Stores long, 24-bit words in the object code, in the usual order of bytes, i.e. LSB first.	.LONG $F1234,99999
.QBYTE		Store 32-bit words in big-endian order (MSB first).	.QBYTE $12345678
.QUAD	.DWORD	Store 32-bit words in little-endian order (LSB first).	.QUAD $12345678
.SBYTE	.SB	Like .BYTE, but understands the given bytes as ASCII values, and converts them to Atari screen codes before storing in the object code.	.SBYTE "TIME:" .SBYTE +$60,"NAME"
.TBYTE		Stores 24-bit long words in the memory in the reverse order of bytes, i.e. MSB first. In other words, it is to .LONG like .DBYTE to .WORD.	.TBYTE $ABCDEF
.WORD	.WO	Stores 16-bit words in the object code, in the usual order of bytes (LSB first).	.WORD $E477,20133

Code generation control

Directive	Alias	Synopsis	Examples

[
...
]
		Define a block of code to be used with the conditional pseudo-instructions Rcc and Scc. If the block contains no code or data directives, the assembler will generate a warning.	LDY #$00
			[
			  LDA $2000,Y
			  STA $3000,Y
			  INY
			]
			RNE

.CODE		Switch the PC to the code section (or: switch back, because the code section is the default one). In practice, it marks an end of the block which was started with .ZP, .DATA or .BSS. See below the section on *Declaring zero-page locations*.	.ZP XY .DS 2 .CODE LDA XY
.DATA		Switch PC to the DATA section. This code allows you to declare initialized static variables, which later will be accumulated in one continuous memory block. See below the section on *Sections DATA and BSS*.	.DATA FOO .BYTE 1,2,3,4 .CODE
.INIT w		Specify a value for the INITAD vector ($02E2). The corresponding init segment will be generated immediately.	.INIT SETUP
.MC w		Move the following code to a different address than the one specified by the .ORG directive. The difference is that the .ORG	.ORG $2000 .MC $0600

		address will be used as the program counter to calculate addresses, while the .MC address will be used to create AtariDOS-style binary headers. In the example shown on the right, the code will be assembled to run at $2000, but the binary loader will load it at $0600 – the code block must be copied over before being started. Thus every use of .MC is likely to generate binary headers (unless they are disabled with .OPT H-). „Moving" to current address (.MC *) disables the effect of this keyword. *Remark: this keyword is only allowed inside the CODE section.*	
.ORG w	.OR	Specifies the address where (a portion of) the object code has to be stored in the memory. The assembler is able to generate up to 1024 separate segments within one binary file.	.ORG $2000
.RUN w		Specify a value for the RUNAD vector ($02E0). If "w" is non-zero, the RUN segment will be generated and appended at the end of the object file.	.RUN START

Memory allocation control

Directive	Alias	Synopsis	Examples
.BSS [base]		Switch the PC to the BSS section. This keyword allows you to declare uninitialized static variables anywhere in your code. The assembler will then accumulate them automatically in one continuous memory block (which may be assigned, for example, to a 16k RAM bank or to a different 64k memory segment). The usage of the keyword is quite similar to .ZP, with some minor differences. See below the section on *Sections DATA and BSS*.	.BSS XY .DS 2 .CODE
.DS w		Declare Storage. In all sections besides DATA, it reserves the "w" number of bytes as uninitialized data array (as small as 1 byte – 0 will generate a warning). In other words, it does not generate code or data, it just adds the given number to the current PC during assembling, thus making an empty "gap" in the memory. In DATA sections it generates the specified number of zeros, being an equivalent to .DC w 0.	.DS 32
.RS w		Reserve space for an individual variable within a structure. The space will be of "w" bytes (0 is allowed, but will generate a	.RS 2

		warning). See below the section on *Defining structures in the memory and on the stack*.	
.RSSET w		Beginning of a structure. The "w" is the offset of the structure's first element.	.RSSET 0
.ZP [b]		Switch the PC to the Zero Page section. This keyword allows to declare zero page variables anywhere in your source code. See below the section on *Declaring zero-page locations*.	

Label control

Directive	Alias	Synopsis	Examples
.LOCAL lb		Define new local namespace named "lb". This automatically ends the current local namespace and switches to the new one. When only terminating the local namespace and switching to the global one is needed, the directive should get no parameters. The "lb" label size is limited to 64 characters.	.LOCAL INIT0 INIT ... .ORG $02E2 .WORD INIT .LOCAL
.SET lb = x		Change the value of the label "lb" which was already defined and has a value assigned. *The form with the dot (backwards incompatible with MAE, but preferred by the vast majority of the millions of users) is official as of 0.93.*	.SET zpc = $0100

Input/output control

Directive	Alias	Synopsis	Examples
.BIN	.BI	Binary Include. Unlike in MAE, the contents of the file is not interpreted in any way, it is just verbatim inserted into the object code. When the file about to be included is located on a file system which does not provide reliable information on file's length (such as AtariDOS/MyDOS file system), use .OPT F- before calling this directive (and .OPT F+ afterwards). *This causes the file being included to be read on both passes.*	.BIN PIC.BMP
.CACHE		Tells the assembler to cache (beginning at the next line) all the source lines in the memory during the first pass, then use the cached data during the second pass instead of re-reading all the source files from the disk. This may greatly speed the assembly up, if the memory is much faster than disk storage, but the system has to have enough RAM available to hold the symbol table and the cached source code (ELSA itself needs over 200 KB to be	.CACHE

		assembled this way). The binary files included with .BIN are not cached.	
.INCLUDE	.IN	Includes another source file. These directives can be nested (i.e. the included file may contain another .INCLUDE directives), just remember that the stack space is not unlimited.	.INCLUDE MATH.S
.OUT	.OU	Specifies the name of the object code. This directive will get ignored, if the output file was specified in the command line. When this file name is not specified either way, the object code will not be stored anywhere.	.OUT TEST.COM

Listing control

Directive	Alias	Synopsis	Examples
.LC		Switch off ("clear") the assembly listing.	(see .LS)
.LL		List just the following line.	.LL STA 24*OFFSET+L,X
.LS		Switch on ("set") the assembly listing.	.LS .ORG $0600 START JMP $E477 .END .LC

X. Pseudo-labels

Pseudo-labels are keywords with a value, which can be used in arithmetic expressions just like normal labels. To differentiate a named pseudo-label from other labels, ELSA marks them by putting two underscore characters (__) before and after the label's name (examples below).

All label names beginning with double underscore character ("__") are reserved and should not be used in programs because of possible conflicts with labels declared by the assembler for internal purposes.

The pseudo-labels usually carry numeric values signalizing currently selected assembling settings, just as register sizes and such things. Therefore they are particularly useful in conditional blocks started with .IF.

Label's name	Synopsis
*	Current value of the Program Counter within the program being compiled. Note that the assembler maintains more than one Program Counter, e.g. there are separate Program Counters for the .ZP segment and for the .CODE segment.
__ASIZE__	Current Accumulator size in bytes, as selected by the directives: .AB, .AW, .RB and .RW.

__BSS__	Current PC value within the BSS section. Note that this is always a value relative to the BSS base.
__CPU__	Current CPU mode: a value of 8 means the 65C802/65C816 emulation mode, a value of 16 means the 65C802/65C816 native mode. If the target CPU is anything below 65C802, the value returned is always 8.
__DATA__	Current PC value within the DATA section. As in BSS, this is always a value relative to the DATA base.
__DATE__	Current date, day, month, year, stored as a 24-bit word. E.g. 13 February 2020 is represented as $14020d in the usual little-endian byte order: $0d, $02, $14. *If the computer is not running SpartaDOS X, for this function to work you have to install a SpartaDOS-compatible "Z:" device in the system*.
__ISIZE__	Current size of X and Y register in bytes, as selected by the respective directives: .IB, .IW, .RB and .RW.
__M6502__	This has value of 1, if the currently selected target CPU is 6502, and 0 otherwise.
__M65C02__	1, if the currently selected target CPU is 65C02, and 0 otherwise.
__M65SC02__	1, if the currently selected target CPU is 65SC02, and 0 otherwise.
__M65C802__	1, if the currently selected target CPU is 65C802, and 0 otherwise. Alias: __M65802__
__M65C816__	1, if the currently selected target CPU is 65C816, and 0 otherwise. Alias: __M65816__
__REPT__	Current iteration within the .REPT/.ENDR block.
__RS__	Current offset of the structure defined by the directives .RSSET and .RS.
__RSSIZE__	Current size of the structure defined by the directives .RSSET and .RS.
__TIME__	Current time of day, stored as a 24-bit word. 24-hour clock is used, so e.g. 19:11:05 will be represented as $050b13, i.e. $13, $0b, $05 in the little endian byte order. *If the computer is not running SpartaDOS X, for this function to work you have to install a SpartaDOS-compatible "Z:" device in the system*.
__ZP__	Current PC within the ZP section, i.e. the zero page defined by the .ZP directive.

XI. Pseudo-instructions

A pseudo-instruction (otherwise known as a macro-instruction) is a kind of a hard-coded macro: the assembler presents it under a single mnemonic, but during the assembling this mnemonic is expanded into a series of actual CPU instructions.

The general rules for a pseudo-instruction in ELSA are these:

1) a pseudo-instruction has to follow the syntax of a real instruction, i.e. only the otherwise existing addressing modes are allowed;

2) a pseudo-instruction must not generate confusing side effects, e.g. so that one which claims to modify the memory also modifies the accumulator and flags, without a warning.

ELSA implements these:

Syntax	Synopsis	Expands to

ADD ...	Add without carry. The same addressing modes are available as for the ADC, this pseudo-instruction is just 1 byte longer and takes 2 cycles more. Counterpart: SUB.	CLC ADC ...
ASR	Arithmetical Shift Right. As LSR, but the highest order bit (= sign bit) of the Accumulator is preserved. Implied addressing only. 3 and 4 cycles for 8-bit Accumulator, or 4 bytes and 5 cycles for a 16-bit one.	CMP #$80 ROR or: CMP #$8000 ROR
B2H	Binary To Hex: convert the lowest nibble of the accumulator into the corresponding hex digit, and store the digit in the lowest byte of the Accumulator. If other nibbles of the Accumulator (8-bit or 16-bit in respective modes) contain anything but zeros, this instruction may yield undefined results. 6 bytes, 8 cycles for 8-bit Accumulator, 8 and 10 respectively for 16-bit Acc.	CMP #$0A SED ADC #$30 CLD or: CMP #$000A SED ADC #$0030 CLD
BSL *address*	Branch to Subroutine Long: a position-independent equivalent of JSR, with the range of a 32k in either direction. 6 bytes, 10 clock cycles.	PER *ret*-1 BRL *address* *ret*
BSR *address*	As above, just the branch is short: the range is 128 up or down. 5 bytes, 9 clock cycles (or 10, when the branch has to cross a page boundary).	PER *ret*-1 BRA *address* *ret*
DEW *address* DEW *address*,X	Decrement Word. The Accumulator gets clobbered in the process and NZ flags are left inconsistent, so the assembler will throw warnings because of that. 8 to 11 bytes, zero page min. 11, max. 15 cycles (17 when index is crossing page boundary), absolute min. 13, max. 18 (20 when index is crossing page boundary). When the target CPU is 65C802 or 65C816 and the currently selected Accumulator and Memory size is 16 bits (M=0), DEW is compiled to a single DEC instruction (7 cycles for the zp, 8 for the absolute addressing mode), which leaves the NZ flags both valid afterwards.	LDA *address*... BNE *skip* DEC *address*+1... *skip* DEC *address*... or: DEC *address*...
DWA *address* DWA *address*,X	Decrement Word using Accumulator. Like DEW, but makes explicit the intermediate use of the Accumulator (hence no warning about it being clobbered). Still, the Accumulator is in undefined state afterwards and so are the NZ flags, which only reflect the state of the LSB of the word being decremented. 8 to 11 bytes, zero page min. 11, max. 15 cycles (17 when index is crossing page boundary), absolute min. 13, max. 18 (20 when index is crossing page boundary). When the target CPU is 65C802 or 65C816 and the currently selected Accumulator and Memory size is 16 bits (M=0), DWA is compiled to the	LDA *address*... BNE *skip* DEC *address*+1... *skip* DEC *address*... or: LDA *address*... DEC STA *address*...

	LDA/DEC/STA series of instructions (10 cycles for the zp, 12 for the absolute addressing mode), which leaves the NZ flags both valid afterwards, and the result of the decrementation in the Accumulator.	
Ecc	Exit (= return from) subroutine if the condition "cc" is met. 3 bytes, 8 cycles taken, 3 cycles not taken (or 4, if the pseudo-instruction components cross a page boundary). This pseudo-instruction is convenient when things are about terminating a subroutine prematurely, but one should remember that using a branch to nearest RTS instead of Ecc may produce better code (i.e. saves one byte, although usually takes one or two cycles more).	Bcc *skip* RTS *skip*
ECC	Exit (= return from) subroutine if Carry Clear.	BCS *skip* RTS *skip*
ECS	Exit subroutine if Carry Set.	BCC *skip* RTS *skip*
EEQ	Exit subroutine if EQual.	BNE *skip* RTS *skip*
EGE	Exit subroutine if Greater or Equal. Same as ECS.	BCC *skip* RTS *skip*
ELT	Exit subroutine if Lesser Than. Same as ECC.	BCS *skip* RTS *skip*
EMI	Exit subroutine if MInus.	BPL *skip* RTS *skip*
ENE	Exit subroutine if Not Equal.	BEQ *skip* RTS *skip*
EPL	Exit subroutine if PLus.	BMI *skip* RTS *skip*
EVC	Exit subroutine if V flag clear.	BVS *skip* RTS *skip*
EVS	Exit subroutine if V flag set.	BVC *skip* RTS *skip*
INW *address* INW *address*,X	INcrement Word. Like an INC *address*, but increments a word located at *address* and *address*+1. When the address is on the zero page, occupies 6 bytes and takes 8 to 12 cycles; when outside the zero	INC *address*... BNE *skip* INC *address*...+1 *skip*

	page, 8 bytes and 9 to 14 cycles. The N flag is not in a consistent state afterwards, Z is. When the target CPU is 65C802 or 65C816 and the currently selected Accumulator and Memory size is 16 bits (M=0), INW is compiled to a single INC instruction, and then the NZ flags are both valid afterwards.	or: INC *address...*
Jcc *address*	Jump if the condition "cc" is met. It is an absolute version of conditional branches Bcc with identical meaning, but the jump range of 64k instead of 256 bytes. 5 bytes, 5 cycles taken, 3 cycles not taken (or 4, when the instruction components cross a page boundary).	Bcc *skip* JMP *address* *skip*
JCC *address*	Jump if Carry Clear.	BCS *skip* JMP *address* *skip*
JCS *address*	Jump if Carry Set. As above, with the opposite condition.	BCC *skip* JMP *address* *skip*
JEQ	Jump if EQual.	BNE *skip* JMP *address* *skip*
JGE	Jump if Greater or Equal. Same as JCS.	BCC *skip* JMP *address* *skip*
JLT	Jump if Lesser Than. Same as JCC.	BCS *skip* JMP *address* *skip*
JMI	Jump if MInus.	BPL *skip* JMP *address* *skip*
JNE	Jump if Not Equal.	BEQ *skip* JMP *address* *skip*
JPL	Jump if PLus.	BMI *skip* JMP *address* *skip*
JSL [abs] JSR [abs]	Jump to Subroutine Long, indirect. Like JSR (abs), just using a long pointer (located at address *abs* in segment 0), therefore requiring RTL to return. Only available for 65C802 and 65C816 targets. Note that the pointer *abs* must be located in segment 0. 7 bytes, 15 cycles.	PHK PEA *ret*-1 JML [*address*] *ret*
JSR (abs)	Jump to SubRoutine, indirect. Like JMP (abs), just pushing the return address onto the stack. Only available for 65C802 and 65C816. Note that the pointer *abs* must be located in segment 0. 6 bytes, 11	PEA *ret*-1 JMP (*address*) *ret*

	cycles.	
JVC	Jump if V flag Clear.	BVS *skip* JMP *address* *skip*
JVS	Jump if V flag Set.	BVC *skip* JMP *address* *skip*
PHR	Push Registers. Counterpart: PLR. The size and execution time depends on the target CPU and current circumstances: 6502: 5 bytes and 13 cycles; 65C02: 3 bytes and 9 cycles 65C802/816: 3 bytes and * 9 cycles for all registers byte-sized; * 10 cycles for word-sized accumulator; * 11 cycles for word-sized index registers; * 12 cycles for all registers word-sized. *Note that on 6502 there is an unpleasant side effect: the Accumulator content gets lost - after the PHR's execution A contains a copy of the Y register. The assembler will therefore generate a warning in this case.*	PHA PHX PHY or (for 6502 target): PHA TXA PHA TYA PHA
PLR	Pull Registers. The reverse of the PHR. The size and execution time depends on the target CPU and current circumstances: 6502: 5 bytes and 16 cycles; 65C02: 3 bytes and 12 cycles 65C802/816: 3 bytes and * 12 cycles for all registers byte-sized; * 13 cycles for word-sized accumulator; * 14 cycles for word-sized index registers; * 15 cycles for all registers word-sized.	PLY PLX PLA or (for 6502 target): PLA TAY PLA TAX PLA
Rcc	Repeat previous instructions if condition "cc" is met. This pseudo-instruction comes in two flavours. In its simple form it just follows one instruction which is to be repeated, in this manner: LDA VCOUNT RNE In its more complex form, an entire block of instructions can be repeated. The block should be defined using the directives [and], in this manner: LDY #$00 [LDA $2000,Y STA $3000,Y INY] RNE When the branch is in 8-bit signed range, this pseudo-instruction is compiled as a Bcc, or as a Jcc	*loop ...* Bcc *loop* or: *loop ...* Jcc *loop*

	otherwise. Therefore the resulting object code may accordingly vary in code size and execution time.	
RCC	Repeat instructions if Carry Clear.	*loop ...* BCC *loop* or: *loop ...* BCS *skip* JMP *loop* *skip*
RCS	Repeat instructions if Carry Set.	*loop ...* BCS *loop* or: *loop ...* BCC *skip* JMP *loop* *skip*
REQ	Repeat instructions if EQual.	*loop ...* BEQ *loop* or: *loop ...* BNE *skip* JMP *loop* *skip*
RGE	Repeat instructions if Greater or Equal. Same as RCS.	*loop ...* BCS *loop* or: *loop ...* BCC *skip* JMP *loop* *skip*
RLA	Rotate bits Left in Accumulator. Counterpart: RRA. Unlike in ROL, the highest bit is copied not only to the C flag, but also to the bit 0. Timings are identical as in ASR.	CMP #$80 ROL or: CMP #$8000 ROL
RLT	Repeat instructions if Lesser Than. Same as RCC.	*loop ...* BCC *loop* or: *loop ...* BCS *skip* JMP *loop* *skip*
RMI	Repeat instructions if MInus.	*loop ...* BMI *loop* or: *loop ...* BPL *skip* JMP *loop* *skip*

RNE	Repeat instructions if Not Equal.	*loop ...* BNE *loop* or: *loop ...* BEQ *skip* JMP *loop* *skip*
RPL	Repeat instructions if PLus.	*loop ...* BPL *loop* or: *loop ...* BMI *skip* JMP *loop* *skip*
RRA	Rotate bits Right in Accumulator. Counterpart: RLA. Unlike in ROR, bit 0 is copied straight into the highest bit. 5 to 6 bytes, 5 to 7 cycles.	LSR BCC *skip* ORA #$80 *skip* or: LSR BCC *skip* ORA #$8000 *skip*
RVC	Repeat instructions if V flag clear.	*loop ...* BVC *loop* or: *loop ...* BVS *skip* JMP *loop* *skip*
RVS	Repeat instructions if V flag set.	*loop ...* BVS *loop* or: *loop ...* BVC *skip* JMP *loop* *skip*

Scc	Skip following instruction if condition "cc" is met. This pseudo-instruction precedes the instruction which is to be skipped, in this manner:
	  INC ADR
	  SNE
	  INC ADR+1
	In the more complex form the [and] may be used to define the block to skip:
	  LDA $2000
	  SEQ
	  [
	  LDY #$00
	  [
		Bcc *skip*
		 ...
		skip

	LDA $2000,Y STA $3000,Y INY] RNE] *Remark: in the current implementation no global labels may be defined within the scope of this pseudo-instruction. For example, the following:* *SNE* *RESET JMP $E477* *...* *will cause the assembler to throw an error. Also, the Scc pseudo-instruction branch range is 127 bytes only. When the defined block exceeds this range, the assembler will throw an error.*	
SCC	Skip instruction if Carry Clear.	BCC *skip* *...* *skip*
SCS	Skip instruction if Carry Set.	BCS *skip* *...* *skip*
SEQ	Skip instruction if EQual.	BEQ *skip* *...* *skip*
SGE	Skip instruction if Greater or Equal. Same as SCS.	BCS *skip* *...* *skip*
SLT	Skip instruction if Lesser Than. Same as SCC.	BCC *skip* *...* *skip*
SMI	Skip instruction if MInus.	BMI *skip* *...* *skip*
SNE	Skip instruction if Not Equal.	BNE *skip* *...* *skip*
SPL	Skip instruction if PLus.	BPL *skip* *...* *skip*
SVC	Skip instruction if V flag Clear.	BVC *skip* *...* *skip*
SVS	Skip instruction if V flag Set.	BVS *skip* *...* *skip*
SUB	Subtract without carry. The same addressing modes	SEC

	are available as for the SBC, this pseudo-instruction is just 1 byte longer and takes 2 cycles more. Counterpart: ADD.	SBC ...

XII. Instruction aliases

An alias is just an alternative mnemonic for an instruction. ELSA implements a handful of these, mostly following the CPU producer's advice.

Syntax	Synopsis	Equivalent to
BGE *address*	Branch if Greater or Equal.	BCS *address*
BLT *address*	Branch if Lesser Than.	BCC *address*
CLR *address* CLR *address,*X	Clear the specified memory location.	STZ *address* STZ *address,*X
CPA ...	Compare with the Accumulator.	CMP ...
DEA	Decrement Accumulator.	DEC
HLT	Halt the processor.	STP
INA	Increment Accumulator.	INC
LSL ...	Logical Shift Left	ASL ...
PEI (*address*)	Push Effective address, Indirect (move word from ZP to stack)	PEA (*address*)
PER *address*	Push Effective address, Relative	PEA *address*
SWA	SWap Accumulator halves.	XBA
TAD	Transfer Accumulator to Direct page register.	TCD
TAS	Transfer Accumulator to Stack pointer.	TCS
TDA	Transfer Direct page register to Accumulator.	TDC
TSA	Transfer Stack pointer to Accumulator.	TSC

XIII. Alternative syntax in some instructions

Some instructions have been given alternative syntax as if they had additional addressing modes, which they obviously do not have; instead, it is just the way ELSA is allowing the programmer either to omit mandatory argument(s), when the value of the argument(s) is implied, or to control whether to add the argument or not for special purposes.

So, first of all, you can omit the arguments for MVN/MVP, if both arguments are to be zeros:

Basic syntax	Alternative syntax
MVN 0,0	MVN
MVP 0,0	MVP

This does not change the code being generated, i.e. the mnemonic MVN without its arguments specified will generate the same code as MVN 0,0.

Another case are the instructions BRK and WDM. Both are in fact two-byte, but the basic syntax does not allow to specify the immediate argument. So ELSA allows this:

Basic syntax	Alternative syntax
BRK	BRK #$xx
WDM	WDM #$xx

This *does* change the code generated. For example, BRK alone will cause one byte (of value of $00) to be generated to the object file, but f.e. BRK #$80 will generate two bytes: $00 $80.

The next case is BIT absolute:

Basic syntax	Alternative syntax
BIT abs	BIT

The alternative syntax will cause just one byte ($2C) to be generated to the object code. As the instruction in fact occupies 3 bytes, this may be used to mask out any following two-byte instruction, effectively skipping it. This effect was traditionally accomplished by putting .BYTE $2C into the instruction stream, ELSA just makes it more explicit.

Basic syntax	Alternative syntax
BCC label	BCC
BCS label	BCS
BEQ label	BEQ
BNE label	BNE
BPL label	BPL
BMI label	BMI
BVC label	BVC
BVS label	BVS

The purpose of these is the similar as above, i.e. masking out any following one-byte instruction. To accomplish that you just need to recognize the current condition, then use the branch for the exactly opposite condition to use it to skip something. For example:

```
CLEAR CLC
      BCS
SET   SEC
```

```
        ROR FLAG
```

Calling the location marked with the label CLEAR will clear the C flag, then the following BCS branch will get ignored together with the SEC instruction which will get interpreted as its argument - and this effectively makes it skipped.

The BIT zp instruction is traditionally used for this purpose (by inserting .BYTE $24 into the instruction stream), but using a branch takes one cycle less and, unlike BIT, does not generate spare memory accesses.

XIV. Divergences from the WDC-recommended syntax

The main divergence from the syntax and mnemonic names, which are recommended by the WDC, concerns the PEA instruction. The WDC syntax is this:

PEA $xxxx – PEA absolute
PEI ($xx) – PEA direct page indirect
PER $xxxx – PEA relative

But this "PEA absolute" simply pushes its 16-bit argument value onto the stack, so you could think that naming it (the argument) "absolute effective address", especially in a machine where effective absolute addresses are 24-bit, is quite an overstatement. Sure, we write JMP $xxxx, and speak of the instruction as being in absolute addressing mode, but JMP actually *uses* its argument as *an address* to change the current location of the PC within the code. If we were thinking of JMP as of a 16-bit move (which it technically is), we could symbolically write it down as MOVE #$xxxx,PC – and yes, in *this* context, *with* the hash.

So, ELSA (and some other assemblers) are treating the first instance of PEA as being in immediate mode. Therefore the syntax is as follows:

ELSA syntax	WDC syntax
PEA #$xxxx	PEA $xxxx
PEA ($xx)	PEI ($xx)
PEA $xxxx	PER $xxxx

As hinted in the previous section, you can still use PEI ($xx) and PER $xxxx besides PEA ($xx) and PEA $xxxx, respectively.

XV. Declaring zero-page locations

Zero-page variables may be declared the traditional way, i.e. assigning labels fixed values, like this:

```
POINTER = $80
TEMP = $82
```

or, more conveniently, using the .ORG directive to set the PC at a zero-page address combined with the .DS directive reserving space, like this:

```
     .ORG $80
POINTER  .DS 2
TEMP     .DS 1
```

Both ways, however, are troublesome when writing or maintaining a larger program which is distributed among several source files (or „modules"); it is best to have the variables declared in the very module which uses them, but the former way makes it difficult to track among several files which locations are occupied and which are not, and the latter one is little improvement: you can easily allocate blocks of variables, but still there may be conflicts, difficult to track down and solve, between the blocks declared by different modules of the program.

So, ELSA provides a mechanism which allows to automatically allocate zero-page variables so that they may be freely declared globally anywhere in the program, and are sequentially allocated at assembly time so that no conflicts are possible and you do not need to trouble yourself with assigning actual addresses.

To accomplish this, ELSA provides two keywords:

.ZP – which begins the zero-page declaration block, and
.CODE – which ends the block.

Between these you declare your variables using the .DS directive, for example:

```
     .ZP $80
POINTER  .DS 2
TEMP     .DS 1
     .CODE
```

The number to the right to the .ZP directive is the address of the zero-page variables to be declared for the entire program. You need to specify this address in the first .ZP directive, because otherwise the assembler will assume address $00 (this is the default) and assign your variable to that location – and this is rarely desired on Atari. But for all following .ZP directives this number should be omitted: the subsequent .ZP directive will then pickup the zero-page address where the last one left it and perform the sequential allocation as desired.

Following the example above, the next declaration block may look like this:

```
     .ZP
CX   .DS 1
CY   .DS 1
CZ   .DS 1
     .CODE
```

These two blocks declare the following locations: POINTER = $80, TEMP = $82,

CX = $83, CY = $84, CZ = $85. Any third declaration block will then begin allocation at the address $86 and so on. Of course, as much actual code or data as you want may intervene between these blocks, so that zero-page variables can easily be declared not only by the modules they belong to, but they also can be just declated straight before the actual procedures which use them.

Technicalia: all this works so that the .ZP maintains own program counter. The numeric parameter next to .ZP sets this counter to a value (which is $00 by default). Each .DS directive increases the counter, and .CODE switches back to the „main" program counter, while the .ZP counter remains intact. Any next .ZP directive (without any additional parameters) will switch to the .ZP counter and use its current value as the starting point for the allocation. The counter is 32-bit, each time it spans a 256-byte boundary the assembler generates a warning.

XVI. Sections DATA and BSS

The keywords .DATA and .BSS allow your program to contain separate sections which will accumulate initialized (.DATA) or uninitialized (.BSS) variables. This way you will be able to easily split your program into two memory blocks: code on the other side, and data on the other side. This in turn will allow to prepare programs which can store its code and data in separate address spaces (such as separate 64k segments of memory).

1) Even if your program will run in unified address space (like all 6502 programs do), the BSS section can still be useful. In small programs (fitting entirely in one source module) it is usually not necessary to define a separate section for that, but in larger assemblies it may be advantageous to accumulate uninitialized variables in one memory block. Particularly all sorts of source code libraries may benefit from that, because these usually want to declare static variables in their own source files, which in turn, when they get included, makes the object code more fragmented – and this increases the size of the program and the necessary loading overhead.

2) The basic usage of the .BSS keyword is generally similar to the .ZP: switch to the BSS section using .BSS, declare space inside using .DS, switch back using .CODE. For example:

```
      .BSS
CX    .DS 1
CY    .DS 1
CZ    .DS 1
      .CODE
```

As in the ZP section, no code or initialized data are allowed within the BSS section. One functional difference is that the BSS section is located in the main memory, so you have to use the absolute (or absolute long) addressing mode to make references to it. But otherwise everything works as in ZP sections as long as your BSS is located in a dedicated 64k segment, forming an address space truly separate from code's.

3) The DATA sections works similarly to the BSS section, except that it contains

actual data (no code or offsets are allowed). The DATA section accumulates the data being generated by keywords such as .BYTE, .WORD etc. then stores them all in one large binary block, which will be appended at the end of the object code. For example:

```
      .DATA
CX    .BYTE 0
CY    .BYTE 0
CZ    .BYTE 0
      .CODE
```

By default it is assumed that both DATA and BSS sections belong to separate address spaces. Therefore both of them are addressed from virtual address $000000 onwards, which means that by default they overlap. How to prevent this, will be explained later on.

4) Things get complicated when your DATA section or BSS section or both have to share the address space with the code of the program. In this case the DATA/BSS sections must form their virtual address spaces within the code segment. And even if your program wants to separate code from data and put them into different address spaces, you still may want to the BSS section to share the same address space with the DATA section.

In all these cases, to avoid overlapping this area with other memory areas, you have to tell the assembler where will be the physical beginnings of the sections, or in other words, where is the DATA base or BSS base (or both).

By default, the DATA and BSS base are both zero, and, as said above, if you intend it to be so (because the DATA/BSS will be physically allocated in different 64k segment(s)), no additional action is necessary: the assembler will be allocating your variables from virtual address $000000 onwards, and it is up to the code (or a loader) to take that into account. *Do not worry: even if a DATA or BSS variable is declared at address lesser than $0100, the assembler will never assume that a zero page addressing mode should be used in the reference.*

But if you want to specify an explicit address for the base, state it as an argument to the .DATA or .BSS keyword. For example:

```
      .BSS $8000
```

When you put such a statement in your code, the assembler will allocate your BSS variables starting from the specified address. The .DATA keyword works identically.

Remark: when the argument is specified, the keyword only declares the section's base, but does not switch to the section, so it is not necessary to use .CODE afterwards.

If your program has to fit in one 64k segment, it is usually not very convenient to declare static addresses for sections – it is more convenient to put data sections directly after the main code block, so that as this block grows while the program is being developed, the data sections also get allocated from higher addresses so that these never overlap.

5) If the program only contains the BSS section (and possibly ZP) besides code, this can be accomplished by putting this directive at the end of your code section, directly after the last instruction or static data declaration:

```
     .BSS *
```

Note that the .BSS directives do not increase the „main" program counter, so in this case, after „.BSS *", the main PC will point to the beginning of the BSS section rather than to the end of it. So if you want to find out, where is the end of the memory occupied by the program, you will have to add the value of the pseudolabel __BSS__ which holds the current BSS offset. In the following example the label ENDP will hold the address of the first byte past the BSS:

```
     .BSS *
ENDP = *+__BSS__
```

For certain technical reason the BSS section (unlike ZP section) can be only one. Therefore its base can be declared only once in a program: an attempt at redeclaration will cause the assembler to throw an error. The BSS PC, however, may be changed at will. It is done with the .ORG directive, there is only one thing to remember, namely that *in this case the .ORG's argument is not an absolute address, but an offset relative to the BSS base*.

Therefore, if you, for example, want your BSS section to occupy two 16k banks of RAM, do this:

```
     .BSS $4000   ;define BSS base: the address of bank select RAM
     .BSS         ;switch to BSS section
BNK1 .DS 16384    ;assign first 16k
     .ORG $0      ;„reset" the BSS PC back to BSS base
BNK2 .DS 16384    ;assign another 16k
     .CODE        ;switch out of the BSS section
```

The labels BNK1 and BNK2 will both get assigned to the same address: $4000 and will be pointing to two overlapping areas, 16384 bytes each. It is of course up to the program to arrange things so that they do not physically overlap, but are properly assigned to different banks of RAM.

6) The DATA section works similarly, and in the (unlikely) case when your program contains only the DATA section besides code (and possibly .ZP), you define its base by putting this at the end of your program:

```
     .DATA *
```

And if you want to find out the first byte past the DATA section, do this:

```
     .DATA *
ENDP = *+__DATA__
```

Also the .ORG directive, when used with the DATA section, works the same way as in the BSS section.

7) When your program contains both DATA and BSS sections, and all this has to fit within the same address space with the code, you have to define the sections' bases so that they would not overlap. If you want to keep the most natural order of sections, i.e. CODE first, then DATA, and BSS at the end, the following will do the trick:

```
      .DATA *
      .BSS *+__DATA__
```

Warning: the declaration of the section base at the end of your code will cause the addresses declared within that section to get assigned different values in the first and the second assembly pass. This may cause obscure phase errors, for example:

```
      .ORG $2010
      .DATA
TEXT  .BYTE "HELLO!",$9B
      .BSS
TXTADR .DS 2
      .CODE
START .IF TEXT&$00FF
      LDA #<TEXT
      STA TXTADR
      .ELSE
      STZ TXTADR
      .ENDIF
      LDA #>TEXT
      STA TXTADR+1
EXIT  RTS
      .DATA *
      .BSS *+__DATA__
```

The label TEXT will get a value of $000000 in the first pass, and a value of $002010 in the second pass. Therefore the conditional will in the second pass cause the code to be 1 instruction shorter than it was in the first pass, so any label declared after the conditional (here EXIT) will trigger the phase error. The solution in this case is to .ALIGN the data section to a page boundary, but it is best to avoid using such tricks while doing references to the DATA and BSS sections, unless they are indeed going to be physically located in a separate address space each.

XVII. Defining structures in the memory and on the stack

1. The keywords .RSSET and .RS (reserve space) are aimed at defining a data structure without reserving the actual memory space for it.[1] The difference between .DS and .RS may be illustrated by the following examples:

```
      .ORG $2000
```

[1] The idea of these keywords and their operation was borrowed from HiSoft's Devpac for Atari ST.

```
DOT
?CX     .DS 1
?CY     .DS 1
?CZ     .DS 1
?CC     .DS 1
DSZ = *-DOT
```

After this, four bytes at address $2000 are allocated for the structure named DOT. The component DOT?CX is to be found at $2000, DOT?CY at $2001, DOT?CZ at $2002, and DOT?CC at $2003. The program counter value ('*') after this will be $2004. The variables in the structure, having been assigned to memory locations, are accessed just as other local labels, e.g.

```
     LDA DOT?CX
```

If you need to declare more DOTs, you have to either assign each a name (DOT1, DOT2, DOT3, ... DOT99 etc. which is absurd) or to declare empty space for the rest of them:

```
DOT
?CX     .DS 1
?CY     .DS 1
?CZ     .DS 1
?CC     .DS 1
DSZ = *-DOT
     .DS DSZ*99
```

Now compare with .RS:

```
COORDS  .RSSET 0
?CX     .RS 1
?CY     .RS 1
?CZ     .RS 1
?CC     .RS 1
CSZ = __RSSIZE__
```

First of all, these are not allocated in the memory and the program counter value ('*') does not change during definition. This only defines how a memory location (of size 'CSZ' bytes) will be internally structured when it will have been eventually allocated. The allocation is to be done as follows:

```
DOT     .DS CSZ
```

So now we have defined an abstract structure COORDS, which describes three-dimensional coordinates and color of an object, then declared a memory object named DOT which uses this structure to hold its individual coordinates and color. References to this structure can be made as follows:

```
     LDA DOT+COORDS?CX
```

This may at first appear more troublesome than the method which uses .DS, but is in fact very handy when the program has to manage not even multiple objects sharing the same internal structure, but rather multiple groups of these, yet not necessarily being allocated consecutively in the memory:

```
TWODOTS     .DS CSZ*2
TEMP        .DS 4
SAVEDOT     .DS CSZ
STACK       .DS 128
HUNDREDDOTS .DS CSZ*100
```

Any change of the internal organization and size of all these memory objects only requires redefining the structure COORDS without redefining all the individual objects or groups of objects which share this structure. *The mechanism described is similar to what C language does when the programmer is declaring a structure using 'typedef struct' then assigning memory to it using 'struct' – ELSA itself uses this technique internally to maintain e. g. multiple program counters*.

Remark: note that the label COORDS used in the examples above will actually be assigned an address equal to the value of the program counter ('*') at the time of .RSSET execution. So (quite differently than in the first example with .DS, where you can reference DOT instead of DOT?CX and get the same result), you cannot use COORDS alone here as an equivalent to COORDS?CX, because the former is an absolute address, while the latter is an offset. So LDA DOT+COORDS will just add the address of your memory object to a random address which was in the PC while the structure COORDS was being defined, which would be very wrong and would lead your program astray. *There is however a good reason why the assembler allows that and does not even generate a warning. This reason will hopefully become clear in the following section.*

Besides, using such a construction as DOT+COORDS (without specifying at which one of the internal variables of the structure we are aiming) would defeat the whole purpose of using the structure (which is to be able to freely alter the internal organization of multiple memory objects without reediting all of them and all of the existing code which is referencing them).

2. Another purpose of the .RSSET and .RS directives is to declare offsets for local variables allocated on the stack. It is actually very convenient to use stack to store variables which are in use only within the scope of a single subroutine instead of allocating static memory locations for them: the ZP storage is too short to waste it for that purpose, and, besides, static variables make the code not reentrant (which may be crucial in interrupt handlers, for example). Also there are programs which simply do not have free static space at their disposal or it is very limited (such as device drivers running under an operating system), and if they do find some, there is always a risk of an obscure conflict with another OS component or even an application program.

In all these cases allocating some stack space, which will vanish after use, comes in quite handy. For example:

```
MUL_A_BY_3
     PHA
     ASL
     ADC $01,S
     PLX
     RTS
```

It is all very easy when there is just one variable on the stack, but a slightly larger number of them may already become a trouble: when it is necessary to reedit the code and add or remove a variable, all offsets must be recalculated from scratch, and it is too easy to lose track what is where. And this is where RSSET/RS come in handy, for instance:

```
MUL_A_BY_14
     .RSSET 1
?M4  .RS 1     ;this one will be on the top of the stack
?M2  .RS 1
     ASL
     PHA
     ASL
     PHA
     ASL
     ADC ?M2,S
     ADC ?M4,S
     PLX
     PLX
     RTS
```

Note that both examples would be of the same size if using zero-page variables instead of the stack, so it is not the code size which we are gaining here: it is the use of static memory locations which is avoided this way.

The latter example also provides explanation on why the assembler allows the label declared straight before the .RSSET to retain its original value – in this case it is simply necessary (as the label is assigned a valid address of a subroutine) and for the assembler there is no way to tell the difference between this situation and the one described in chapter XVII.1 above.

Also note that the .ZP directive may be used for the same purpose, i.e. allocating variables, which are visible in the scope of a specific subroutine (in other words, variables local to that subroutine). This is wasteful, but in small programs, especially those which have to run on vanilla 6502, may be very convenient:

```
MUL_A_BY_14
     .ZP
?M4  .DS 1
?M2  .DS 1
     .CODE
     ASL
     STA ?M2
     ASL
     STA ?M4
```

```
    ASL
    ADC ?M2
    ADC ?M4
    RTS
```

Appendix A: MAE's directives not supported in ELSA

Directive	Synopsis
.24	In MAE this enables 24-bit address calculations (16-bit otherwise). In ELSA all expressions are evaluated as 32-bit values, so this directive has no purpose. It causes no error, however.
.EN	This is only supported as an alias for .END, and not as an alias for .ENDIF.
.MD, .ME, .MG	These are: Macro Definition, Macro End and Macro Global. They are not supported at the moment because ELSA does not support macros (yet).

Appendix B: MAE's bugs

There are several known bugs in MAE's compiler, here is how ELSA will behave in the same circumstances:

Code	Problem	MAE's behaviour	ELSA's behaviour
SOMELONGLABEL012 = 1	Label longer than 15 characters.	Likely crash.	Labels up to 240 characters are allowed.
LDA ($1234),Y	No such addressing mode.	Silently accepted as LDA ($34),Y	Accepted with a warning as LDA ($34),Y
LDX $800000	No such addressing mode.	Silently accepted as LDX $00	Error, improper addressing mode.
LDA #$<1234	Nonsense syntax.	Silently accepted as LDA #$01	Error, bad constant.
AA = BB BB = CC CC = 1 .ORG $0600 LDA AA	AA undefined during second pass.	Silently accepted, LDA AA compiled as LDA 32768	Error, undefined label.
.LONG 0*2	None apparent.	Compiled as .LONG $2A0000	Compiled as .LONG $000000
.WORD -256	None apparent.	Compiled as .WORD $FE00 (= -512)	Compiled as .WORD $FF00

Appendix C: ELSA's statistics

* Labels defined: ca 1950 (ca 35 KB)
* Source code lines: ca 13500

* Source code size: ca 147 KB
* Number of source files: 30
* Shortest source file: 71 bytes
* Longest source file: 24 KB

Time spent self-assembling	
Rapidus 20 MHz, Fast RD/WR, spinning platter HDD	20 sec.
Rapidus 20 MHz, ditto, Rapidus banked SDRAM RAM-disk	16 sec.

Time spent creating 6000 labels, 11 characters each	
Rapidus 20 MHz, Fast RD/WR, case-sensitive mode (default)	88 sec. (1 min. 28 s.)
Rapidus 20 MHz, ditto, case-insensitive mode	94 sec. (1 min. 34 s.)
Altirra 21.28 MHz, case-sensitive mode	83 sec. (1 min. 23 s.)
Altirra 21.28 MHz, case-insensitive mode	89 sec. (1 min. 29 s.)
Antonia 1.77 MHz, case-sensitive mode	1099 sec. (18 min. 19 s.)

The code to test the 6000 labels:

```
      .REPT 6000
#
      .ENDR
```

Time spent generating 32768 times LDA #$FF	
Rapidus 20 MHz, Fast RD/WR	9.60 sec.
Altirra 21.28 MHz	8.68 sec.
Antonia 1.77 MHz	101 sec. (1 min. 41 sec.)

The code:

```
      .REPT 32768
      LDA #$FF
      .ENDR
```