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

An overview

(c) 2020-2026 KMK/DLT

Version 1.04

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 34k.

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.

Warszawa, 01/06/2026

Table of contents

Preface ... 1
I. Operation ... 4
II. Command line arguments ... 5
III. General assembler syntax ... 7
IV. Labels ... 8
    1. Label name ... 8
    2. General syntax ... 8
    3. Local labels, MAE-style ... 8
    4. Local namespaces ... 9
    5. Definition order ... 9
V. Expressions ... 10
    1. Value types ... 10
VI. Constants ... 11
VII. Unary operators ... 12
VIII. Binary operators ... 14
IX. Directives ... 16
    1. Target CPU control ... 16
    2. Conditionals and general assembly control ... 18
    3. Repetitions ... 20
    4. Fixed data definition control ... 21
    5. Code generation control ... 23
    6. Memory allocation control ... 24
    7. Label control ... 25
    8. Input/output control ... 26
    9. Listing control ... 27
    10. SpartaDOS X support ... 28
        10.1 Example code: 6502 emulation mode ... 30
        10.2 Example code: 65C816 native mode ... 30
X. Pseudo-labels ... 32
XI. Pseudo-instructions ... 34
XII. Instruction aliases ... 46
XIII. Alternative syntax for some instructions ... 47
XIV. Divergences from the WDC-recommended syntax ... 49
XV. Declaring zero-page locations ... 50
XVI. Sections DATA and BSS ... 52

1. BSS section....52
1.1. BSS PC....54
1.2. BSS section in 8-bit REL blocks....54
1.3. BSS section in 16-bit REL blocks....54
2. DATA section....55
2.1. DATA section in REL blocks....56
3. Combining DATA and BSS sections....56
4. DATA/BSS quirks....56
XVII. Defining structures....58
1. Defining a structure in memory....58
2. Defining a structure on the stack....60
XVIII. XREFs and XDEFs....63
1. XREFs....63
2. XDEFs....64
XIX. Starting relocatable programs....65
XX. Error messages....66
XXI. Warning messages....70
Appendix A: SpartaDOS X system loader relocation rules....72
Appendix B: MAE's directives not supported in ELSA....74
Appendix C: MAE's bugs....75
Appendix D: ELSA's source statistics....76

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.

Altirra users, please use a version of the emulator not older than 4.20-test6.

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.

Limitations:

a) Maximum source line length: 255 characters
b) Maximum length of a label: 240 characters
c) Maximum line count per file: 16777216 (8-9 hours of assembling)
d) Maximum global line count: 4294967296 (3 months of assembling)

II. Command line arguments

The syntax is:

ELSA [options] source_file_name.ext [options]

The options are:

Switch	Function
/A	Disable caching source code in the memory. In that case the source will be read from the disk in both passes. Useful for large assemblies when there is not enough free RAM in your computer.
/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
/E	Count line numbers from 1.
/L[fname.ext]	Generate assembly listing during the second pass. The listing will be printed to the specified file, or, if no file was specified, it will appear on the screen. In both cases it will be 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.
/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

1. 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 pseudo-labels (explained somewhat below) and labels declared implicitly 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.

2. 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.

The percent-sign ("%") appended at the end of a label being declared is a special declarator: that label will be exported as an externally accessible symbol (XDEF). Otherwise the use of that sign in labels is not allowed.

3. 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 sys-

tem as a simple and elegant solution of the problem of label locality.

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

4. 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, from within the main program, to accidentally reference something, that 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.

5. 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 in 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.

1. 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).

As of ELSA version 0.98, if the CPU selected is 65C816 (which is the default), the results of address calculations will be cut down to 24 bits, and to 16 bits otherwise.

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 expres-

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

1. Target CPU control

Directive	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 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. .CPU 16 implies .OPT P-. If the current target CPU is not 65C802 or 65C816, .CPU 16 will generate a warning. *As of ELSA v.1.04 .CPU 16 is implied after .REL $x3.*	.cpu 16
.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	.ib

	the directive and generate a warning.	
.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

2. Conditionals and general assembly control

Directive	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. In .ZP, .BSS and .ORG blocks the PC is adjusted by adding offset to its current value. In .REL and .DATA blocks – by generating a suitable number of zeros. *It is to be kept in mind that .ALIGN operates on PC value as it is during the assembly time – in .REL blocks it has no influence on how the program will get aligned in the destination memory.*	.align $0100
.ELSE	This inverts the result of the expression	(see .if)

.EL	evaluation made by the .IF directive.	
.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 / .ENDIF).	.error "lm=",lm
.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 sysequ rtclock=18 .endif
.OPT	Specify additional assembly options: * B- - disable warnings on BRK instruction. * 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. This only applies to the .ABS and .ORG headers, .REL headers are unaffected. * P+ - enable warnings on cross-page branches. P- is the default, unless /P was	.opt h-,f- .opt h+,f+

	specified in the command line. If it was, P+ or P- used in the code have no effect. .CPU 16 implies .OPT P-. * W- disable all 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. * Z+ enable silent conversion of .ORG $00xx to .ZP $xx. Z- is the default.	
.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:",*

The general remark to the conditionals .IF, .IFDEF and .IFNDEF is that you should watch out if the conditional expression keeps the same logical value during both assembly passes. For instance, doing something like this:

```
  .ifndef foobar
foobar = 1
  .endif
```

is asking for trouble, because the label *foobar*, when undefined during the first pass, will anyway be defined during the second pass. In this case the ELSA's symbol table will lose consistency and weird errors will get reported.

3. Repetitions

Directive	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	lda math .rept 8 asl

	end of that block should be marked with .ENDR. Within the block, the pseudo-label __REPT__ contains the number of the current iteration, 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.	rol math+1 bcs skip# inc nuls skip# .endr

4. Fixed data definition control

Directive	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 $ff,'a,'b,0 .byte <val,>val .byte size*2+1 .byte "Hey",$9b .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 l b	Define Constant-filled block. The consecutive 16-bit "l" number of bytes will be filled with the 8-bit value of "b". A value of 0 for "l" 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	.float "3.14","0" .float 256*3,-2+1

	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 about normally, i.e. 4294967295.	
.HEX .HE	Generate a series of hexadecimal numbers. It 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.	.hex ae 19 00 44
.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

5. Code generation control

Directive	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 .code
.INIT w	Specify a value for the INITAD vector ($02E2). The corresponding init segment will be generated immediately. *This keyword is allowed for .REL blocks, but has no effect – the binary loader does not support partial execution of relocatable binaries.*	.init setup
.MC w	Move the following code to a different address than the one specified by the .ORG or .ABS directives. The difference is that the .ORG/.ABS address will be used as the program counter to calculate addresses, while the .MC address will be used to create binary headers of the absolute type (with the signature of $FFFF or $FFFA). 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 be-	.org $2000 .mc $0600

	ing 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 w	Specifies the address where (a portion of) the object code has to be stored in the memory. This generates the Atari-style absolute binary segment (type $FFFF). The assembler is able to generate up to 1024 separate segments within one binary file. Related: .ABS *Remarks: when 'w' is equal to the current PC value, no new header will be generated.*	.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. *This keyword is **allowed** for .REL blocks.*	.run start

6. Memory allocation control

Directive	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 l	Declare Storage. In all sections besides DATA, it reserves the "l" number of bytes as uninitialized data array (as small as 1 byte – 0 will generate a warning). In other	.ds 32

	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 l 0. See also the remarks on .REL.	
.RS w	Define size of an individual variable within a structure. The space will be of "w" bytes (0 is allowed, it may be used to create an „union", i.e. an alias name for a field already named inside a structure). See below the section on *Defining structures in the memory and on the stack*.	.rs 2
.RSSET w	Beginning of a structure. The "w" is the offset of the structure's first element. Negative values allowed.	.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*.	.zp $80 .zp

7. Label control

Directive	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*	.set zpc = $0100

	official as of 0.93.	
.USED lb,...	Mark the specified label(s) as "used", i.e. referenced. It may happen that a label is assigned to a memory variable which is used implicitly (e.g. as one of two byte values being fetched or written to at once as a word) or by some external code. So this keyword is useful to improve the reliability of the final statistics ELSA prints on the screen about the number of the labels defined and used, as well as it prevents you from inadvertently deleting such a variable from the source code. *The label specified must be defined in the second pass.*	.used foo,bar

8. Input/output control

Directive	Synopsis	Examples
.BIN fn .BI fn	Binary Include. About the „fn" parameter, see .INCLUDE below. 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 physically read out on both passes.*	.bin pic.bmp .bin $bar.dat
.INCDIR fn	Define a default directory for the .INCLUDE directive. When the given string does not end with a path separator, the assembler will append one. When no argument is given, any previously defined string will get deleted.	.incdir >foo>
.INCLUDE fn .IN fn	Includes another source file found at the given pathname. When no device specific-	.include math.s .include $bar.s

	ation was given, a „D:" is prepended automatically. A special character $ at its first occurrence within the pathname will get replaced with the string that has been defined using .INCDIR. For example, after .INCDIR >FOO>, a D2:$BAR.S specified as an argument to .INCLUDE will be expanded into D2:>FOO>BAR.S. And when nothing was defined, the $ will simply get removed from the pathname. These directives can be nested (i.e. the included file may contain .INCLUDE directives), just remember that the stack space is not unlimited.	
.OUT fn .OU fn	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
.STDINC fn	As in *STanDard INClude*. This is basically the same as .INCLUDE, with one additional feature, namely that all labels defined in the file being included are internally marked as „standard". The standard labels are not reported as defined or used at the end of the assembly – and this may be useful, if you want to know, how many labels are defined, and how many of them are never used, by the local source files of your program, not counting the global system equates it possibly also includes. In other words, using this keyword for common includes causes assembling your programs with /V parameter make actual sense.	.stdinc vbxe.i .stdinc $sysequ.i

9. Listing control

Directive	Synopsis	Examples

.LC	Switch off ("clear") the assembly listing.	(see .LS)
.LL	List just the following line.	.ll sta 24*offset+l,x
.LS .LS fn	Switch on ("set") the assembly listing. When no additional parameters are given, the listing will be displayed on the screen. When the „fn" (or file name) was given, the listing will be written to the specified file instead. You can disable it temporarily with .LC then re-enable with .LS, but the parameter „fn" can be used only once – any later time it will be ignored.	.ls .org $0600 start jmp $e477 .end .lc

10. SpartaDOS X support

Directive	Synopsis	Examples
.ABS w	Like .ORG, but enables the assembler to generate SpartaDOS-style absolute binary headers (signature $FFFA) instead of Atari-style ones ($FFFF). ABS segments (unlike the .ORG segments) allow to use external label (.XREF or .XREFW) references within themselves. .ABS cannot be used, when some data or code was already generated with .ORG – and vice versa, .ORG cannot be used after .ABS.	.abs $2000
.REL b	Generate SpartaDOS X relocatable binary block ($FFFE/9/8). The „b" parameter is the control byte to be inserted into the header: when its bit 7 is set to 1, an „empty" block will be generated, i.e. the instruction for the SpartaDOS X system loader to allocate up to 65536 bytes of memory for uninitialized variables; and when the bit 7 is cleared, the „relocatable text segment" will be generated, i.e. a position-independent block containing up to 65536 bytes of code and/or data. *For the*	.rel $00

	meaning of lower 7 bits please refer to the SpartaDOS X programming documentation. The .REL segments (unlike .ABS segments) allow to use both XREF and XDEF declarations within themselves. In the .REL segments with b < $80 (relocatable text segments) the .DS directive generates zeros, whereas on b>= $80 (memory allocation segments) the .DS directive generates offset (i.e. empty space).	
.XREF lb	Declare a label, named „lb", as an external symbol. Its value is unknown at the assembly time, it gets defined when the program is loaded to the memory. It is assumed that such a symbol is an address pointing to an object in the global memory: therefore the only form of arithmetic on those which makes sense is adding or subtracting constant values (offsets). The result of arithmetic performed on an external label may be assigned to a new label, in that case the latter becomes an alias to the former (inheriting its XREF status and the originally associated symbol name too). Also see below.	.xref PRINTF
.XREFW lb	The same as .XREF, except that it declares the specified symbol as „weak". A weak symbol, when not defined during loading a binary requesting it, does not cause an error, it is ignored instead. It is useful to check for symbols optionally present in the system without engaging system calls such as S_LOOKUP. *This feature is supported as of SpartaDOS X 4.50*.	.xrefw _RAWCON

The standard 6502 SpartaDOS X loader only allows a program to contain up to 7 .REL-type blocks, therefore ELSA up to v. 0.99 also had this limitation. As of version 1.0, however, it allows up to 127 blocks: such binaries are handled by the extended 65C816 loader (embedded in the 65816.SYS driver), but even using that one the 6502 address space is still limited to 7 relocatable blocks: the extra ones are allowed

for the 65C816 High RAM only.

Remark 1: in relocatable programs it is not allowed to split-up addresses into separate bytes or words, because the binary loader would not be able to fix them up properly.

Remark 2: it is not allowed to use the PC-relative addressing mode (Bxx, BRL, PEA/PER) to reference a block of code different than the one which contains the instruction. This is most visible in branch instructions: it is not allowed to do a branch (even the BRL) from one program block to another one, because at the assembly time it is not possible to predict where the respective blocks will get loaded to the memory, and as a consequence, it is also impossible to calculate the relative offset between them. But, of course, there are no objections to use PC-relative instructions to reference locations within the same („their") block of code.

10.1 Example code: 6502 emulation mode

```
      .out exa.com       ;output binary name

      .xref PRINTF       ;import global symbol

      .rel 0             ;mem index 0 (conventional 64k)
@exa%                    ;export @EXA to global symbols
      jsr PRINTF         ;call SpartaDOS X library
      .byte $9b,"Hello world!",$9b,0
      rts                ;end program
```

10.2 Example code: 65C816 native mode

Remark: the program requires the EXT816.SYS module to be loaded.

```
      .out exb.com       ;output binary name

      .xref H_PRINTF     ;import global symbol

dosvec = $0a             ;define system vector location

      .rel 0             ;mem index 0 (conventional 64k)

      .cpu 16            ;tell ELSA we will be switching modes
      .rb                ;tell ELSA all registers are 8-bit
@exb%                    ;export @EXB to global symbols
      clc                ;switch to native mode
      xce
      jml >greetme       ;call code loaded to high RAM
```

```
      .rel 3              ;mem index 3 (high RAM)

      .rb
greetme
      jsl >H_PRINTF       ;call 65C816 extension library
      .byte $9b,"Hello world!",$9b,0
      pea (dosvec)        ;end program
      cop #$00
```

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.
__LINE__	A number of the current source file line.
__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__ __M65802__	1, if the currently selected target CPU is 65C802, and 0 otherwise.
__M65C816__ __M65816__	1, if the currently selected target CPU is 65C816, and 0 otherwise.
__RELCNT__	Total count of .REL blocks defined in the program (0 if none). Valid during second pass only.
__RELNUM__	Current .REL block number (0 if .ABS or .ORG block).
__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	per *ret*-1 brl *address* *ret*

	bytes, 10 clock cycles.	
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*[,x] bne *skip* dec *address*+1[,x] *skip* dec *address*[,x] or: dec *address*[,x]
DWA *address* DWA *address*,X	Decrement Word using Accumulator. Like DEW, but makes explicit the intermediate use of the Accumulator (hence no warning on 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/DEC/STA series of instructions (10 cycles for the zp, 12 for the absolute addressing mode), which leaves the NZ	lda *address*[,x] bne *skip* dec *address*+1[,x] *skip* dec *address*[,x] or: lda *address*[,x] dec sta *address*[,x]

	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 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.	inc *address*[,x] bne *skip* inc *address*+1[,x] *skip* or: inc *address*[,x]
Jcc *address*	Jump (do actual JMP) if the condition "cc" is met. It is an absolute version of conditional branches *Bcc* with identical meaning, but the jump range is 64k instead of +/-128 bytes. 5 bytes, 5 cycles taken, 3 cycles not taken (or 4, when the instruction components cross a page boundary in 6502 mode).	b*cc 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*
JSK abs	Jump to Subroutine long within K segment. With this you can call far procedures (normally requiring a JSL) from within the same program block, thus avoiding problems with address relocation. 4 bytes, 9 cycles. The subroutine called must end with an RTL.	phk jsr *address*
JSK (abs,x)	As above, indirect indexed X. 4 bytes, 11 cycles.	phk jsr (*address*,X)
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 cycles.	pea *ret*-1 jmp (*address*) *ret*
JVC	Jump if V flag Clear.	bvs *skip* jmp *address*

		skip
JVS	Jump if V flag Set.	bvc *skip* jmp *address* *skip*
Lcc *offset*	Branch (do actual BRL) if the condition "cc" is met. It is a 16-bit version of a conditional branch *Bcc* with identical meaning, but the branch range is +/- 32k instead of +/-128 bytes. 5 bytes, 6 cycles taken, 3 cycles not taken (or 4, when the instruction components cross a page boundary in 6502 mode). Notice: the *Jcc* jumps are faster (5 cycles when taken), whereas the Lcc variants generate shorter relocatable binaries (no fixup necessary).	b*cc skip* brl *address* *skip*
LCC *address*	Long branch if Carry Clear.	bcs *skip* brl *address* *skip*
LCS *address*	Long branch if Carry Set. As above, with the opposite condition.	bcc *skip* brl *address* *skip*
LEQ	Long branch if EQual.	bne *skip* brl *address* *skip*
LGE	Long branch if Greater or Equal. Same as LCS.	bcc *skip* brl *address* *skip*
LLT	Long branch if Lesser Than. Same as LCC.	bcs *skip* brl *address* *skip*
LMI	Long branch if MInus.	bpl *skip* brl *address* *skip*
LNE	Long branch if Not Equal.	beq *skip* brl *address* *skip*
LPL	Long branch if PLus.	bmi *skip*

		brl *address* *skip*
LVC	Long branch if V flag Clear.	bvs *skip* brl *address* *skip*
LVS	Long branch if V flag Set.	bvc *skip* brl *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	*loop* ... bcc *loop* or:

	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 otherwise. Therefore the resulting object code may accordingly vary in code size and execution time.	*loop* ... jcc *loop*
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 [lda $2000,y sta $3000,y iny	bcc *skip* ... *skip*

	] 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 are available as for the SBC, this pseudo-instruction is just 1 byte longer and takes 2 cycles more. Counterpart: ADD.	sec 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 *adr*	Branch if Greater or Equal.	BCS *adr*
BLT *adr*	Branch if Lesser Than.	BCC *adr*
CLR *adr* CLR *adr*,x	Clear the specified memory location.	STZ *adr* STZ *adr*,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 (*zp*)	Push Effective address, Indirect (move word from ZP to stack)	PEA (*zp*)
PER *adr*	Push Effective address, Relative	PEA *adr*
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 for 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 two zero bytes ($00, $00) to be generated to the object file, but f.e. BRK #$80 will generate $00 $80 instead.

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 base of the zero-page variables to be declared for the entire program. *Declaring this is required for the first .ZP directive in your program.* But for all following .ZP directives this number should be omitted: the subsequent ones 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 declared 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: the code on the other side, and the 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).

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.

1. BSS section

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. *Do not worry: even if a BSS variable is declared at virtual address lesser than $0100, the assembler will never assume that a zero page addressing mode should be used in the reference.* But otherwise everything works as in ZP sections as long as your BSS is located in a dedicated 64k segment, forming an ad-

dress space truly separate from code's.

However, in programs which are smaller than 64k it is usually inconvenient to keep the BSS segment in a separate address space. To put the BSS section to the same address space where your code is living, just specify the BSS base address in the parameter, for example:

```
.bss $8000
```

This will make the assembler to allocate your BSS variables from the specified address onwards. *Note: this only defines the BSS base, and (unlike .ZP adr) does not switch sections. To switch to the BSS section you still have to use .BSS keyword without the parameter.*

If your program has to fit in one 64k segment (as 100% 6502 programs do), it is usually not very convenient to declare static addresses for sections – it is more convenient to put the BSS section directly after the main code block, so that, as this block grows while the program is being developed, the BSS section also get allocated from higher addresses so that these never overlap. To accomplish that, put the following directive after the last byte of the defined data or code in your program:

```
.bss *
```

If you do not add this, all your .BSS variables will get allocated starting at the virtual address $000000 and you will have to handle this situation on your own (for example, by allocating a suitable memory block at a 64k boundary and loading the bits 16-23 of its address to the DBR register).

Note that the .BSS directives do not increase the code's program counter, so in this case, after „.BSS *", the code's 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 true end of the memory occupied by your 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__
```

1.1. BSS PC

For certain technical reason the BSS 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: 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.

1.2. BSS section in 8-bit REL blocks

In SpartaDOS X „traditional", 8-bit relocatable executables everything seemingly works as depicted above. But in fact there is no real BSS section in this case: the variables declared inside .BSS/.CODE scope are created as usual, but the „.BSS *" at the end of a .REL 0 block will implicitly aggregate them into a separate .REL $8x block allocating memory. The same applies to the .REL 2 or .REL 4 type blocks.

Therefore you may get an impression that there are multiple .BSS sections possible, but beware: every .BSS * appended at the end of a .REL block is in fact an equivalent to .REL $80 – and the maximum number of 8-bit compatible .REL blocks in a program is currently only 7!

Also, for that same reason – no real BSS – you cannot use the .ORG directive to change the BSS PC value.

1.3. BSS section in 16-bit REL blocks

The situation in extended relocatable blocks – those which are be-

ginning with the .REL 3 type directive – is slightly different: a .BSS * directive used at the end of such a block does not create a separate block, but both code and the BSS section get aggregated into one block containing code at the beginning, with all the BSS variables directly following the end of the code. The size of such a block (code+data) may not exceed 65536 bytes.

The maximum block count in this case is 128 and each one may contain its own BSS section. Example:

```
      .xref H_PRINTF

      .rel 3                  ;code block no 1
      .rb
@test%
      .bss                    ;its BSS variables
var1  .ds 1
      .code
      lda #$01
      sta var1
      jsl H_PRINTF
      .byte "%b",$9b,0
      .word var1
      jml >blk2
      .bss *                  ;fix BSS section here

      .rel 3                  ;code block no 2
blk2  .bss                    ;its BSS variables
var2  .ds 1
      .code
      phk                     ;do not forget to switch segments!
      plb
      lda #$02
      sta var2
      jsl H_PRINTF
      .byte "%b",$9b,0
      .word var2
      rtl                     ;end program
      .bss *                  ;fix BSS section here
```

2. DATA section

The DATA section 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
```

The remarks about addressing are the same as in the case of the BSS section. You define the DATA section 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 within the DATA section, works the same way as in the BSS section.

2.1. DATA section in REL blocks

In the current implementation it is not possible to use the .DATA keyword inside a .REL block – this will throw an error.

3. Combining DATA and BSS sections

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 appended at the end of the last code block will do the trick:

```
     .data *
     .bss *+__DATA__
```

4. DATA/BSS quirks

The declaration of the section base at the end of your code will cause the addresses declared within that section to get assigned differ-

ent 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.

Similar trouble may be caused by the .ALIGN directive used within a DATA or BSS section, if these sections are to be assigned a physical address at the end of the first pass. In this case, if a portion of the BSS section has to be aligned, it is better to align the end of the code section – the BSS section which follows will then be automatically aligned.

XVII. Defining structures

1. Defining a structure in memory

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
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
```

To access this, you need to use an indexed mode, probably with 16-

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

bit index registers:

```
     .iw
     ldx #dot_number*dsz
     lda dot?cx,x
```

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 itself, 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 members 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 re-editing all of them and all of the existing code which is referencing them).

Warning: the assembler does not check if a construction like LDA DOT+COORDS?CX makes sense at all, i.e., if the memory object DOT may contain a structure called COORDS: it is thus perfectly possible to reference a structure to a wrong object which is not supposed to contain it.

2. Defining a structure on the stack

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 re-entrant (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 re-edit 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 is 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
```

XVIII. XREFs and XDEFs

1. XREFs

As of version 0.94 ELSA supports external symbols for use with SpartaDOS X relocatable binaries. There are two types of external symbols: a) the symbols which have been defined externally in global memory by the system, your program may want to import these (XREFs); b) the symbols your program wants to export to the global memory for the system or other programs to use (XDEFs).

XREFs are declared using the keyword .XREF. The argument to this keyword is the label of an external symbol which is predefined by the system and otherwise known (see SpartaDOS X programming documentation), for example:

```
      .xref COMTAB
```

will declare label 'COMTAB' so that your program may reference it without further definition. When your program gets assembled, that label will cause the assembler to generate an XREF record and append it to the resulting binary file. At loading time, the SpartaDOS X relocating loader will take that into account and, if the symbol exists, will resolve it to an address. XREFs are allowed in .ABS, .REL and .DATA segment types. *An XREF record will contain a symbol name converted to upper-case and cut down to 8 characters, when it is longer than that, or space-padded to 8 characters otherwise.*

As for that particular example, COMTAB is the SpartaDOS internal structure containing many fields located at offsets negative and positive from the point the symbol points to. Using references such as COMTAB-4 or COMTAB+255 may be inconvenient, therefore you can assign them labels, for example:

```
      .xref COMTAB

divend = COMTAB-6
decout = COMTAB-19
decout2 = COMTAB-21
```

The references to these „secondary" labels, when used in your pro-

gram, will cause the assembler to generate XREF records for the originating symbol. *An XREF-type label is declared as type value (not address!), and it is to be kept in mind that the result of any arithmetic, where one of the components is XREF, will also be XREF. It is also not allowed to have two or more XREFs in one equation – it would not make sense as the exact value of an XREF is unknown during assembling.*

Notice: an XREF label gets a value of 0 by default (the actual address being filled in at loading time, as explained above). ELSA's integer evaluator performs 32-bit computations, thus any calculation sets all the 32 bits of the result. Accordingly, COMTAB-1 = 0-1 = -1, and -1 is $FFFFFFFF in 32-bit integer representation. Thus e.g. LDX COMTAB-1, intended as being in absolute addressing mode, could easily be „cast" to the long absolute addressing mode, which in turn does not exist for LDX! To prevent such unpleasant surprises, the result of an arithmetic expression which contains an XREF is (as of 0.95) cut down to 16 bits. Please keep in mind that this does not prevent XREFs from referencing locations anywhere in the 24-bit address space: this just prevents these references from spanning 64k boundaries (as e.g. $000000-1 will wrap back to $00FFFF – but the relocating loader is still able to fill all three bytes of a longword, when applicable).

2. XDEFs

XDEFs, on the other hand, do not require separate directives, a label is declared as a symbol to be exported using the declarator '%'; for example, this declaration:

```
@grep%
```

will cause the assembler to generate an XDEF record defining the symbol @GREP. At loading time the SpartaDOS X relocating loader will append that symbol to the global list, and remove it, when your program terminates (unless your program is a TSR). XDEFs are allowed in .REL segment type only. *Unlike XREF, the XDEF status is not preserved in assignments; in other words, if your assign the @grep label's value as defined above to some other label, the latter will not become an XDEF.*

XIX. Starting relocatable programs

SpartaDOS programs are by default started at the beginning of the block which was consecutively the first one to be loaded to the memory. Nevertheless a relocatable program can be started from another point than its very beginning, and it can be done without additional support code.

It is just enough to know that the RUNAD vector ($02E0) is still operational and will be used by the loader, when set; and that pointers located within an .ABS block will get fixed up during loading. Therefore, assuming that *START* is a label pointing to a location in a relocatable block, write:

```
.run start
```

This will generate an .ABS block loading the appropriate word pointer to the RUNAD vector. The program will then be started from that label rather than from the beginning.

The use of .INIT in a relocatable program is allowed by the assembler and a formally correct binary file will be generated, but the init records will be ignored by the loader.

XX. Error messages

Errors are generated when the assembling process cannot be continued due to a condition. The assembling is aborted then and ELSA returns to DOS. On SpartaDOS X the error code is handed back to the system, so that it can be detected and acted upon in a batch file, for example.

Code	Error message	Synopsis
2	BAD EXPRESSION	Invalid syntax of an arithmetic expression (e.g. unpaired parentheses of a cast operator and such).
3	BAD CONSTANT	A numeric constant contains invalid characters.
4	BAD OPERATOR	An arithmetic expression contains an unknown operator.
5	BAD DECLARATION	A label to be declared may only begin with a letter, ?, @ and underscore.
6	BAD ARGUMENT	Improper format of arguments in .FLOAT, .PRINT, .ERROR, .OPT and .SET.
7	ILLEGAL CHARACTER	Illegal character within a label's body detected.
8	LINE TOO LONG	A source line is longer than 255 characters (+ EOL).
9	UNDEFINED	Undefined label or symbol.
10	MISSING]	A [] block was opened, but not terminated before the end of its source file.
11	LOCAL NOT ALLOWED	A local label cannot be used as an argument to .LOCAL or .XREF directives.
12	MUL/DIV OVERFLOW	The result of a multiplication or division does not fit in 32 bits.
13	DIV BY 0	Division by zero.
14	VALUE OUT OF RANGE	The value given as an argument is out of the range allowed for the specific keyword.

Code	Error message	Synopsis
15	UNKNOWN KEYWORD	There is no such CPU instruction or assembler directive.
16	DECLARED TWICE	An attempt to declare a label or symbol that was already declared.
17	NO MATCHING GLOBAL	A MAE-style local label was declared outside the scope of any global one.
18	BAD SYNTAX	1) Invalid qualifiers in a label, 2) invalid operators or separators in .BYTE, .SBYTE, .CBYTE, .ERROR, .PRINT and .SET.
19	UNEXPECTED EOL	An end of line character has been encountered in the middle of an expression.
20	OUT OF RAM	The assembly requires more memory than available.
21	CODE/DATA NOT ALLOWED	There was an attempt to insert code or initialized data into the ZP or BSS segment.
22	BLOCK COUNT EXCESS	A SpartaDOS X relocatable binary has more than 128 blocks.
23	TOO MANY SEGMENTS	An AtariDOS binary file has more than 1024 segments.
24	PHASE/DECLARATION	An address generated during the second assembly phase does not match its value that had been generated during the first phase. This usually means a forward-declaration of a zero-page label; or, in other words, a zero-page label was used before its declaration.
25	NO SUCH ADDR. MODE	No such addressing mode for the selected target processor.
26	IMPROPER ADDR. MODE	No such addressing mode for that instruction.
27	BRANCH RANGE	An attempt to branch to a location that is too far.
28	MUST EVALUATE	The value of the expression must be known in the first assembly pass.

Code	Error message	Synopsis
29	ALREADY OPEN	The file specified in .OUT, .INCLUDE, .STDINC, .BIN is already open.
30	KEYWORD NOT ALLOWED	1) .MC is not allowed outside fixed-address code segments, 2) .ORG and .DATA are not allowed in SpartaDOS X relocatable programs 3) Scc and Rcc are not allowed in ZP segments.
31	Scc: NOTHING TO SKIP	Scc is the last instruction in a source file.
32	FILE TOO LONG	A source file is longer than 16777215 lines.
33	BASE EXPECTED	The first .ZP directive used in your program must declare an explicit base address on zero-page, e.g.: `.zp $80`
34	TOO MANY LEVELS	Too many nested directives .IF or .INCLUDE, or [] code blocks.
35	NO MATCHING .IF	An .ELSE or .ENDIF was encountered and there was no matching .IF previously.
36	MISSING .ENDIF	An .IF block was not terminated before the end of its source file.
37	NO NESTING	A directive was nested which cannot be nested.
38	USER-DEFINED	Generated by the directive .ERROR
39	NO MATCHING .REPT	An .ENDR was encountered and there was no matching .REPT previously.
40	MISSING .ENDR	A .REPT block was not terminated before the end of its source file.
41	INTERNAL ERROR	A label, which was successfully declared, cannot be found in the symbol table. This means that the symbol table is corrupt, which can be amounted either to a bug in the assembler or a hardware problem.
42	Rcc: NOTHING TO REPEAT	Rcc is the first instruction in a source file.

Code	Error message	Synopsis
43	NO MATCHING [	A ']' was encountered and there was no matching '[' previously.
44	LABEL EXPECTED	The .XREF directive expects a label as an argument.
45	ADDRESS NOT ALLOWED	The .SET directive cannot be used to re-declare addresses. Also, splitting an address into LSB and MSB is not allowed in SpartaDOS X relocatable programs.
46	XREF CONFLICT	Two or more external symbols are used within one arithmetic expression.
47	XDEF NOT ALLOWED	XDEFs are only allowed in relocatable SpartaDOS X binaries.
48	XREF NOT ALLOWED	XREFs are not allowed in AtariDOS binaries.
49	BLOCK TOO LONG	Program block exceeds 65535 bytes.
50	RELOCATION NOT ALLOWED	In a relocatable program, the long absolute addressing mode was used to reference the same program block, or the short absolute addressing mode was used to reference a different program block (see Appendix A).
51	PCREL NOT ALLOWED	PC-relative addressing modes (branches, PEA) cannot be used to make references between different blocks of a relocatable program (see chapter IX, section 10).
52	DATA NOT ALLOWED	DATA segments are not allowed in SpartaDOS X relocatable programs.
53	ADR. INCOMPATIBLE	An attempt was made to perform arithmetics on two incompatible addresses, e.g. to perform subtraction to calculate offset between two addresses which do not belong to the same memory type in a SpartaDOS X relocatable binary program.

XXI. Warning messages

Warnings are generated when the assembler detects a condition, which does not prevent the program from being assembled, but which may be easily overlooked and it may prove useful to turn programmer's attention to that. The assembling is not aborted.

Warnings can be disabled using /Q in the command line options, or using .OPT W- in the source file. The effect of the latter can be reversed using .OPT W+.

Warning message	Synopsis
ADD/SUB OVERFLOW	The result of addition or subtraction does not fit on 32 bits.
VALUE EXPECTED	Keyword's argument should be a value rather than address.
FISHY MATHS	Adding an address to an address, or multiplying/dividing an address by an address.
SHORT BRANCH?	A long branch (Jcc, Lcc) or a JMP can be replaced with a short equivalent (Bcc/BRA).
NUMBER TOO BIG	Instruction's operand has more bits than expected.
OFFSET ZERO	The operand of a relative instruction is 0.
CROSS-PAGE BRANCH	A short branch may be crossing page boundary (pretty useless in relocatable code).
NO EFFECT ON TARGET CPU	REP or SEP was used to set or clear the MX bits and the target CPU declared is not 65C802 or 65C816.
ACC. CLOBBERED	A pseudo-instruction (such as PHR) clobbers the accumulator contents.
EMPTY []	An [] block was declared with nothing in it.
EMPTY REPT	A .REPT block was declared with nothing in it.
REPT 0	A .REPT block was declared with 0 repetitions.
NZ LEFT INCONSISTENT	A pseudo-instruction (such as DEW) does not leave the NZ flags in a state reflecting its result.

Warning message	Synopsis
EXTRA CHARS IGNORED	There is some garbage after the last meaningful piece of text in the source line.
WRONG CPU	The instruction does not belong to the target CPU instruction list (e.g. a BRA on 6502). Use proper directives to let the assembler know what target is meant.
ADDRESSING OVERKILL	Use of a 24-bit addressing mode on 65C802 target.
SUSPICIOUS ARG SIZE	Immediate argument is an ASCII character, but the argument size selected is not 8-bit.
KEYWORD IGNORED	A directive was overridden by a corresponding command line switch.
SEGMENT PC EXCESS	The program counter crossed the 64k boundary.
NULL JUMP	A JMP or JML to an instruction located directly behind the jump, e.g. JMP *+3.
MX SET IN EMU MODE	REP or SEP was used to set or clear the MX bits while the 65C802/65C816 CPU is in emulation mode. To suppress this warning, use the directive .CPU 16.
.RS W/O .RSSET	An .RS directive was used without a previous .RSSET.
SEGMENT LEN EXCESS	The program segment length exceeded 65536 bytes.
DECLARED 0 BYTES	A .DS or .DC directive's size parameter is 0.
DECLARE DATA BASE	The data segment address is not declared in the program.
BREAKPOINT?	A BRK instruction has been encountered. The reason why it generates a warning is that it may be a debugging breakpoint, unwanted in the final version of your program, which was inadvertently left in the source code. This warning can be disabled using .OPT B-
PROMOTED TO ABS,Y	A zero-page reference has been promoted to an absolute addressing mode due to the Y-indexing.

Appendix A: SpartaDOS X system loader relocation rules

Normally, the SpartaDOS X system loader performs fixing up the 16-bit words, which the fixup records or XREF records, appended to the binary file, identify as 16-bit addresses (within 64k address space) to be fixed during loading. Loading blocks of code to 130XE-type or Axlon-type banked memory does not make any difference, because all that physically fits in the 64k address space.

Extending that scheme to the 24-bit address space requires introducing additional rules which allow the SpartaDOS X system loader to decide on the fly if the memory location to be fixed up is a 16-bit word or a 24-bit long word.

To explain that we have first to define two terms:

a) base block: it is the program's block which provides the base address for relocation. For example, if I have loaded some code at $002000 and want to reference it from another block (loaded somewhere else), $002000 will be the base address for my relocations: the relocating loader will add that address to any absolute reference (e.g. a JSR) I am making to that block.

b) target block: it is the program's block inside which the relocating loader will be fixing up the addresses. These locations may contain absolute references (e.g. a JSR) to the same block (internal references) as well as to other blocks (external references).

Now, as said already, the default size of all relocations is 16-bit: if I want to make a JSR from the „target" block to an address in the „base" block, the assembler will generate a JSR instruction with the argument relative to the beginning of the block the JSR is aiming at (= the base block); and the relocating loader, while the program is being loaded, will fix that word up by adding the base block load address to the argument of the JSR.

For the 6502's 64k address space, this is always the case. For the 65C816 24-bit address space the following additional rules apply:

a) if the base block is not the same as the target block, **and**

b) if the base block is loaded outside the first 64k (segment 0), **then**

c) the relocation size is 24-bit.

In pseudocode:

```
fixups = word
if base_block <> target_block
        b1 = base_block_address & $ff0000
        if b1 <> 0
             fixups = long
        endif
endif
```

This means that:

a) internal references to any block, whenever it is loaded, will be resolved as 16-bit words; this makes internal long word references possible only for blocks loaded to the first 64k (segment 0);

b) external references to any block loaded to the segment 0 may be both 16-bit and 24-bit; in 24-bit long words the most significant byte will be unchanged by the relocating loader;

c) external references to any block loaded outside the segment 0 must be 24-bit.

Appendix B: 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 C: 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.	LDA AA silently accepted as if it was 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 D: ELSA's source statistics

* Labels defined: ca 2200 (ca 50 KB)
* Source code lines: ca 16000
* Source code size: ca 192 KB
* Number of source files: 32
* Shortest source file: 70 bytes
* Longest source file: 30 KB

Time spent self-assembling	
Rapidus 20 MHz, Fast RD/WR, spinning platter HDD	26 sec.
Rapidus 20 MHz, ditto, Rapidus banked SDRAM RAM-disk	22 sec.
Antonia 1.77 MHz, case-sensitive mode	330 sec. (5 min. 30 s.)

Time spent creating 6000 labels, 11 characters each	
Rapidus 20 MHz, Fast RD/WR, case-sensitive mode (default)	87 sec. (1 min. 27 s.)
Rapidus 20 MHz, ditto, case-insensitive mode	93 sec. (1 min. 33 s.)
Altirra 21.28 MHz, case-sensitive mode	82 sec. (1 min. 22 s.)
Altirra 21.28 MHz, case-insensitive mode	88 sec. (1 min. 28 s.)
Antonia 1.77 MHz, case-sensitive mode	1089 sec. (18 min. 9 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

Time spent generating 32768 times LDA #$FF	
	sec.)

The code:

```
.rept 32768
lda #$ff
.endr
```

Time spent creating a 512k file filled with $FFs	
ELSA, Rapidus 20 MHz, Fast RD/WR, SpartaDOS X 4.49f, GR.0	72 sec. (1 min. 12 s.)
ELSA, Rapidus 20 MHz, Fast RD/WR, SpartaDOS X 4.49f, VBXE	69 sec. (1 min. 9 s.)
ELSA, Rapidus 40 MHz, Fast RD/WR, SpartaDOS X 4.49f, VBXE	47.04 sec.
MADS 2.1.3, Pentium III 1200 MHz, FreeBSD 8	30.89 sec.

The code for ELSA:

```
.out empty.rom
.opt h-
.rept 32768
.hex ff ff ff ff ff ff ff ff
.endr
.rept 32768
.hex ff ff ff ff ff ff ff ff
.endr
```

The roughly equivalent code for MADS:

```
.opt h-
.rept 32768
.byte $ff,$ff,$ff,$ff,$ff,$ff,$ff,$ff
.endr
.rept 32768
.byte $ff,$ff,$ff,$ff,$ff,$ff,$ff,$ff
.endr
```