
wla-dx Documentation

Release 9.12

vhelin

Mar 20, 2021

1	Assembler Directives	3
1.1	.16BIT	7
1.2	.24BIT	7
1.3	.8BIT	7
1.4	.ACCU 8	8
1.5	.ADDR 16000, main, 255	8
1.6	.ASC "HELLO WORLD!"	8
1.7	.ASCIITABLE	9
1.8	.ASCSTR "HELLO WORLD!", \$A	9
1.9	.ASCTABLE	9
1.10	.ASM	9
1.11	.BACKGROUND "parallax.gb"	10
1.12	.BANK 0 SLOT 1	10
1.13	.BASE \$80	10
1.14	.BLOCK "Block1"	10
1.15	.BREAKPOINT	11
1.16	.BR	11
1.17	.BYT 100, \$30, %1000, "HELLO WORLD!"	11
1.18	.CARTRIDGETYPE 1	11
1.19	.COMPUTEGBCHECKSUM	11
1.20	.COMPUTEGBCOMPLEMENTCHECK	11
1.21	.COMPUTESMSCHECKSUM	12
1.22	.COMPUTESNESCHECKSUM	12
1.23	.COUNTRYCODE 1	12
1.24	.DATA \$ff00, 2	12
1.25	.DB 100, \$30, %1000, "HELLO WORLD!"	13
1.26	.DBCOS 0.2, 10, 3.2, 120, 1.3	13
1.27	.DBM filtermacro 1, 2, "encrypt me"	13
1.28	.DBRND 20, 0, 10	13
1.29	.DBSIN 0.2, 10, 3.2, 120, 1.3	14
1.30	.DD \$1ffffff, \$2000000	14
1.31	.DDM filtermacro 1, 2, 3	14
1.32	.DEF IF \$FF0F	14
1.33	.DEFINE IF \$FF0F	14
1.34	.DESTINATIONCODE 1	15
1.35	.DL \$102030, \$405060	16

1.36	.DLM filtermacro 1, 2, 3	16
1.37	.DS 256, \$10	16
1.38	.DSB 256, \$10	16
1.39	.DSD 256, \$1ffffff	16
1.40	.DSL 16, \$102030	16
1.41	.DSTRUCT waterdrop INSTANCEOF water VALUES	16
1.42	.DSW 128, 20	18
1.43	.DW 16000, 10, 255	18
1.44	.DWCOS 0.2, 10, 3.2, 1024, 1.3	18
1.45	.DWM filtermacro 1, 2, 3	18
1.46	.DWRND 20, 0, 10	18
1.47	.DWSIN 0.2, 10, 3.2, 1024, 1.3	18
1.48	.ELSE	19
1.49	.EMPTYFILL \$C9	19
1.50	.ENDASM	19
1.51	.ENDA	19
1.52	.ENDB	19
1.53	.ENDEMUVECTOR	19
1.54	.ENDE	19
1.55	.ENDIF	20
1.56	.ENDME	20
1.57	.ENDM	20
1.58	.ENDNATIVEVECTOR	20
1.59	.ENDRO	20
1.60	.ENDR	20
1.61	.ENDSNES	20
1.62	.ENDST	21
1.63	.ENDS	21
1.64	.ENDU	21
1.65	.ENUM \$C000	21
1.66	.ENUMID ID_1 0	22
1.67	.EQU IF \$FF0F	23
1.68	.EXHIROM	23
1.69	.EXPORT work_x	23
1.70	.FAIL "THE EYE OF MORDOR HAS SEEN US!"	24
1.71	.FARADDR main, irq_1	24
1.72	.FASTROM	24
1.73	.FCLOSE FP_DATABIN	24
1.74	.FOPEN "data.bin" FP_DATABIN	24
1.75	.FREAD FP_DATABIN DATA	24
1.76	.FSIZE FP_DATABIN SIZE	25
1.77	.GBHEADER	25
1.78	.HEX "a0A0ffDE"	25
1.79	.HIROM	25
1.80	.IF DEBUG == 2	26
1.81	.IFDEF IF	26
1.82	.IFDEFM \2	26
1.83	.IFEQ DEBUG 2	26
1.84	.IFEXISTS "main.s"	26
1.85	.IFGR DEBUG 2	27
1.86	.IFGREQ DEBUG 2	27
1.87	.IFLE DEBUG 2	27
1.88	.IFLEEQ DEBUG 2	27
1.89	.IFNDEF IF	27

1.90	.IFNDEFM \2	27
1.91	.IFNEQ DEBUG 2	28
1.92	.INC "cgb_hardware.i"	28
1.93	.INCBIN "sorority.bin"	28
1.94	.INCDIR "/usr/programming/gb/include/"	29
1.95	.INCLUDE "cgb_hardware.i"	29
1.96	.INDEX 8	30
1.97	.INPUT NAME	30
1.98	.LICENSEECODENEW "1A"	30
1.99	.LICENSEECODEOLD \$1A	31
1.100	.LONG \$102030, \$405060	31
1.101	.LOROM	31
1.102	.MACRO TEST	31
1.103	.MEMORYMAP	33
1.104	.NAME "NAME OF THE ROM"	35
1.105	.NEXTU name	35
1.106	.NINTENDOLOGO	35
1.107	.NOWDC	35
1.108	.ORG \$150	35
1.109	.ORGA \$150	36
1.110	.OUTNAME "other.o"	36
1.111	.PRINT "Numbers 1 and 10: ", DEC 1, " \$", HEX 10, "\n"	36
1.112	.PRINTT "Here we are...\n"	37
1.113	.PRINTV DEC DEBUG+1	37
1.114	.RAMSECTION "Vars" BANK 0 SLOT 1 ALIGN 256 OFFSET 32	37
1.115	.RAMSIZE 0	39
1.116	.REDEF IF \$0F	40
1.117	.REDEFINE IF \$0F	40
1.118	.REPEAT 6	40
1.119	.REPT 6	40
1.120	.ROMBANKMAP	40
1.121	.ROMBANKS 2	41
1.122	.ROMBANKSIZE \$4000	41
1.123	.ROMDMG	41
1.124	.ROMGBCONLY	42
1.125	.ROMGBC	42
1.126	.ROMSGB	42
1.127	.ROW \$ff00, 1, "3"	42
1.128	.SDSCTAG 1.0, "DUNGEON MAN", "A wild dungeon exploration game", "Ville Helin"	42
1.129	.SECTION "Init" FORCE	43
1.130	.SEED 123	45
1.131	.SHIFT	45
1.132	.SLOT 1	45
1.133	.SLOWROM	46
1.134	.SMC	46
1.135	.SMSHEADER	46
1.136	.SMSTAG	47
1.137	.SNESEMUVECTOR	47
1.138	.SNESHEADER	47
1.139	.SNESNATIVEVECTOR	48
1.140	.STRINGMAP script "Hello\n"	49
1.141	.STRINGMAPTABLE script "script.tbl"	49
1.142	.STRUCT enemy_object	50

1.143	.SYM SAUSAGE	51
1.144	.SYMBOL SAUSAGE	51
1.145	.TABLE byte, word, byte	51
1.146	.UNBACKGROUND \$1000 \$1FFF	51
1.147	.UNDEF DEBUG	52
1.148	.UNDEFINE DEBUG	52
1.149	.UNION name	52
1.150	.VERSION 1	53
1.151	.WDC	53
1.152	.WORD 16000, 10, 255	53
2	Assembler Syntax	55
2.1	Case Sensitivity	55
2.2	Comments	55
2.3	Line splitting	55
2.4	Labels	56
2.5	Number Types	58
2.6	Strings	58
2.7	Mnemonics	59
2.8	Brackets?	59
3	Error Messages	61
4	Supported ROM/RAM/Cartridge Types (WLA-GB)	63
4.1	ROM Size	63
4.2	RAM Size	63
4.3	Cartridge Type	64
5	Bugs	65
6	Files	67
6.1	tests	67
6.2	tests/gb-z80/lib	67
6.3	memorymaps	67
7	Temporary Files	69
8	Compiling	71
8.1	Compiling Object Files	71
8.2	Compiling Library Files	72
9	Linking	73
10	Arithmetics	77
11	Binary to DB Conversion	79
12	Things you should know about coding for...	81
12.1	Z80	81
12.2	6502	81
12.3	65C02	82
12.4	65CE02	82
12.5	6510	83
12.6	65816	83
12.7	HUC6280	84
12.8	SPC-700	84

12.9	Pocket Voice (GB-Z80)	85
12.10	GB-Z80	85
13	WLA Flags	87
14	Extra compile time definitions	89
15	Good things to know about WLA	91
16	WLA DX's architectural overview	93
16.1	WLA	93
16.2	WLALINK	94
17	WLA Symbols	95
17.1	Information For Emulator Developers	95
18	Legal Note	99
19	Manpage: WLA-LINK	101
19.1	SYNOPSIS	101
19.2	OPTIONS	101
19.3	DESCRIPTION	102
19.4	EXAMPLES	103
20	Manpage: WLA-CPU	105
20.1	SYNOPSIS	105
20.2	OPTIONS	105
20.3	DESCRIPTION	106
20.4	EXAMPLES	106
21	Manpage: WLAB	107
21.1	SYNOPSIS	107
21.2	OPTIONS	107
21.3	DESCRIPTION	107
21.4	EXAMPLES	107

The history behind WLA DX, from the original author, Ville Helin:

I wrote this because I had never written an assembler before and I really needed a macro assembler which could compile the GB-Z80 code I wrote. ;) Gaelan Griffin needed real Z80 support for his SMS projects so I thought I could write WLA to be a little more open and nowadays it supports all the Z80 systems you can think of. You'll just have to define the memorymap of the destination machine for your project. After fixing some bugs I thought I could add support for 6502 systems so all NES-people would get their share of WLA as well. After finishing that few people said they'd like 65816 support (they had SNES developing in mind) so I added support for that. And then I thought I should write a 6510 version of WLA as well...

This is my ideal GB-Z80 macro assembler (not in final form, not yet). ;) Tastes differ. Thus WLA! Notice that WLA was initially made for Game Boy developers so the GB-Z80 version and the rest differ a little.

Good to know about WLA DX:

Almost all rules that apply to Z80 source code processing with WLA DX apply also to 6502, 65C02, 65CE02, 6510, 65816, 6800, 6801, 6809, 8008, 8080, HUC6280 and SPC-700.

About the names: WLA DX means all the tools covered in this documentation. So WLA DX includes WLA GB-Z80/Z80/6502/65C02/65CE02/6510/65816/6800/6801/6809/8008/8080/HUC6280/SPC-700 macro assembler (what a horribly long name), WLAB, and WLALINK GB-Z80/Z80/6502/65C02/65CE02/6510/65816/6800/6801/6809/8008/8080/ HUC6280/SPC-700 linker. We use plain WLA to refer to the macro assembler in this document.

There was WLAD, an GB-Z80 dissassembler, but it has been discontinued and removed from the project and the documentation.

Currently WLA can also be used as a patch tool. Just include the original ROM image into the project with `.BACKGROUND` and insert e.g., `OVERWRITE .SECTION s` to patch the desired areas. Output the data into a new ROM image and there you have it. 100% readable (asm coded) patches are reality!

Note that you can directly compile only object and library files. You must use WLALINK to link these (or only one, if you must) into a ROM/program file.

WLA DX's old homepage: <http://www.villehelin.com/wla.html>

WLA DX's new homepage: <https://github.com/vhelin/wla-dx>

CHAPTER 1

Assembler Directives

Here's the order in which the data is placed into the output:

1. Data and group 3 directives outside sections.
2. Group 2 directives.
3. Data and group 3 directives inside sections.
4. Group 1 directives.

ALL	All, GB-Z80, Z80, 6502, 65C02, 65CE02, 6510, 65816, HUC6280, SPC-700, 6800, 6801, 6809, 8008 and 8080 versions apply.
GB	Only the GB-Z80 version applies.
GB8	Only the GB-Z80 and 65816 versions apply.
Z80	Only the Z80 version applies.
658	Only the 65816 version applies.
680	Only the 6800, 6801 and 6809 versions apply.
800	Only the 8008 version applies.
808	Only the 8080 version applies.
SPC	Only the SPC-700 version applies.
65x	Only the 6502, 65C02, 65CE02, 6510, 65816 and HUC6280 versions apply.
!GB	Only the Z80, 6502, 65C02, 65CE02, 6510, 65816, HUC6280 and SPC-700 versions apply.

Group 1:

GB	.COMPUTEGBCHECKSUM
Z80	.COMPUTESMSCHECKSUM
658	.COMPUTESNESCHECKSUM
Z80	.SDSCTAG 1.0, "DUNGEON MAN", "A wild dungeon exploration game", "Ville Helin"
Z80	.SMSTAG

Group 2:

GB	.CARTRIDGETYPE 1
GB	.COMPUTEGBCOMPLEMENTCHECK
GB	.COUNTRYCODE 1
GB	.DESTINATIONCODE 1
ALL	.EMPTYFILL \$C9
658	.ENDEMUVECTOR
658	.ENDNATIVEVECTOR
658	.ENDSNES
658	.EXHIROM
ALL	.EXPORT work_x
658	.FASTROM
GB	.GBHEADER
658	.HIROM
GB	.LICENSEECODENEW "1A"
GB	.LICENSEECODEOLD \$1A
658	.LOROM
GB8	.NAME "NAME OF THE ROM"
GB	.NINTENDOLOGO
ALL	.OUTNAME "other.o"
GB	.RAMSIZE 0
GB	.ROMDMG
GB	.ROMGBC
GB	.ROMGBCONLY
GB	.ROMSGB
658	.SLOWROM
658	.SMC
Z80	.SMSHEADER
658	.SNESEMUVECTOR
658	.SNESHEADER
658	.SNESNATIVEVECTOR
GB	.VERSION 1

Group 3:

65x	.16BIT
658	.24BIT
65x	.8BIT
658	.ACCU 8
ALL	.ADDR 16000, main, 255
ALL	.ASC "HELLO WORLD!"
ALL	.ASCIITABLE
ALL	.ASCSTR "HELLO WORLD!", \$A
ALL	.ASCTABLE
ALL	.ASM
ALL	.BACKGROUND "parallax.gb"
ALL	.BANK 0 SLOT 1
ALL	.BASE \$80
ALL	.BLOCK "Block1"
ALL	.BR
ALL	.BREAKPOINT

Continued on next page

Table 2 – continued from previous page

ALL	.BYT 100, \$30, %1000, "HELLO WORLD!"
ALL	.DATA \$ff00, 2
ALL	.DB 100, \$30, %1000, "HELLO WORLD!"
ALL	.DBCOS 0.2, 10, 3.2, 120, 1.3
ALL	.DBM filtermacro 1, 2, "encrypt me"
ALL	.DBRND 20, 0, 10
ALL	.DBSIN 0.2, 10, 3.2, 120, 1.3
ALL	.DD \$1ffffff, \$2000000
ALL	.DDM filtermacro 1, 2, 3
ALL	.DEF IF \$FF0F
ALL	.DEFINE IF \$FF0F
ALL	.DL \$102030, \$405060
ALL	.DLM filtermacro 1, 2, 3
ALL	.DS 256, \$10
ALL	.DSB 256, \$10
ALL	.DSD 256, \$1ffffff
ALL	.DSL 16, \$102030
ALL	.DSTRUCT waterdrop INSTANCEOF water DATA "tingle", 40, 120
ALL	.DSW 128, 20
ALL	.DW 16000, 10, 255
ALL	.DWCOS 0.2, 10, 3.2, 1024, 1.3
ALL	.DWM filtermacro 1, 2, 3
ALL	.DWRND 20, 0, 10
ALL	.DWSIN 0.2, 10, 3.2, 1024, 1.3
ALL	.ELSE
ALL	.ENDA
ALL	.ENDASM
ALL	.ENDB
ALL	.ENDE
ALL	.ENDIF
ALL	.ENDM
ALL	.ENDME
ALL	.ENDR
ALL	.ENDRO
ALL	.ENDS
ALL	.ENDST
ALL	.ENDU
ALL	.ENUM \$C000
ALL	.ENUMID ID_1 0
ALL	.EQU IF \$FF0F
ALL	.FAIL "THE EYE OF MORDOR HAS SEEN US!"
658	.FARADDR main, irq_1
ALL	.FCLOSE FP_DATABIN
ALL	.FOPEN "data.bin" FP_DATABIN
ALL	.FREAD FP_DATABIN DATA
ALL	.FSIZE FP_DATABIN SIZE
ALL	.HEX "a0A0ffDE"
ALL	.IF DEBUG == 2
ALL	.IFDEF IF
ALL	.IFDEFM \2

Continued on next page

Table 2 – continued from previous page

ALL	.IFEQ DEBUG 2
ALL	.IFEXISTS "main.s"
ALL	.IFGR DEBUG 2
ALL	.IFGREQ DEBUG 1
ALL	.IFLE DEBUG 2
ALL	.IFLEEQ DEBUG 1
ALL	.IFNDEF IF
ALL	.IFNDEFM \2
ALL	.IFNEQ DEBUG 2
ALL	.INC "cgb_hardware.i"
ALL	.INCBIN "sorority.bin"
ALL	.INCDIR "/usr/programming/gb/include/"
ALL	.INCLUDE "cgb_hardware.i"
658	.INDEX 8
ALL	.INPUT NAME
658	.LONG \$102030, \$405060
ALL	.MACRO TEST
ALL	.MEMORYMAP
ALL	.NEXTU name
658	.NOWDC
ALL	.ORG \$150
ALL	.ORGA \$150
ALL	.PRINT "Numbers 1 and 10: ", DEC 1, " \$", HEX 10, "\n"
ALL	.PRINTT "Here we are...\n"
ALL	.PRINTV DEC DEBUG+1
ALL	.RAMSECTION "Vars" BANK 0 SLOT 1 ALIGN 256 OFFSET 32
ALL	.REDEF IF \$F
ALL	.REDEFINE IF \$F
ALL	.REPEAT 6
ALL	.REPT 6
ALL	.ROMBANKMAP
ALL	.ROMBANKS 2
ALL	.ROMBANKSIZE \$4000
ALL	.ROW \$ff00, 1, "3"
ALL	.SECTION "Init" FORCE
ALL	.SEED 123
ALL	.SHIFT
ALL	.SLOT 1
ALL	.STRINGMAP script "Hello\n"
ALL	.STRINGMAPTABLE script "script.tbl"
ALL	.STRUCT enemy_object
ALL	.SYM SAUSAGE
ALL	.SYMBOL SAUSAGE
ALL	.TABLE byte, word, byte
ALL	.UNBACKGROUND \$1000 \$1FFF
ALL	.UNDEF DEBUG
ALL	.UNDEFINE DEBUG
ALL	.UNION name
658	.WDC
ALL	.WORD 16000, 10, 255

Descriptions:

1.1 .16BIT

Analogous to .8BIT. .16BIT forces all addresses and immediate values to be expanded into 16-bit range, when possible, that is:

```
LSR 11      ; $46 $0B
```

That would be the case, normally, but after .16BIT it becomes:

```
LSR 11      ; $4E $0B $00
```

This is not a compulsory directive.

1.2 .24BIT

Analogous to .8BIT and .16BIT. .24BIT forces all addresses to be expanded into 24-bit range, when possible, that is:

```
AND $11     ; $25 $11
```

That would be the case, normally, but after .24BIT it becomes:

```
AND $11     ; $2F $11 $00 $00
```

If it is not possible to expand the address into .24BIT range, then WLA tries to expand it into 16-bit range.

This is not a compulsory directive.

1.3 .8BIT

There are a few mnemonics that look identical, but take different sized arguments. Here's a list of such 6502 mnemonics:

ADC, AND, ASL, BIT, CMP, CPX, CPY, DEC, EOR, INC, LDA, LDX, LDY, ORA, ROL, SBC, STA, STX and STY.

For example:

```
LSR 11      ; $46 $0B
LSR $A000   ; $4E $00 $A0
```

The first one could also be:

```
LSR 11      ; $4E $0B $00
```

.8BIT is here to help WLA to decide to choose which one of the opcodes it selects. When you give .8BIT (default) no 8-bit address/value is expanded to 16-bits.

By default WLA uses the smallest possible size. This is true also when WLA finds a computation it can't solve right away. WLA assumes the result will be inside the smallest possible bounds, which depends on the type of the mnemonic.

You can also use the fixed argument size versions of such mnemonics by giving the size with the operand (i.e., operand hinting). Here are few examples:

```
LSR 11.B    ; $46 $0B
LSR 11.W    ; $4E $0B $00
```

In WLA-65816 `.ACCU/.INDEX/SEP/REP` override `.8BIT/.16BIT/.24BIT` when considering the immediate values, so be careful. Still, operand hints override all of these, so use them to be sure.

This is not a compulsory directive.

1.4 `.ACCU 8`

Forces WLA to override the accumulator size given with `SEP/REP`. `.ACCU` doesn't produce any code, it only affects the way WLA interprets the immediate values (8 for 8 bit operands, 16 for 16 bit operands) for opcodes dealing with the accumulator.

So after giving `.ACCU 8`:

```
AND #6
```

will produce `$29 $06`, and after giving `.ACCU 16`:

```
AND #6
```

will yield `$29 $00 $06`.

Note that `SEP/REP` again will in turn reset the accumulator/index register size.

This is not a compulsory directive.

1.5 `.ADDR 16000, main, 255`

`.ADDR` is an alias for `.DW`.

This is not a compulsory directive.

1.6 `.ASC "HELLO WORLD!"`

`.ASC` is an alias for `.DB`, but if you use `.ASC` it will remap the characters using the mapping given via `.ASCIITABLE`.

You can also use `ASC('?')` to map individual characters in the code

```
.DB ASC('A'), ASC('B')
```

and

```
LD A, ASC('A')
```

This is not a compulsory directive.

1.7 .ASCIITABLE

.ASCIITABLE's only purpose is to provide character mapping for .ASC and ASC ('?'). Take a look at the example:

```
.ASCIITABLE
MAP "A" TO "Z" = 0
MAP "!" = 90
.ENDA
```

Here we set such a mapping that character A is equal to 0, B is equal to 1, C is equal to 2, and so on, and ! is equal to 90.

After you've given the .ASCIITABLE, use .ASC to define bytes using this mapping (.ASC is an alias for .DB, but with .ASCIITABLE mapping). For example, .ASC "ABZ" would define bytes 0, 1 and 25 in our previous example.

Note that the following works as well:

```
.ASCIITABLE
MAP 'A' TO 'Z' = 0
MAP 65 = 90 ; 65 is the decimal for ASCII 'A'
.ENDA
```

Also note that the characters that are not given any mapping in .ASCIITABLE map to themselves (i.e., 2 maps to 2 in our previous example, etc.).

This is not a compulsory directive.

1.8 .ASCSTR "HELLO WORLD!", \$A

.ASCSTR is the same as .ASC, but it maps only supplied strings. All given bytes are not touched.:

```
.ASCSTR "HELLO WORLD!", $A
```

In this example the string "HELLO WORLD!" is mapped using the mapping given via .ASCIITABLE, but the last byte \$A is left as it is.

This is not a compulsory directive.

1.9 .ASCTABLE

.ASCTABLE is an alias for .ASCIITABLE.

This is not a compulsory directive.

1.10 .ASM

Tells WLA to start assembling. Use .ASM to continue the work which has been disabled with .ENDASM. .ASM and .ENDASM can be used to mask away big blocks of code. This is analogous to the ANSI C -comments (/ * . . . */), but .ASM and .ENDASM can be nested, unlike the ANSI C -counterpart.

This is not a compulsory directive.

1.11 .BACKGROUND "parallax.gb"

This chooses an existing ROM image (`parallax.gb` in this case) as a background data for the project. You can overwrite the data with `OVERWRITE` sections only, unless you first clear memory blocks with `.UNBACKGROUND` after which there's room for other sections as well.

Note that `.BACKGROUND` can be used only when compiling an object file.

`.BACKGROUND` is useful if you wish to patch an existing ROM image with new code or data.

This is not a compulsory directive.

1.12 .BANK 0 SLOT 1

Defines the ROM bank and the slot it is inserted into in the memory. You can also type the following:

```
.BANK 0
```

This tells WLA to move into BANK 0 which will be put into the `DEFAULTSLOT` of `.MEMORYMAP`.

Every time you use `.BANK`, supply `.ORG/` `.ORGA` as well, just to make sure WLA calculates addresses correctly.

This is a compulsory directive.

1.13 .BASE \$80

Defines the base value for the bank number (used only in 24-bit addresses and when getting a label's bank number with `:`). Here are few examples of how to use `.BASE` (both examples assume the label resides in the first ROM bank):

```
.BASE $00
label1:
.BASE $80
label2:

    JSL label1    ; if label1 address is $1234, this will assemble into
                  ; JSL $001234
    JSL label2    ; label2 is also $1234, but this time the result will be
                  ; JSL $801234
```

`.BASE` defaults to `$00`. Note that the address of the label will also contribute to the bank number (bank number == `.BASE` + ROM bank of the label).

On 65816, use `.LOROM`, `.HIROM` or `.EXHIROM` to define the ROM mode.

This is not a compulsory directive.

1.14 .BLOCK "Block1"

Begins a block (called `Block1` in the example). These blocks have only one function: to display the number of bytes they contain. When you embed such a block into your code, WLA displays its size when it assembles the source file.

Use `.ENDB` to terminate a `.BLOCK`. Note that you can nest `.BLOCK` s.

This is not a compulsory directive.

1.15 .BREAKPOINT

.BREAKPOINT is an alias for .BR.

This is not a compulsory directive.

1.16 .BR

Inserts a breakpoint that behaves like a .SYM without a name. Breakpoints can only be seen in WLALINK's symbol file.

This is not a compulsory directive.

1.17 .BYT 100, \$30, %1000, "HELLO WORLD!"

.BYT is an alias for .DB.

This is not a compulsory directive.

1.18 .CARTRIDGETYPE 1

Indicates the type of the cartridge (mapper and so on). This is a standard Gameboy cartridge type indicator value found at \$147 in a Gameboy ROM, and there this one is put to also.

This is not a compulsory directive.

1.19 .COMPUTEGBCHECKSUM

When this directive is used WLA computes the ROM checksum found at \$14E and \$14F in a Gameboy ROM. Note that this directive can only be used with WLA-GB.

Note that you can also write .COMPUTECHECKSUM (the old name for this directive), but it's not recommended.

This is not a compulsory directive.

1.20 .COMPUTEGBCOMPLEMENTCHECK

When this directive is used WLA computes the ROM complement check found at \$14D in a Gameboy ROM.

Note that you can still use .COMTECOMPLEMENTCHECK (the old name for this directive), but it's not recommended.

This is not a compulsory directive.

1.21 .COMPUTESMSCHECKSUM

When this directive is used WLA computes the ROM checksum found at \$7FFA and \$7FFB (or \$3FFA - \$3FFB is the ROM is 16KBs, or \$1FFA - \$1FFB for 8KB ROMs) in a SMS/GG ROM. Note that this directive can only be used with WLA-z80. Also note that the ROM size must be at least 8KBs. The checksum is calculated using bytes 0x0000 - 0x1FEF/0x3FEF/0x7FEF.

This is not a compulsory directive.

1.22 .COMPUTESNESCHECKSUM

When this directive is used WLA computes the SNES ROM checksum and inverse checksum found at \$7FDC - \$7FDF (LoROM), \$FFDC - \$FFDF (HiROM) or \$40FFDC - \$40FFDF and \$FFDC - \$FFDF (ExHiROM). Note that this directive can only be used with WLA-65816. Also note that the ROM size must be at least 32KB for LoROM images, 64KB for HiROM images and 32.5MBit for ExHiROM.

.LOROM, .HIROM or .EXHIROM must be issued before .COMPUTESNESCHECKSUM.

This is not a compulsory directive.

1.23 .COUNTRYCODE 1

Indicates the country code located at \$14A of a Gameboy ROM.

This is not a compulsory directive.

1.24 .DATA \$ff00, 2

Defines bytes after a .TABLE has been used to define the format. An alternative way of defining bytes to .DB/.DW.

Note that when you use .DATA you can give as many items .TABLE defines. The next time you'll use .DATA you'll continue from the point the previous .DATA ended.

Examples:

```
.TABLE dsw 2, dsb 2
```

This defines two rows worth of bytes:

```
.DATA $ff00, $aabb, $10, $20, $1020, $3040, $50, $60
```

This does the same:

```
.DATA $ff00, $aabb  
.DATA $10, $20  
.DATA $1020, $3040  
.DATA $50, $60
```

This is not a compulsory directive.

1.25 .DB 100, \$30, %1000, "HELLO WORLD!"

Defines bytes.

This is not a compulsory directive.

1.26 .DBCOS 0.2, 10, 3.2, 120, 1.3

Defines bytes just like .DSB does, only this time they are filled with cosine data. .DBCOS takes five arguments.

The first argument is the starting angle. Angle value ranges from 0 to 359.999..., but you can supply WLA with values that are out of the range - WLA fixes them ok. The value can be integer or float.

The second argument describes the amount of additional angles. The example will define 11 angles.

The third argument is the adder value which is added to the angle value when next angle is calculated. The value can be integer or float.

The fourth and fifth arguments can be seen from the pseudo code below, which also describes how .DBCOS works. The values can be integer or float.

Remember that `cos` (and `sin`) here returns values ranging from -1 to 1:

```
.DBCOS A, B, C, D, E

for (B++; B > 0; B--) {
    output_data((D * cos(A)) + E)
    A = keep_in_range(A + C)
}
```

This is not a compulsory directive.

1.27 .DBM filtermacro 1, 2, "encrypt me"

Defines bytes using a filter macro. All the data is passed to `filtermacro` in the first argument, one byte at a time, and the byte that actually gets defined is the value of definition `_OUT` (`_out` works as well). The second macro argument holds the offset from the beginning (the first byte) in bytes (the series being 0, 1, 2, 3, ...).

Here's an example of a filter macro that increments all the bytes by one:

```
.macro increment
.redefine _out \1+1
.endm
```

This is not a compulsory directive.

1.28 .DBRND 20, 0, 10

Defines bytes, just like .DSB does, only this time they are filled with (pseudo) random numbers. We use the integrated Mersenne Twister to generate the random numbers. If you want to seed the random number generator, use `.SEED`.

The first parameter (20 in the example) defines the number of random numbers we want to generate. The next two tell the range of the random numbers, i.e. min and max.

Here's how it works:

```
.DBRND A, B, C

for (i = 0; i < A; i++)
    output_data((rand() % (C-B+1)) + B);
```

This is not a compulsory directive.

1.29 .DBSIN 0.2, 10, 3.2, 120, 1.3

Analogous to .DBCOS, but does `sin()` instead of `cos()`.

This is not a compulsory directive.

1.30 .DD \$1fffffff, \$2000000

Defines double words (four bytes each). .DD takes only numbers, labels and characters as input, not strings.

This is not a compulsory directive.

1.31 .DDM filtermacro 1, 2, 3

Defines 32-bit words using a filter macro. Works just like .DBM, .DWM and .DLM.

This is not a compulsory directive.

1.32 .DEF IF \$FF0F

.DEF is an alias for .DEFINE.

This is not a compulsory directive.

1.33 .DEFINE IF \$FF0F

Assigns a number or a string to a definition label.

By default all defines are local to the file where they are presented. If you want to make the definition visible to all the files in the project, use .EXPORT or add EXPORT to the end of .DEFINE:

```
.DEFINE ID_0 0 EXPORT
```

WARNING: Please declare your definition lexically before using it as otherwise the assembler might make incorrect assumptions about its value and size and choose e.g. wrong opcodes and generate binary that doesn't run properly.

Here are some examples:

```
.DEFINE X 1000
.DEFINE FILE "level01.bin"
.DEFINE TXT1 "hello and welcome", 1, "to a new world...", 0
.DEFINE BYTES 1, 2, 3, 4, 5
.DEFINE COMPUTATION X+1
.DEFINE DEFAULTV
```

All definitions with multiple values are marked as data strings, and `.DB` is about the only place where you can later on use them:

```
.DEFINE BYTES 1, 2, 3, 4, 5
.DB 0, BYTES, 6
```

is the same as:

```
.DB 0, 1, 2, 3, 4, 5, 6
```

If you omit the definition value (in our example `DEFAULTV`), WLA will default to 0.

Note that you must do your definition before you use it, otherwise WLA will use the final value of the definition. Here's an example of this:

```
.DEFINE AAA 10
.DB AAA          ; will be 10.
.REDEFINE AAA 11
```

but:

```
.DB AAA          ; will be 11.
.DEFINE AAA 10
.REDEFINE AAA 11
```

You can also create definitions on the command line. Here's an example of this:

```
wla-gb -v1 -DMOON -DNAME=john -DPRICE=100 -DADDRESS=$100 math.s
```

`MOON`'s value will be 0, `NAME` is a string definition with value `john`, `PRICE`'s value will be 100, and `ADDRESS`'s value will be \$100.

Note that:

```
.DEFINE AAA = 10 ; the same as ".DEFINE AAA 10".
```

works as well. And this works also:

```
AAA = 10
```

This is not a compulsory directive.

1.34 .DESTINATIONCODE 1

`.DESTINATIONCODE` is an alias for `.COUNTRYCODE`.

This is not a compulsory directive.

1.35 `.DL $102030, $405060`

Defines long words (three bytes each). `.DL` takes only numbers, labels and characters as input, not strings.

This is not a compulsory directive.

1.36 `.DLM filtermacro 1, 2, 3`

Defines 24-bit words using a filter macro. Works just like `.DBM`, `.DWM` and `.DDM`.

This is not a compulsory directive.

1.37 `.DS 256, $10`

`.DS` is an alias for `.DSB`.

This is not a compulsory directive.

1.38 `.DSB 256, $10`

Defines 256 bytes of `$10`.

This is not a compulsory directive.

1.39 `.DSD 256, $1fffffff`

Defines 256 double words (four bytes) of `$1fffffff`.

This is not a compulsory directive.

1.40 `.DSL 16, $102030`

Defines 16 long words (three bytes) of `$102030`.

This is not a compulsory directive.

1.41 `.DSTRUCT waterdrop INSTANCEOF water VALUES`

Defines an instance of struct `water`, called `waterdrop`, and fills it with the given data. Before calling `.DSTRUCT` we must have defined the structure, and in this example it could be like:

```
.STRUCT water
    name    ds 8
    age     db
    weight  dw
.ENDST
```


There are two syntaxes for `.DSTRUCT`; the new and legacy versions. To use the new syntax, put the keyword “VALUES” at the end of the first line. The old syntax uses the keyword “DATA” or none at all.

The new syntax looks like this:

```
.DSTRUCT waterdrop INSTANCEOF water VALUES
    name:    .db "tingle"
    age:     .db 40
    weight:  .dw 120
.ENDST
```

The fields can be put in any order. Any omitted fields are set to the `.EMPTYFILL` value (\$00 by default). Any data-defining directive can be used within `.DSTRUCT`, as long as it does not exceed the size of the data it is being defined for. The only exception is `.DSTRUCT` itself, which cannot be nested.

The old syntax looks like this:

```
.DSTRUCT waterdrop INSTANCEOF water DATA "tingle", 40, 120
```

The `DATA` and `INSTANCEOF` keywords are optional. This will assign data for each field of the struct in the order they were defined.

In either example you would get the following labels:

```
waterdrop
waterdrop.name
waterdrop.age
waterdrop.weight
_sizeof_waterdrop      = 11
_sizeof_waterdrop.name = 8
_sizeof_waterdrop.age  = 1
_sizeof_waterdrop.weight = 2
```

The legacy syntax does not support unions; it will give an error if you attempt to define data for a union.

For the new syntax, nested structs are supported like so (assume the `water` struct is also defined:

```
.STRUCT drop_pair
    waterdrops: instanceof water 2
.ENDST

.DSTRUCT drops INSTANCEOF drop_pair VALUES
    waterdrops.1:    .db "qwertyui" 40
                    .dw 120
    waterdrops.2.name: .db "tingle"
    waterdrops.2.age:  .db 40
    waterdrops.2.weight: .dw 12
.ENDST
```

In this case, the properties of `waterdrops.1` were defined implicitly; 8 bytes for the name, followed by a byte for the age, followed by a word for the weight. The values for `waterdrops.2` were defined in a more clear way.

In this case, `waterdrops` and `waterdrops.1` are equivalent. `waterdrops.1.name` is different, even though its address is the same, because it has a size of 8. If you attempted to do this:

```
.DSTRUCT drops INSTANCEOF drop_pair VALUES
    waterdrops.1.name: .db "qwertyui" 40
                    .dw 120
.ENDST
```

It would fail, because only the 8 name bytes are available to be defined in this context, as opposed to the 11 bytes for the entire `waterdrops.1` structure.

Named unions can be assigned to in a similar way, by writing its full name with a `.` separating the union name and the field name.

The struct can be defined namelessly, in which case no labels will be generated, like so:

```
.DSTRUCT INSTANCEOF drop_pair VALUES
    * * *
.ENDST
```

This is not a compulsory directive.

1.42 `.DSW 128, 20`

Defines 128 words (two bytes) of 20.

This is not a compulsory directive.

1.43 `.DW 16000, 10, 255`

Defines words (two bytes each). `.DW` takes only numbers, labels and characters as input, not strings.

This is not a compulsory directive.

1.44 `.DWCOS 0.2, 10, 3.2, 1024, 1.3`

Analogous to `.DBCOS` (but defines 16-bit words).

This is not a compulsory directive.

1.45 `.DWM filtermacro 1, 2, 3`

Defines 16-bit words using a filter macro. Works just like `.DBM`, `.DLM` and `.DDM`.

This is not a compulsory directive.

1.46 `.DWRND 20, 0, 10`

Analogous to `.DBRND` (but defines words).

This is not a compulsory directive.

1.47 `.DWSIN 0.2, 10, 3.2, 1024, 1.3`

Analogous to `.DBCOS` (but defines 16-bit words and does `sin()` instead of `cos()`).

This is not a compulsory directive.

1.48 .ELSE

If the previous `.IFxxx` failed then the following text until `.ENDIF` is acknowledged.

This is not a compulsory directive.

1.49 .EMPTYFILL \$C9

This byte is used in filling the unused areas of the ROM file. `EMPTYFILL` defaults to `$00`.

This is not a compulsory directive.

1.50 .ENDASM

Tells WLA to stop assembling. Use `.ASM` to continue the work.

This is not a compulsory directive.

1.51 .ENDA

Ends the ASCII table.

This is not a compulsory directive, but when `.ASCIITABLE` or `.ASCTABLE` are used this one is required to terminate them.

1.52 .ENDB

Terminates `.BLOCK`.

This is not a compulsory directive, but when `.BLOCK` is used this one is required to terminate it.

1.53 .ENDEMUVECTOR

Ends definition of the emulation mode interrupt vector table.

This is not a compulsory directive, but when `.SNESEMUVECTOR` is used this one is required to terminate it.

1.54 .ENDE

Ends the enumeration.

This is not a compulsory directive, but when `.ENUM` is used this one is required to terminate it.

1.55 .ENDIF

This terminates any `.IFxxx` directive.

This is not a compulsory directive, but if you use any `.IFxxx` then you need also to apply this.

1.56 .ENDME

Terminates `.MEMORYMAP`.

This is not a compulsory directive, but when `.MEMORYMAP` is used this one is required to terminate it.

1.57 .ENDM

Ends a `.MACRO`.

This is not a compulsory directive, but when `.MACRO` is used this one is required to terminate it.

1.58 .ENDNATIVEVECTOR

Ends definition of the native mode interrupt vector table.

This is not a compulsory directive, but when `.SNESNATIVEVECTOR` is used this one is required to terminate it.

1.59 .ENDRO

Ends the rom bank map.

This is not a compulsory directive, but when `.ROMBANKMAP` is used this one is required to terminate it.

1.60 .ENDR

Ends the repetition.

This is not a compulsory directive, but when `.REPEAT` is used this one is required to terminate it.

1.61 .ENDSNES

This ends the SNES header definition.

This is not a compulsory directive, but when `.SNESHEADER` is used this one is required to terminate it.

1.62 .ENDST

Ends the structure definition.

This is not a compulsory directive, but when `.STRUCT` is used this one is required to terminate it.

1.63 .ENDS

Ends the section.

This is not a compulsory directive, but when `.SECTION` or `.RAMSECTION` is used this one is required to terminate it.

1.64 .ENDU

Ends the current union.

1.65 .ENUM \$C000

Starts enumeration from `$C000`. Very useful for defining variables.

To start a descending enumeration, put `DESC` after the starting value. WLA defaults to `ASC` (ascending enumeration).

You can also add `EXPORT` after these if you want to export all the generated definitions automatically.

Here's an example of `.ENUM`:

```
...
.STRUCT mon                ; check out the documentation on
name ds 2                  ; .STRUCT
age db
.ENDST

.ENUM $A000
_scroll_x DB               ; db - define byte (byt and byte work also)
_scroll_y DB
player_x: DW               ; dw - define word (word works also)
player_y: DW
map_01: DS 16              ; ds - define size (bytes)
map_02 DSB 16              ; dsb - define size (bytes)
map_03 DSW 8               ; dsw - define size (words)
monster INSTANCEOF mon 3   ; three instances of structure mon
dragon INSTANCEOF mon      ; one mon
.ENDE
...
```

Previous example transforms into following definitions:

```
.DEFINE _scroll_x    $A000
.DEFINE _scroll_y    $A001
.DEFINE player_x     $A002
.DEFINE player_y     $A004
```

(continues on next page)

(continued from previous page)

```
.DEFINE map_01      $A006
.DEFINE map_02      $A016
.DEFINE map_03      $A026
.DEFINE monster     $A036
.DEFINE monster.1    $A036
.DEFINE monster.1.name $A036
.DEFINE monster.1.age $A038
.DEFINE monster.2    $A039
.DEFINE monster.2.name $A039
.DEFINE monster.2.age $A03B
.DEFINE monster.3    $A03C
.DEFINE monster.3.name $A03C
.DEFINE monster.3.age $A03E
.DEFINE dragon       $A03F
.DEFINE dragon.name  $A03F
.DEFINE dragon.age   $A041
```

DB, DW, DS, DSB, DSW and INSTANCEOF can also be in lowercase. You can also use a dotted version of the symbols, but it doesn't advance the memory address. Here's an example:

```
.ENUM $C000 DESC EXPORT
bigapple_h db
bigapple_l db
bigapple: .dw
.ENDE
```

And this is what is generated:

```
.DEFINE bigapple_h $BFFF
.DEFINE bigapple_l $BFFE
.DEFINE bigapple   $BFFE
.EXPORT bigapple, bigapple_l, bigapple_h
```

This way you can generate a 16-bit variable address along with pointers to its parts.

Note that you can also use DL (define long word, a 24-bit value) and DSL (define size, long words) when running wla-65816.

If you want more flexible variable positioning, take a look at .RAMSECTION s.

This is not a compulsory directive.

1.66 .ENUMID ID_1 0

.ENUMID will create definitions with an autoincrementing value. For example:

```
.ENUMID 0
.ENUMID ID_1
.ENUMID ID_2
.ENUMID ID_3
```

... will create the following definitions:

```
ID_1 = 0
ID_2 = 1
ID_3 = 2
```

You can also specify the address:

```
.ENUMID 0 STEP 2
.ENUMID MONSTER_ID_1
.ENUMID MONSTER_ID_2
.ENUMID MONSTER_ID_3
```

... to create definitions:

```
MONSTER_ID_1 = 0
MONSTER_ID_2 = 2
MONSTER_ID_3 = 4
```

If you wish to export the definitions automatically, use EXPORT:

```
.ENUMID 16 STEP 2 EXPORT
.ENUMID MUSIC_1
.ENUMID MUSIC_2
.ENUMID MUSIC_3
```

... will create the following definitions and export them all:

```
MUSIC_1 = 16
MUSIC_2 = 18
MUSIC_3 = 20
```

This is not a compulsory directive.

1.67 .EQU IF \$FF0F

.EQU is an alias for .DEFINE.

This is not a compulsory directive.

1.68 .EXHIROM

With this directive you can define the SNES ROM mode to be ExHiROM. Issuing .EXHIROM will override the user's ROM bank map when WLALINK computes 24-bit addresses and bank references. If no .HIROM, .LOROM or .EXHIROM are given then WLALINK obeys the banking defined in .ROMBANKMAP.

.EXHIROM also sets the ROM mode bit in \$40FFD5 (mirrored in \$FFD5).

This is not a compulsory directive.

1.69 .EXPORT work_x

Exports the definition `work_x` to outside world. Exported definitions are visible to all object files and libraries in the linking procedure. Note that you can only export value definitions, not string definitions.

You can export as many definitions as you wish with one .EXPORT:

```
.EXPORT NUMBER, NAME, ADDRESS, COUNTRY
.EXPORT NAME, AGE
```

This is not a compulsory directive.

1.70 .FAIL "THE EYE OF MORDOR HAS SEEN US!"

Terminates the compiling process. The string after .FAIL is optional.

This is not a compulsory directive.

1.71 .FARADDR main, irq_1

.FARADDR is an alias for .DL.

This is not a compulsory directive.

1.72 .FASTROM

Sets the ROM memory speed bit in \$FFD5 (.HIROM), \$7FD5 (.LOROM) or \$FFD5 and \$40FFD5 (.EXHIROM) to indicate that the SNES ROM chips are 120ns chips.

This is not a compulsory directive.

1.73 .FCLOSE FP_DATABIN

Closes the filehandle FP_DATABIN.

This is not a compulsory directive.

1.74 .FOPEN "data.bin" FP_DATABIN

Opens the file data.bin for reading and associates the filehandle with name FP_DATABIN.

This is not a compulsory directive.

1.75 .FREAD FP_DATABIN DATA

Reads one byte from FP_DATABIN and creates a definition called DATA to hold it. DATA is an ordinary definition label, so you can .UNDEFINE it.

Here's an example on how to use .FREAD:

```
.fopen "data.bin" fp
.fsize fp t
.repeat t
.fread fp d
.db d+26
.endr
.undefine t, d
```


This is not a compulsory directive.

1.76 .FSIZE FP_DATABIN SIZE

Creates a definition called `SIZE`, which holds the size of the file associated with the filehandle `FP_DATABIN`. `SIZE` is an ordinary definition label, so you can `.UNDEFINE` it.

This is not a compulsory directive.

1.77 .GBHEADER

This begins the GB header definition, and automatically defines `.COMPUTEGBCHECKSUM`. End the header definition with `.ENDGB`. Here's an example:

```
.GBHEADER
    NAME "TANKBOMBPANIC" ; identical to a freestanding .NAME.
    LICENSEECODEOLD $34  ; identical to a freestanding .LICENSEECODEOLD.
    LICENSEECODENNEW "HI" ; identical to a freestanding .LICENSEECODENNEW.
    CARTRIDGETYPE $00    ; identical to a freestanding .CARTRIDGETYPE.
    RAMSIZE $09          ; identical to a freestanding .RAMSIZE.
    COUNTRYCODE $01      ; identical to a freestanding .COUNTRYCODE/DESTINATIONCODE.
    DESTINATIONCODE $01  ; identical to a freestanding .DESTINATIONCODE/COUNTRYCODE.
    NINTENDOLOGO         ; identical to a freestanding .NINTENDOLOGO.
    VERSION $01          ; identical to a freestanding .VERSION.
    ROMDMG               ; identical to a freestanding .ROMDMG.
                        ; Alternatively, ROMGBC or ROMGBCONLY can be used
.ENDGB
```

This is not a compulsory directive.

1.78 .HEX "a0A0ffDE"

Defines bytes using the supplied string that contains the bytes in hexadecimal format. For example, the same result can be obtained using `.DB`

```
.DB $a0, $A0, $ff, $DE
```

This is not a compulsory directive.

1.79 .HIROM

With this directive you can define the SNES ROM mode to be HiROM. Issuing `.HIROM` will override the user's ROM bank map when `WLALINK` computes 24-bit addresses and bank references. If no `.HIROM`, `.LOROM` or `.EXHIROM` are given then `WLALINK` obeys the banking defined in `.ROMBANKMAP`.

`.HIROM` also sets the ROM mode bit in `$FFD5`.

This is not a compulsory directive.

1.80 .IF DEBUG == 2

If the condition is fulfilled the following piece of code is acknowledged until `.ENDIF/.ELSE` occurs in the text, otherwise it is skipped. Operands must be immediate values or strings.

The following operators are supported:

<	less than
<=	less or equal to
>	greater than
>=	greater or equal to
==	equals to
!=	doesn't equal to

All `IF` directives (yes, including `.IFDEF`, `.IFNDEF`, etc) can be nested. They can also be used within `ENUM` s, `RAMSECTION` s, `STRUCT` s, `ROMBANKMAP` s, and most other directives that occupy multiple lines.

This is not a compulsory directive.

1.81 .IFDEF IF

If `IF` is defined, then the following piece of code is acknowledged until `.ENDIF/.ELSE` occurs in the text, otherwise it is skipped.

This is not a compulsory directive.

1.82 .IFDEFM \2

If the specified argument is defined (argument number two, in the example), then the following piece of code is acknowledged until `.ENDIF/.ELSE` occurs in the macro, otherwise it is skipped.

This is not a compulsory directive. `.IFDEFM` works only inside a macro.

1.83 .IFEQ DEBUG 2

If the value of `DEBUG` equals to 2, then the following piece of code is acknowledged until `.ENDIF/.ELSE` occurs in the text, otherwise it is skipped. Both arguments can be computations, defines or immediate values.

This is not a compulsory directive.

1.84 .IFEXISTS "main.s"

If `main.s` file can be found, then the following piece of code is acknowledged until `.ENDIF/.ELSE` occurs in the text, otherwise it is skipped.

By writing the following few lines you can include a file if it exists without breaking the compiling loop if it doesn't exist:

```
.IFEXISTS FILE
.INCLUDE FILE
.ENDIF
```

This is not a compulsory directive.

1.85 .IFGR DEBUG 2

If the value of `DEBUG` is greater than 2, then the following piece of code is acknowledged until `.ENDIF/.ELSE` occurs in the text, otherwise it is skipped. Both arguments can be computations, defines or immediate values.

This is not a compulsory directive.

1.86 .IFGREQ DEBUG 2

If the value of `DEBUG` is greater or equal to 2, then the following piece of code is acknowledged until `.ENDIF/.ELSE` occurs in the text, otherwise it is skipped. Both arguments can be computations, defines or immediate values.

This is not a compulsory directive.

1.87 .IFLE DEBUG 2

If the value of `DEBUG` is less than 2, then the following piece of code is acknowledged until `.ENDIF/.ELSE` occurs in the text, otherwise it is skipped. Both arguments can be computations, defines or immediate values.

This is not a compulsory directive.

1.88 .IFLEEQ DEBUG 2

If the value of `DEBUG` is less or equal to 2, then the following piece of code is acknowledged until `.ENDIF/.ELSE` occurs in the text, otherwise it is skipped. Both arguments can be computations, defines or immediate values.

This is not a compulsory directive.

1.89 .IFNDEF IF

If `IF` is not defined, then the following piece of code is acknowledged until `.ENDIF/.ELSE` occurs in the text, otherwise it is skipped.

This is not a compulsory directive.

1.90 .IFNDEFM \2

If the specified argument is not defined, then the following piece of code is acknowledged until `.ENDIF/.ELSE` occurs in the macro, otherwise it is skipped.

This is not a compulsory directive. `.IFNDEFM` works only inside a macro.

1.91 .IFNEQ DEBUG 2

If the value of `DEBUG` doesn't equal to 2, then the following piece of code is acknowledged until `.ENDIF/.ELSE` occurs in the text, otherwise it is skipped. Both arguments can be computations, defines or immediate values.

This is not a compulsory directive.

1.92 .INC "cgb_hardware.i"

`INC` is an alias for `INCLUDE`.

This is not a compulsory directive.

1.93 .INCBIN "sorority.bin"

Includes the specified data file into the source file. `.INCBIN` caches all files into memory, so you can `.INCBIN` any data file millions of times, but it is loaded from hard drive only once.

You can optionally use `SWAP` after the file name, e.g.,

```
.INCBIN "kitten.bin" SWAP
```

`.INCBIN` data is divided into blocks of two bytes, and inside every block the bytes are exchanged (like `SWAP r` does to nibbles). This requires that the size of the file is even.

You can also force WLA to skip `n` bytes from the beginning of the file by writing for example:

```
.INCBIN "kitten.bin" SKIP 4
```

Four bytes are skipped from the beginning of `kitten.bin` and the rest is incbinned.

It is also possible to incbin only `n` bytes from a file:

```
.INCBIN "kitten.bin" READ 10
```

Will read ten bytes from the beginning of `kitten.bin`.

You can also force WLA to create a definition holding the size of the file:

```
.INCBIN "kitten.bin" FSIZE size_of_kitten
```

Want to circulate all the included bytes through a filter macro? Do this:

```
.INCBIN "kitten.bin" FILTER filtermacro
```

The filter macro is executed for each byte of the included data, data byte being the first argument, and offset from the beginning being the second parameter, just like in the case of `.DBM`, `.DWM`, `.DLM` and `.DDM`.

And you can combine all these four commands:

```
.INCBIN "kitten.bin" SKIP 10 READ 8 SWAP FSIZE size_of_kitten FILTER filtermacro
```

This example shows how to incbin eight bytes (swapped) after skipping 10 bytes from the beginning of file `kitten.bin`, and how to get the size of the file into a definition label `size_of_kitten`. All the data bytes are circulated through a filter macro.

Here's an example of a filter macro that increments all the bytes by one:

```
.macro filtermacro      ; the input byte is \1, the output byte is in "_out"
.redefine _out \1+1
.endm
```

Note that the order of the extra commands is important.

If the file's not found in the `.INCDIR` directory, WLA tries to find it in the current working directory. If the `INCDIR` is specified in the command line, WLA will first search for the file in that directory. If not found, it will then proceed as aforementioned.

This is not a compulsory directive.

1.94 `.INCDIR "/usr/programming/gb/include/"`

Changes the current include root directory. Use this to specify main directory for the following `.INCLUDE` and `.INCBIN` directives. If you want to change to the current working directory (WLA also defaults to this), use:

```
.INCDIR ""
```

If the `INCDIR` is specified in the command line, that directory will be searched before the `.INCDIR` in the file. If the file is not found, WLA will then silently search the specified `.INCDIR`.

This is not a compulsory directive.

1.95 `.INCLUDE "cgb_hardware.i"`

Includes the specified file to the source file. If the file's not found in the `.INCDIR` directory, WLA tries to find it in the current working directory. If the `INCDIR` is specified in the command line, WLA will first try to find the file specified in that directory. Then proceed as mentioned before if it is not found.

If you want to prefix all labels inside the included file with something, use:

```
.INCLUDE "music_player.s" NAMESPACE "musicplayer"
```

In the case of this example, all sections, macros, labels and references to those labels inside the included file are prefixed with "musicplayer.", though there are a couple of exceptions. If a `.SECTION` inside the included file has its own namespace, the `.INCLUDE`'s namespace doesn't affect it. If a `.SECTION` inside the included file uses `APPENDTO` with a section name that starts with `"*:"`, that `APPENDTO` is considered to belong to the global namespace and we won't prefix it with the `.INCLUDE`'s namespace.

Note that you can create the file name from pieces:

```
.INCLUDE ROOTDIR, SUBDIR, "cthulhu.s" NAMESPACE "cthulhu"
```

This might end up looking for a file "root/subdir/cthulhu.s", depending on the definitions.

If you are using the `.INCLUDE` inside a `.MACRO` and want to have the file included only once, use the keyword `ONCE`:

```
.INCLUDE "include_one.s" NAMESPACE "once" ONCE
```

This is not a compulsory directive.

1.96 .INDEX 8

Forces WLA to override the index (X/Y) register size given with SEP/REP. .INDEX doesn't produce any code, it only affects the way WLA interprets the immediate values (8 for 8 bit operands, 16 for 16 bit operands) for opcodes dealing with the index registers.

So after giving .INDEX 8

```
CPX #10
```

will produce \$E0 \$A0, and after giving .INDEX 16

```
CPX #10
```

will yield \$E0 \$00 \$A0.

Note that SEP/REP again will in turn reset the accumulator/index register size.

This is not a compulsory directive.

1.97 .INPUT NAME

.INPUT is much like any Basic-language input: .INPUT asks the user for a value or string. After .INPUT is the variable name used to store the data.

.INPUT works like .REDEFINE, but the user gets to type in the data.

Here are few examples how to use input:

```
.PRINTT "The name of the ROM? "
.INPUT NAME
.NAME NAME

...

.PRINTT "Give the .DB amount.\n"
.INPUT S
.PRINTT "Give .DB data one at a time.\n"
.REPEAT S
    .INPUT B
    .DB B
.ENDR

...
```

This is not a compulsory directive.

1.98 .LICENSEECODENEW "1A"

This is a standard new licensee code found at \$144 and \$145 in a Gameboy ROM, and there this one is put to also. .LICENSEECODENEW cannot be defined with .LICENSEECODEOLD. \$33 is inserted into \$14B, as well.

This is not a compulsory directive.

1.99 .LICENSEECODEOLD \$1A

This is a standard old licensee code found at \$14B in a Gameboy ROM, and there this one is put to also. `.LICENSEECODEOLD` cannot be defined with `.LICENSEECODENEW`.

This is not a compulsory directive.

1.100 .LONG \$102030, \$405060

`.LONG` is an alias for `.DL`.

This is not a compulsory directive.

1.101 .LOROM

With this directive you can define the SNES ROM mode to be LoROM. Issuing `.LOROM` will override the user's ROM bank map when WLALINK computes 24-bit addresses and bank references. If no `.HIROM`, `.LOROM` or `.EXHIROM` are given then WLALINK obeys the banking defined in `.ROMBANKMAP`.

WLA defaults to `.LOROM`.

This is not a compulsory directive.

1.102 .MACRO TEST

Begins a macro called `TEST`.

You can use `\@` inside a macro to e.g., separate a label from the other macro `TEST` occurrences. `\@` is replaced with an integer number indicating the amount of times the macro has been called previously so it is unique to every macro call. `\@` can also be used inside strings inside a macro or just as a plain value. Look at the following examples for more information.

You can also type `\!` to get the name of the source file currently being parsed. `\.` can be used the same way to get the name of the macro.

Also, if you want to use macro arguments in e.g., calculation, you can type `\X` where `X` is the number of the argument. Another way to refer to the arguments is to use their names given in the definition of the macro (see the examples for this).

Remember to use `.ENDM` to finish the macro definition. Note that you cannot use `.INCLUDE` inside a macro. Note that WLA's macros are in fact more like procedures than real macros, because WLA doesn't substitute macro calls with macro data. Instead WLA jumps to the macro when it encounters a macro call at compile time.

You can call macros from inside a macro. Note that the preprocessor does not expand the macros. WLA traverses through the code according to the macro calls.

Here are some examples:

```
.MACRO NOPMONSTER
    .REPT 32          ; gives us 32 NOPs
    NOP
    .ENDR
.ENDM
```

(continues on next page)

(continued from previous page)

```
.MACRO LOAD_ABCD
    LD A, \1
    LD B, \2
    LD C, \3
    LD D, :\4      ; load the bank number of \4 into register D.
    NOPMONSTER     ; note that \4 must be a label or ROM address
    LD HL, 1<<\1   ; for this to work...
.INCBIN \5
.ENDM

.MACRO QUEEN
QUEEN\@:
    LD A, \1
    LD B, \1
    CALL QUEEN\@

    .DB "\@", 0      ; will translate into a zero terminated string
                    ; holding the amount of macro QUEEN calls.
    .DB "\\@", 0     ; will translate into a string containing
                    ; \@.
    .DB \@          ; will translate into a number indicating
                    ; the amount of macro QUEEN calls.

.ENDM

.MACRO LOAD_ABCD_2 ARGS ONE, TWO, THREE, FOUR, FIVE
    LD A, ONE
    LD B, TWO
    LD C, THREE
    LD D, FOUR
    NOPMONSTER
    LD HL, 1<<ONE
.INCBIN FIVE
.ENDM

.MACRO TEST NARGS 3
    .DB \1, \2, \3
.ENDM
```

And here's how they can be used:

```
NOPMONSTER
LOAD_ABCD $10, $20, $30, XYZ, "merman.bin"
QUEEN 123
LOAD_ABCD_2 $10, $20, $30, XYZ, "merman.bin"
TEST 1, 2, 3
```

Note that you must separate the arguments with commas.

Here is a special case:

```
.DEF prev_test $0000

.MACRO .test ARGS str
__\._\@+1:      ; this will become __.test_1 during
```

(continues on next page)

(continued from previous page)

```
.PRINT __\._\@+1, "\n"      ; the first call, __.test_2 during the
.WORD  prev_test            ; second call...
.REDEF  prev_test __\._\@+1
.BYTE  str.length, str, 0
.ENDM
```

When creating a label inside a macro, you can add a super simple addition or subtraction after \@ to adjust the value. Only one digit number is supported.

If you want to give names to the macro's arguments you can do that by listing them in order after supplying ARGS after the macro's name.

Every time a macro is called a definition NARGS is created. It shows only inside the macro and holds the number of arguments the macro was called with. So don't have your own definition called NARGS. Here's an example:

```
.MACRO LUPIN
  .IF NARGS != 1
    .FAIL
  .ENDIF

  .PRINTT "Totsan! Ogenki ka?\n"
.ENDM
```

You can also use \? to ask for the type of the argument in the following fashion:

```
.macro .differentThings
  .if \?1 == ARG_NUMBER
    .db 1
  .endif
  .if \?1 == ARG_STRING
    .db 2
  .endif
  .if \?1 == ARG_LABEL
    .db 3
  .endif
  .if \?1 == ARG_PENDING_CALCULATION
    .db 4
  .endif
.endm

.section "TestingDifferentThings"
TDT1:
  .differentThings 100
  .differentThings "HELLO"
  .differentThings TDT1
  .differentThings TDT1+1
.ends
```

The previous example will result in .db 1, 2, 3, 4

This is not a compulsory directive.

1.103 .MEMORYMAP

Begins the memory map definition. Using .MEMORYMAP you must first describe the target system's memory architecture to WLA before it can start to compile the code. .MEMORYMAP gives you the freedom to use WLA to compile

data for numerous different real Z80/6502/65C02/65CE02/6510/6800/6801/6809/8008/8080/65816/HUC6280/SPC-700 based systems.

Examples:

```
.MEMORYMAP
DEFAULTSLOT 0
SLOTSIZE $4000
SLOT 0 $0000
SLOT 1 $4000
.ENDME

.MEMORYMAP
DEFAULTSLOT 0
SLOT 0 $0000 $4000 "ROMSlot"
SLOT 1 $4000 $4000 "RAMSlot"
.ENDME

.MEMORYMAP
DEFAULTSLOT 0
SLOT 0 START $0000 SIZE $4000 NAME "ROMSlot"
SLOT 1 START $4000 SIZE $4000 NAME "RAMSlot"
.ENDME

.MEMORYMAP
DEFAULTSLOT 1
SLOTSIZE $6000
SLOT 0 $0000
SLOTSIZE $2000
SLOT 1 $6000
SLOT 2 $8000
.ENDME
```

Here's a real life example from Adam Klotblix. It should be interesting for all the ZX81 coders:

```
...

.MEMORYMAP
DEFAULTSLOT 1
SLOTSIZE $2000
SLOT 0 $0000
SLOTSIZE $6000
SLOT 1 $2000
.ENDME

.ROMBANKMAP
BANKSTOTAL 2
BANKSIZE $2000
BANKS 1
BANKSIZE $6000
BANKS 1
.ENDRO

.BANK 1 SLOT 1
.ORG $2000

...
```

SLOTSIZE defines the size of the following slots, unless you explicitly specify the size of the slot, like in the second

and third examples. You can redefine `SLOTSIZE` as many times as you wish.

`DEFAULTSLOT` describes the default slot for banks which aren't explicitly inserted anywhere. Check `.BANK` definition for more information.

`SLOT` defines a slot and its starting address. `SLOT` numbering starts at 0 and ends to 255 so you have 256 slots at your disposal.

This is a compulsory directive, and make sure all the object files share the same `.MEMORYMAP` or you can't link them together.

1.104 `.NAME "NAME OF THE ROM"`

If `.NAME` is used with WLA-GB then the 16 bytes ranging from `$0134` to `$0143` are filled with the provided string. WLA-65816 fills the 21 bytes from `$FFC0` to `$FFD4` in HiROM and from `$7FC0` to `$7FD4` in LoROM mode with the name string (SNES ROM title). For ExHiROM the ranges are from `$40FFC0` to `$40FFD4` and from `$FFC0` to `$FFD4` (mirrored).

If the string is shorter than 16/21 bytes the remaining space is filled with `$00`.

This is not a compulsory directive.

1.105 `.NEXTU name`

Proceeds to the next entry in a union.

1.106 `.NINTENDOLOGO`

Places the required Nintendo logo into the Gameboy ROM at `$104`.

This is not a compulsory directive.

1.107 `.NOWDC`

Turns WLA-65816 into a mode where it accepts its default syntax assembly code, which doesn't support WDC standard. This is the default mode for WLA-65816.

This is not a compulsory directive.

1.108 `.ORG $150`

Defines the starting address. The value supplied here is relative to the ROM bank given with `.BANK`.

When WLA starts to parse a source file, `.ORG` is set to `$0`, but it's always a good idea to explicitly use `.ORG`, for clarity.

This is a compulsory directive.

1.109 .ORGA \$150

Defines the starting address. The value supplied here is absolute and used directly in address computations. WLA computes the right position in ROM file. By using .ORGA you can instantly see from the source file where the following code is located in the 16-bit memory.

Here's an example:

```
.MEMORYMAP
SLOTSIZE $4000
DEFAULTSLOT 0
SLOT 0 $0000
SLOT 1 $4000
.ENDME

.ROMBANKMAP
BANKSTOTAL 2
BANKSIZE $4000
BANKS 2
.ENDRO

.BANK 0 SLOT 1
.ORG $4000

MAIN:      JP      MAIN
```

Here MAIN is at \$0000 in the ROM file, but the address for label MAIN is \$4000. By using .ORGA instead of .ORG, you can directly see from the value the address where you want the code to be as .ORG is just an offset to the SLOT.

1.110 .OUTNAME "other.o"

Changes the name of the output file. Here's an example:

```
wla-gb -o test.o test.s
```

would normally output test.o, but if you had written:

```
.OUTNAME "new.o"
```

somewhere in the code WLA would write the output to new.o instead.

This is not a compulsory directive.

1.111 .PRINT "Numbers 1 and 10: ", DEC 1, " \$", HEX 10, "\n"

Prints strings and numbers to stdout. A combination and a more usable version of .PRINTT and .PRINTV. Useful for debugging.

Optional: Give DEC (decimal) or HEX (hexadecimal) before the value you want to print.

This is not a compulsory directive.

1.112 .PRINTT "Here we are...\n"

Prints the given text into stdout. Good for debugging stuff. PRINTT takes only a string as argument, and the only supported formatting symbol is \n (line feed).

This is not a compulsory directive.

1.113 .PRINTV DEC DEBUG+1

Prints the value of the supplied definition or computation into stdout. Computation must be solvable at the time of printing (just like definitions values). PRINTV takes max two parameters. The first describes the type of the print output. DEC means decimal, HEX means hexadecimal. This is optional. Default is DEC.

Use PRINTV with PRINTT as PRINTV doesn't print linefeeds, only the result. Here's an example:

```
.PRINTT "Value of \"DEBUG\" = $"
.PRINTV HEX DEBUG
.PRINTT "\n"
```

This is not a compulsory directive.

1.114 .RAMSECTION "Vars" BANK 0 SLOT 1 ALIGN 256 OFFSET 32

RAMSECTION s accept only variable labels and variable sizes, and the syntax to define these is identical to .ENUM (all the syntax rules that apply to .ENUM apply also to .RAMSECTION). Additionally you can embed structures (.STRUCT) into a RAMSECTION. Here's an example:

```
.RAMSECTION "Some of my variables" BANK 0 SLOT 1 RETURNORG PRIORITY 100
vbi_counter:    db
player_lives:   db
.ENDS
```

By default RAMSECTION s behave like FREE sections, but instead of filling any banks RAM sections will occupy RAM banks inside slots. You can fill different slots with different variable labels. It's recommend that you create separate slots for holding variables (as ROM and RAM don't usually overlap).

If you want that WLA returns the ORG to what it was before issuing the RAMSECTION, use the keyword RETURNORG.

Keyword PRIORITY means just the same as PRIORITY of a .SECTION, it is used to prioritize some sections when placing them in the output ROM/PRG. The RAMSECTION s with higher PRIORITY are placed first in the output, and if the priorities match, then bigger RAMSECTION s are placed first.

NOTE! Currently WLA-DX assumes that there are 256 RAM banks available for each slot in the memory map. There is no other way to limit this number at the moment than manually keep the BANK number inside real limits.

Anyway, here's another example:

```
.MEMORYMAP
SLOTSIZE $4000
DEFAULTSLOT 0
SLOT 0 $0000          ; ROM slot 0.
SLOT 1 $4000          ; ROM slot 1.
```

(continues on next page)

(continued from previous page)

```

SLOT 2 $A000 "RAMSlot" ; variable RAM is here!
.ENDME

.STRUCT game_object
x DB
y DB
.ENDST

.RAMSECTION "vars 1" BANK 0 SLOT 2
moomin1    DW
phantom     DB
nyanko      DB
enemy       INSTANCEOF game_object
.ENDS

.RAMSECTION "vars 2" BANK 1 SLOT "RAMSlot" ; Here we use slot 2
moomin2     DW
.ENDS

.RAMSECTION "vars 3" BANK 1 SLOT $A000      ; Slot 2 here as well...
moomin3_all .DSB 3
moomin3_a   DB
moomin3_b   DB
moomin3_c   DB
.ENDS

.RAMSECTION "vars 4" BANK 1 SLOT $A000
enemies     INSTANCEOF game_object 2 STARTFROM 0 ; If you leave away "STARTFROM 0"
↳the indexing will start from 1
.ENDS

```

If no other RAM sections are used, then this is what you will get:

```

.DEFINE moomin1    $A000
.DEFINE phantom    $A002
.DEFINE nyanko     $A003
.DEFINE enemy      $A004
.DEFINE enemy.x    $A004
.DEFINE enemy.y    $A005
.DEFINE moomin2    $A000
.DEFINE moomin3_all $A002
.DEFINE moomin3_a   $A002
.DEFINE moomin3_b   $A003
.DEFINE moomin3_c   $A004
.DEFINE enemies    $A005
.DEFINE enemies.0   $A005
.DEFINE enemies.0.x $A005
.DEFINE enemies.0.y $A006
.DEFINE enemies.1   $A007
.DEFINE enemies.1.x $A007
.DEFINE enemies.1.y $A008

```

BANK in .RAMSECTION is optional so you can leave it away if you don't switch RAM banks, or the target doesn't have them (defaults to 0).

NOTE! The generated `_sizeof_` labels for .RAMSECTION "vars 3" will be:

```
_sizeof_moomin3_all  (== 3)
_sizeof_moomin3_a    (== 1)
_sizeof_moomin3_b    (== 1)
_sizeof_moomin3_c    (== 1)
```

It is also possible to merge two or more sections using APPENDTO:

```
.RAMSECTION "RAMSection1" BANK 0 SLOT 0
label1      DB
.ENDS

.RAMSECTION "RAMSection2" APPENDTO "RAMSection1"
label2      DB
.ENDS
```

If you wish to skip some bytes without giving them labels, use `.` as a label:

```
.RAMSECTION "ZERO_PAGE" BANK 0 SLOT 0
UsingThisByte1: DB
.                DB ; RESERVED
.                DB ; RESERVED
UsingThisByte2: DB
.                DB ; RESERVED
UsingThisByte3: DB
.ENDS
```

If you want to use FORCE RAMSECTIONS that are fixed to a specified address, do as follows:

```
.RAMSECTION "FixedRAMSection" BANK 0 SLOT 0 ORGA $0 FORCE
.                DB ; SYSTEM RESERVED
.                DB ; SYSTEM RESERVED
PlayerX          DB
PlayerY          DB
.ENDS
```

Other types that are supported: SEMIFREE and SEMISUBFREE.

Here's the order in which WLA writes the RAM sections:

1. FORCE
2. SEMISUBFREE
3. SEMIFREE & FREE

NOTE: You can use ORGA to specify the fixed address for a FORCE RAMSECTION. ORG is also supported.

NOTE: When you have RAMSECTIONs inside libraries, you must give them BANKs and SLOTs in the linkfile, under [ramsections].

This is not a compulsory directive.

1.115 .RAMSIZE 0

Indicates the size of the RAM. This is a standard Gameboy RAM size indicator value found at \$149 in a Gameboy ROM, and there this one is put to also.

This is not a compulsory directive.

1.116 .REDEF IF \$0F

.REDEF is an alias for .REDEFINE.

This is not a compulsory directive.

1.117 .REDEFINE IF \$0F

Assigns a new value or a string to an old definition. If the definition doesn't exist, .REDEFINE performs .DEFINE's work.

When used with .REPT REDEFINE helps creating tables:

```
.DEFINE CNT 0

.REPT 256
.DB CNT
.REDEFINE CNT CNT+1
.ENDR
```

This is not a compulsory directive.

1.118 .REPEAT 6

Repeats the text enclosed between .REPEAT x and .ENDR x times (6 in this example). You can use .REPEAT s inside .REPEAT s. x must be bigger or equal than 0.

It's also possible to have the repeat counter/index in a definition:

```
.REPEAT 6 INDEX COUNT
.DB COUNT
.ENDR
```

This would define bytes 0, 1, 2, 3, 4 and 5.

This is not a compulsory directive.

1.119 .REPT 6

.REPT is an alias for .REPEAT.

This is not a compulsory directive.

1.120 .ROMBANKMAP

Begins the ROM bank map definition. You can use this directive to describe the project's ROM banks. Use .ROMBANKMAP when not all the ROM banks are of equal size. Note that you can use .ROMBANKSIZE and .ROMBANKS instead of .ROMBANKMAP, but that's only when the ROM banks are equal in size. Some systems based on a real Z80 chip, 6502/65C02/65CE02/6510/65816/6800/6801/6809/8008/8080/HUC6280/SPC-700 CPUs and Pocket Voice cartridges for Game Boy require the usage of this directive.

Examples:

```
.ROMBANKMAP
BANKSTOTAL 16
BANKSIZE $4000
BANKS 16
.ENDRO

.ROMBANKMAP
BANKSTOTAL 510
BANKSIZE $6000
BANKS 1
BANKSIZE $2000
BANKS 509
.ENDRO
```

The first one describes an ordinary ROM image of 16 equal sized banks. The second one defines a 4MB Pocket Voice ROM image. In the PV ROM image the first bank is \$6000 bytes and the remaining 509 banks are smaller ones, \$2000 bytes each.

`BANKSTOTAL` tells the total amount of ROM banks. It must be defined prior to anything else.

`BANKSIZE` tells the size of the following ROM banks. You can supply WLA with `BANKSIZE` as many times as you wish.

`BANKS` tells the amount of banks that follow and that are of the size `BANKSIZE` which has been previously defined.

This is not a compulsory directive when `.ROMBANKSIZE` and `.ROMBANKS` are defined.

You can redefine `.ROMBANKMAP` as many times as you wish as long as the old and the new ROM bank maps match as much as possible. This way you can enlarge the size of the project on the fly.

1.121 .ROMBANKS 2

Indicates the size of the ROM in rombanks. This value is converted to a standard Gameboy ROM size indicator value found at \$148 in a Gameboy ROM, and there this one is put into.

This is a compulsory directive unless `.ROMBANKMAP` is defined.

You can redefine `.ROMBANKS` as many times as you wish as long as the old and the new ROM bank maps match as much as possible. This way you can enlarge the size of the project on the fly.

1.122 .ROMBANKSIZE \$4000

Defines the ROM bank size. Old syntax is `.BANKSIZE x`.

This is a compulsory directive unless `.ROMBANKMAP` is defined.

1.123 .ROMDMG

Inserts data into the specific ROM location to mark the ROM as a DMG (Gameboy) ROM (\$00 -> \$0146). It will only run in DMG mode.

This is not a compulsory directive. `.ROMDMG` cannot be used with `.ROMSGB`.

1.124 .ROMGBCONLY

Inserts data into the specific ROM location to mark the ROM as a Gameboy Color ROM (\$C0 -> \$0143, so ROM name is max. 15 characters long). It will only run in GBC mode.

This is not a compulsory directive.

1.125 .ROMGBC

Inserts data into the specific ROM location to mark the ROM as a dual-mode ROM (\$80 -> \$0143, so ROM name is max. 15 characters long). It will run in either DMG or GBC mode.

This is not a compulsory directive.

1.126 .ROMSGB

Inserts data into the specific ROM location to mark the ROM as a Super Gameboy enhanced ROM (\$03 -> \$0146).

This is not a compulsory directive. .ROMSGB cannot be used with .ROMDMG.

1.127 .ROW \$ff00, 1, "3"

Defines bytes after a .TABLE has been used to define the format. An alternative way of defining bytes to .DB/.DW.

Note that when you use .ROW you'll need to give all the items .TABLE defines, i.e. one full row. To give more or less bytes use .DATA.

Example:

```
.TABLE word, byte, word
.ROW $aabb, "H", $ddee
```

This is the same as

```
.DW $aabb .DB "H" .DW $ddee
```

This is not a compulsory directive.

1.128 .SDSCTAG 1.0, "DUNGEON MAN", "A wild dungeon exploration game", "Ville Helin"

.SDSCTAG adds SDSC tag to your SMS/GG ROM file. The ROM size must be at least 8KB just like with .COMPUTESMSCHECKSUM and .SMSTAG. For more information about this header take a look at <http://www.smspower.org/dev/sdsc/>. Here's an explanation of the arguments:

```
.SDSCTAG {version number}, {program name}, {program release notes}, {program author}
```

Note that program name, release notes and program author can also be pointers to strings instead of being only strings (which WLA terminates with zero, and places them into suitable locations inside the ROM file). So:

```
.SDSCTAG 0.8, PRGNAME, PRGNOTES, PRGAUTHOR
...
PRGNAME: .DB "DUNGEON MAN", 0
PRGNOTES: .DB "A wild and totally crazy dungeon exploration game", 0
PRGAUTHOR: .DB "Ville Helin", 0
```

works also. All strings supplied explicitly to `.SDSCTAG` are placed somewhere in `.BANK 0 SLOT 0:`

```
.SDSCTAG 1.0, "", "", ""
.SDSCTAG 1.0, 0, 0, 0
```

are also valid, here 0 and "" mean the user doesn't want to use any descriptive strings. Version number can also be given as an integer, but then the minor version number defaults to zero.

`.SDSCTAG` also defines `.SMSTAG` (as it's part of the SDSC ROM tag specification).

This is not a compulsory directive.

1.129 .SECTION "Init" FORCE

Section is a continuous area of data which is placed into the output file according to the section type and `.BANK` and `.ORG` directive values.

The example begins a section called `Init`. Before a section can be declared, `.BANK` and `.ORG` must be used unless WLA is in library file output mode. Library file's sections must all be `FREE` ones. `.BANK` tells the bank number where this section will be later relocated into. `.ORG` tells the offset for the relocation from the beginning of `.BANK`.

You can put sections inside a namespace. For instance, if you put a section into a namespace called `bank0`, then labels in that section can be accessed with `bank0.label`. This is not necessary inside the section itself. The namespace directive should immediately follow the name:

```
.SECTION "Init" NAMESPACE "bank0"
```

You can give the size of the section, if you wish to force the section to some specific size, the following way:

```
.SECTION "Init" SIZE 100 FREE
```

It's possible to force WLALINK to align the `FREE`, `SEMIFREE` and `SUPERFREE` sections by giving the alignment as follows:

```
.SECTION "Init" SIZE 100 ALIGN 4 FREE
```

If you need an offset from the alignment, use `OFFSET`:

```
.SECTION "Init" SIZE 10 ALIGN 256 OFFSET 32 FREE
```

And if you want that WLA returns the `ORG` to what it was before issuing the section, put `RETURNORG` at the end of the parameter list:

```
.SECTION "Init" SIZE 100 ALIGN 4 FREE RETURNORG
```

By default WLA advances the `ORG`, so, for example, if your `ORG` was \$0 before a section of 16 bytes, then the `ORG` will be 16 after the section.

Note also that if your section name begins with double underlines (e.g., `__UNIQUE_SECTION!!`) the section will be unique in the sense that when WLALINK receives files containing sections which share the same name, WLALINK

will save only the first of them for further processing, all others are deleted from memory with corresponding labels, references and calculations.

If a section name begins with an exclamation mark (!) it tells WLALINK to not to drop it, even if you use WLALINK's ability to discard all unreferenced sections and there are no references to the section. You can achieve the same effect by adding KEEP to the end of the list:

```
.SECTION "Init" SIZE 100 ALIGN 4 FREE RETURNORG KEEP
```

FORCE after the name of the section tells WLA that the section *must* be inserted so it starts at .ORG. FORCE can be replaced with FREE which means that the section can be inserted somewhere in the defined bank, where there is room. You can also use OVERWRITE to insert the section into the memory regardless of data collisions. Using OVERWRITE you can easily patch an existing ROM image just by .BACKGROUND'ing the ROM image and inserting OVERWRITE sections into it. SEMIFREE sections are also possible and they behave much like FREE sections. The only difference is that they are positioned somewhere in the bank starting from .ORG. SEMISUBFREE sections on the other hand are positioned somewhere in the bank starting from \$0 and ending to .ORG.

SUPERFREE sections are also available, and they will be positioned into the first suitable place inside the first suitable bank (candidates for these suitable banks have the same size with the slot of the section, no other banks are considered). You can also leave away the type specifier as the default type for the section is FREE.

You can name the sections as you wish, but there is one special name. A section called BANKHEADER is placed in the front of the bank where it is defined. These sections contain data that is not in the memory map of the machine, so you can't refer to the data of a BANKHEADER section, but you can write references to outside. So no labels inside BANKHEADER sections. These special sections are useful when writing e.g., MSX programs. Note that library files don't take BANKHEADER sections.

Here's an example of a BANKHEADER section:

```
.BANK 0
.ORG 0
.SECTION "BANKHEADER"
    .DW MAIN
    .DW VBI
.ENDS

.SECTION "Program"
MAIN: CALL MONTY_ON_THE_RUN
VBI:  PUSH HL
      ...
      POP HL
      RETI
.ENDS
```

Here's an example of an ordinary section:

```
.BANK 0
.ORG $150
.SECTION "Init" FREE PRIORITY 1000
    DI
    LD SP, $FFFE
    SUB A
    LD ($FF00+R_IE), A
.ENDS
```

This tells WLA that a FREE section called Init must be located somewhere in bank 0 and it has a sorting PRIORITY of 1000. If you replace FREE with SEMIFREE the section will be inserted somewhere in the bank 0, but not in the \$0 - \$14F area. If you replace FREE with SUPERFREE the section will be inserted somewhere in any bank with the same size as bank 0.

Here's the order in which WLA writes the sections:

1. FORCE
2. SEMISUBFREE
3. SEMIFREE & FREE
4. SUPERFREE
5. OVERWRITE

Before the sections are inserted into the output file, they are sorted by priorities, so that the section with the highest priority is processed first. If priorities are the same, then the size of the section matters, and bigger sections are processed before smaller ones. The default PRIORITY, when not explicitly given, is 0.

You can also create a RAM section. For more information about them, please read the `.RAMSECTION` directive explanation.

It is also possible to merge two or more sections using APPENDTO:

```
.SECTION "Base"
.DB 0
.ENDS

.SECTION "AppendToBase" FREE RETURNORG APPENDTO "Base"
.DB 1
.ENDS
```

This is not a compulsory directive.

1.130 .SEED 123

Seeds the random number generator.

This is not a compulsory directive. The random number generator is initially seeded with the output of `time()`, which is, according to the manual, *the time since the Epoch (00:00:00 UTC, January 1, 1970), measured in seconds*. So if you don't `.SEED` the random number generator yourself with a constant value, `.DBRND` and `.DWRND` give you different values every time you run WLA.

In WLA DX 9.4a and before we used the stdlib's `srand()` and `rand()` functions making the output differ on different platforms. Since v9.4 WLA DX contains its own Mersenne Twister pseudo random number generator.

1.131 .SHIFT

Shifts the macro arguments one down (`\2` becomes `\1`, `\3` -> `\2`, etc.). `.SHIFT` can thus only be used inside a `.MACRO`.

This is not a compulsory directive.

1.132 .SLOT 1

Changes the currently active memory slot. This directive is meant to be used with `SUPERFREE` sections, where only the slot number is constant when placing the sections.

You can use the number, address or name of the slot here:

```
.SLOT 1          ; Use slot 1.
.SLOT $2000      ; Use a slot with starting address of $2000.
.SLOT "SlotOne"  ; Use a slot with a name "SlotOne"
```

This is not a compulsory directive.

1.133 .SLOWROM

Clears the ROM memory speed bit in \$FFD5 (.HIROM), \$7FD5 (.LOROM) or \$FFD5 and \$40FFD5 (.EXHIROM) to indicate that the SNES ROM chips are 200ns chips.

This is not a compulsory directive.

1.134 .SMC

Forces WLALINK to compute a proper SMC header for the ROM file.

SMC header is a chunk of 512 bytes. WLALINK touches only its first three bytes, and sets the rest to zeroes. Here's what will be inside the first three bytes:

Byte	Description
0	low byte of 8KB page count.
1	high byte of 8KB page count.
2	<ul style="list-style-type: none">• Bit 7: 0• Bit 6: 0• Bit 5: 0 = LoROM, 1 = HiROM• Bit 4: 0 = LoROM, 1 = HiROM• Bit 3 and 2: SRAM size (00 = 256Kb, 01 = 64Kb, 10 = 16Kb, 11 = 0Kb)• Bit 1: 0• Bit 0: 0

This is not a compulsory directive.

1.135 .SMSHEADER

All the fields in .SMSHEADER are optional and default to zero except ROMSIZE. If ROMSIZE is not specified it will be calculated automatically:

```
.SMSHEADER
  PRODUCTCODE 26, 70, 2 ; 2.5 bytes
  VERSION 1      ; 0-15
  REGIONCODE 4    ; 3-7
  RESERVEDSPACE 0, 0 ; 2 bytes
  ROMSIZE 0      ; 0-15
.ENDSMS
```

The REGIONCODE also defines the system:

3	SMS Japan
4	SMS Export
5	GG Japan
6	GG Export
7	GG International

When `.SMSHEADER` is defined, also the checksum is calculated, and TMR SEGA, two reserved bytes and ROM size are defined.

See <http://www.smspower.org/Development/ROMHeader> for more information about SMS header.

This is not a compulsory directive.

1.136 .SMSTAG

`.SMSTAG` forces WLA to write an ordinary SMS/GG ROM tag to the ROM file. Currently only the string TMR SEGA and ROM checksum are written (meaning that `.SMSTAG` also defines `.COMPUTESMSCHECKSUM`). The ROM size must be at least 8KBs.

This is not a compulsory directive.

1.137 .SNESEMUVECTOR

Begins definition of the emulation mode interrupt vector table:

```
.SNESEMUVECTOR
COP      COPHandler
UNUSED $0000
ABORT    BRKHandler
NMI      VBlank
RESET    Main
IRQBRK   IRQBRKHandler
.ENDEDUMVECTOR
```

These can be defined in any order, but they will be placed into memory starting at \$7FF4 (\$FFF4 in HiROM, \$40FFF4 and \$FFF4 in ExHiROM) in the order listed above. All the vectors default to \$0000.

This is not a compulsory directive.

1.138 .SNESHEADER

This begins the SNES header definition, and automatically defines `.COMPUTESNESCHECKSUM`. From here you may define any of the following:

- ID "ABCD" - inserts a one to four letter string starting at \$7FB2 (lorom) or \$FFB2 (hirom).
- NAME "Hello World!" - identical to a freestanding `.NAME`.
- LOROM - identical to a freestanding `.LOROM`.
- HIROM - identical to a freestanding `.HIROM`.

- EXHIROM - identical to a freestanding `.EXHIROM`.
- SLOWROM - identical to a freestanding `.SLOWROM`.
- FASTROM - identical to a freestanding `.FASTROM`.
- CARTRIDGETYPE `$00` - Places the given 8-bit value in `$7FD6` (`$FFD6` in HiROM, `$40FFD6` and `$FFD6` in ExHiROM). Some possible values I've come across but cannot guarantee the accuracy of:

<code>\$00</code>	ROM		
<code>\$01</code>	ROM	RAM	
<code>\$02</code>	ROM	SRAM	
<code>\$03</code>	ROM		DSP1
<code>\$04</code>	ROM	RAM	DSP1
<code>\$05</code>	ROM	SRAM	DSP1
<code>\$13</code>	ROM		Super FX

- ROMSIZE `$09` - Places the given 8-bit value in `$7FD7` (`$FFD7` in HiROM, `$40FFD7` and `$FFD7` in ExHiROM). Possible values include (but may not be limited to):

<code>\$08</code>	2 Megabits
<code>\$09</code>	4 Megabits
<code>\$0A</code>	8 Megabits
<code>\$0B</code>	16 Megabits
<code>\$0C</code>	32 Megabits

- SRAMSIZE `$01` - Places the given 2-bit value into `$7FD8` (`$FFD8` in HiROM, `$40FFD8` and `$FFD8` in ExHiROM). I believe these are the only possible values:

<code>\$00</code>	0 kilobits
<code>\$01</code>	16 kilobits
<code>\$02</code>	32 kilobits
<code>\$03</code>	64 kilobits

- COUNTRY `$00` - Places the given 8-bit value into `$7FD9` (`$FFD9` in HiROM, `$40FFD9` and `$FFD9` in ExHiROM). `$00` is Japan and `$01` is the United States, and there several more for other regions that I cannot recall off the top of my head.
- LICENSEECODE `$00` - Places the given 8-bit value into `$7FDA` (`$FFDA` in HiROM, `$40FFDA` and `$FFDA` in ExHiROM). You must find the legal values yourself as there are plenty of them. ;)
- VERSION `$01` - Places the given 8-bit value into `$7FDB` (`$FFDB` in HiROM, `$40FFDB` and `$FFDB` in ExHiROM). This is supposedly interpreted as version 1.byte, so a `$01` here would be version 1.01.

This is not a compulsory directive.

1.139 `.SNESNATIVEVECTOR`

Begins definition of the native mode interrupt vector table:

```
.SNESNATIVEVECTOR
COP      COPHandler
BRK      BRKHandler
```

(continues on next page)

(continued from previous page)

```

ABORT    ABORTHandler
NMI      VBlank
UNUSED   $0000
IRQ      IRQHandler
.ENDNATIVEVECTOR

```

These can be defined in any order, but they will be placed into memory starting at \$7FE4 (\$FFE4 in HiROM, \$40FFE4 and \$FFE4 in ExHiROM) in the order listed above. All the vectors default to \$0000.

This is not a compulsory directive.

1.140 .STRINGMAP script "Hello\n"

.ASC is an alias for .DB, but if you use .ASC it will remap the characters using the mapping given via .ASCIITABLE.

This is not a compulsory directive.

1.141 .STRINGMAPTABLE script "script.tbl"

.STRINGMAPTABLE's only purpose is to provide string mapping for .STRINGMAP. Take a look at the example:

```
.STRINGMAPTABLE script "script.tbl"
```

This will load the file “script.tbl” and define a new string mapping called “script”. This file is in the “table file” format commonly used for game translations; take a look at an example of one:

```

00=A
01=B
; This is a comment
ff01=
ff02=
fe=\n

```

The values to the left of the ‘=’ are a variable number of bytes expressed in hex, which map to the text value on the right. Note that depending on the text encoding of the file, this may be a variable number of bytes too. Thus this is a more flexible version of .ASCIITABLE.

After you've given the .STRINGMAPTABLE, use .STRINGMAP to define bytes using this mapping. For example:

```
.STRINGMAP script, "A\n"
```

This will map to the byte values FF 02 00 FE, provided the source file and TBL file use the same string encoding - use of UTF-8 is advised.

Note that all characters must be defined in the mapping - there is no fallback to ASCII encoding. You also cannot mix in byte values like with .DB and .ASC.

You can define multiple named string map tables.

This is not a compulsory directive.

1.142 .STRUCT enemy_object

Begins the definition of a structure. These structures can be placed inside `RAMSECTION` s and `ENUM` s. Here's an example:

```
.STRUCT enemy_object
id      dw          ; the insides of a .STRUCT are 1:1 like in .ENUM
x       db          ; except that no structs inside structs are
y       db          ; allowed.
data    ds  10
info    dsb 16
stats   dsw  4
.ENDST
```

This also creates a definition `_sizeof_[struct name]`, in our example this would be `_sizeof_enemy_object`, and the value of this definition is the size of the object, in bytes (2+1+1+10+16+4*2 = 38 in the example).

You'll get the following definitions as well:

```
enemy_object.id      (== 0)
enemy_object.x       (== 2)
enemy_object.y       (== 3)
enemy_object.data    (== 4)
enemy_object.info    (== 14)
enemy_object.stats   (== 30)
```

After defining a `.STRUCT` you can create an instance of it in a `.RAMSECTION` / `.ENUM` by typing:

```
<instance name> INSTANCEOF <struct name> [optional, the number of structures]
```

Here's an example:

```
.RAMSECTION "enemies" BANK 4 SLOT 4
enemies  INSTANCEOF enemy_object 4
enemyman INSTANCEOF enemy_object
enemyboss INSTANCEOF enemy_object
.ENDS
```

This will create definitions like `enemies`, `enemies.1.id`, `enemies.1.x`, `enemies.1.y` and so on. Definition `enemies` is followed by four `enemy_object` instances. After those four come `enemyman` and `enemyboss` instances, but as they are single instances, their definitions lack the index: `enemyman`, `enemyman.id`, `enemyman.x`, `enemyman.y` and so on.

Take a look at the documentation on `.RAMSECTION` & `.ENUM`, they have more examples of how you can use `.STRUCT` s.

A WORD OF WARNING: Don't use labels `b`, `B`, `w` and `W` inside a structure as e.g., WLA sees `enemy.b` as a byte sized reference to `enemy`. All other labels should be safe:

```
lda enemy1.b ; load a byte from zeropage address enemy1 or from the address
              ; of enemy1.b??? i can't tell you, and WLA can't tell you...
```

This is not a compulsory directive.

1.143 .SYM SAUSAGE

WLA treats symbols (SAUSAGE in this example) like labels, but they only appear in the symbol files WLALINK outputs. Useful for finding out the location where WLALINK puts data.

This is not a compulsory directive.

1.144 .SYMBOL SAUSAGE

.SYMBOL is an alias for .SYM.

This is not a compulsory directive.

1.145 .TABLE byte, word, byte

Defines table's columns. With .DATA and .ROW you can define data much like using .DB or .DW, but .TABLE makes it convenient to feed big amounts of data in mixed format.

For example:

```
.TABLE byte, word, byte
```

After the columns have been defined, you can define rows using e.g.,

```
.ROW $01, $0302, $04
```

This is the same as:

```
.DB $01
.DW $0302
.DB $04
```

Note that .DATA can also be used instead of .ROW, if one wants to give the data in pieces.

All supported column formats:

- DB, BYT, BYTE
- DW, WORD, ADDR
- DL, LONG, FARADDR ; wla-65816 only
- DS, DSB
- DSW
- DSL ; wla-65816 only

This is not a compulsory directive.

1.146 .UNBACKGROUND \$1000 \$1FFF

After issuing .BACKGROUND you might want to free some parts of the backgrounded ROM image for e.g., FREE sections. With .UNBACKGROUND you can define such regions. In the example a block starting at \$1000 and ending at \$1FFF was released (both ends included). You can issue .UNBACKGROUND as many times as you wish.

This is not a compulsory directive.

1.147 .UNDEF DEBUG

.UNDEF is an alias for .UNDEFINE.

This is not a compulsory directive.

1.148 .UNDEFINE DEBUG

Removes the supplied definition label from system. If there is no such label as given no error is displayed as the result would be the same.

You can undefine as many definitions as you wish with one .UNDEFINE:

```
.UNDEFINE NUMBER, NAME, ADDRESS, COUNTRY
.UNDEFINE NAME, AGE
```

This is not a compulsory directive.

1.149 .UNION name

Begins a “union”. This can only be used in enums, ramsections, and structs.

When entering a union, the current address in the enum is saved, and the following data is processed as normal. When the .NEXTU directive is encountered, the address is reverted back to the start of the union. This allows one to assign an area of memory to multiple labels:

```
.ENUM $C000
    .UNION
        pos_lowbyte:  db
        pos_highbyte: db
        extra_word:   dw
    .NEXTU
        pos:          dw
    .ENDU
    after: db
.ENDE
```

This example is equivalent to:

```
.DEFINE pos_lowbyte  $c000
.DEFINE pos_highbyte $c001
.DEFINE extra_word   $c002
.DEFINE pos          $c000
.DEFINE after        $c004
```

The .UNION and .NEXTU commands can be given an argument to assign a prefix to the labels that follow:

```
.ENUM $C000
    .UNION union1
        bytel: db
```

(continues on next page)

(continued from previous page)

```

        byte2: db
    .NEXTU union2
        word1: dw
    .ENDU
.ENDE

```

This example is equivalent to:

```

.DEFINE union1.byte1 $c000
.DEFINE union1.byte2 $c001
.DEFINE union2.word1 $c000

```

Unions can be nested.

1.150 .VERSION 1

Indicates the Mask ROM version number located at \$14C of a Gameboy ROM.

This is not a compulsory directive.

1.151 .WDC

Turns WLA-65816 into a mode where it accepts WDC standard assembly code, in addition to WLA's own syntax. In WDC standard mode:

```

AND <x ; 8-bit
AND |? ; 16-bit
AND >& ; 24-bit

```

are the same as:

```

AND x.b ; 8-bit
AND ?.w ; 16-bit
AND &.l ; 24-bit

```

in WLA's own syntax. Beware of the situations where you use '<' and '>' to get the low and high bytes!

This is not a compulsory directive.

1.152 .WORD 16000, 10, 255

.WORD is an alias for .DW.

This is not a compulsory directive.

2.1 Case Sensitivity

WLA is case sensitive, except with directives, so be careful.

2.2 Comments

Comments begin with `;` or `*` and end along with the line. `;` can be used anywhere, but `*` can be placed only at the beginning of a new line.

WLA supports also ANSI C style commenting. This means you can start a multiline comment with `/*` and end it with `*/`.

What also is supported are C++ style comments. This means you can start a comment with `//`.

You can also use `.ASM` and `.ENDASM` directives to skip characters. These function much like ANSI C comments, but unlike the ANSI C comments these can be nested.

2.3 Line splitting

Lines can be split using a `\` between elements. So instead of writing

```
.db 1, 2, 3, 4, 5, 6, 7, 8
```

it's possible to write

```
.db 1, 2, 3, 4 \ 5, 6, 7, 8
```

Note that line splitting works only in places where WLA expects a new label, number, calculation, etc. String splitting isn't currently supported.

2.4 Labels

Labels are ordinary strings (which can also end to a `:`). Labels starting with `_` are considered to be local labels and do not show outside sections where they were defined, or outside object files, if they were not defined inside a section.

Here are few examples of different labels:

```
VBI_IRQ:
VBI_IRQ2
_VBI_LOOP:
main:
```

Labels starting with `@` are considered to be child labels. They can only be referenced within the scope of their parent labels, unless the full name is specified. When there is more than one `@`, the label is considered to be a child of a child.

Here are some examples of child labels:

```
PARENT1:
@CHILD:
@@SUBCHILD

PARENT2:
@CHILD:
```

This is legal, since each of the `@CHILD` labels has a different parent. You can specify a parent to be explicit, like so:

```
jr PARENT1@CHILD@SUBCHILD
```

You can also use `__label__` to refer to the last defined parent label:

```
main:                                ; #
    nop
    nop
@child:
    nop
    nop
@@grandchild:
    nop
    nop
    jmp __label__ ; jump -> #
loop:  nop                ; %
    nop
    jmp __label__ ; jump -> %
```

Note that when you place `:` in front of the label string when referring to it, you'll get the bank number of the label, instead of the label's address. Here's an example:

```
LD A, :LOOP
.BANK 2 SLOT 0
LOOP:
```

Here `LD A, :LOOP` will be replaced with `LD A, 2` as the label `LOOP` is inside the bank number two.

When you are referring to a label and you are adding something to its address (or subtracting, any arithmetics apply) the result will always be bytes.

```
.org 20
DATA: .dw 100, 200, 300
```

(continues on next page)

(continued from previous page)

```
ld a, DATA+1
^^^^^^ = r
```

So here the result `r` will be the address of `DATA` plus one, here 21. Some x86 assemblers would give here 22 as the result `r` as `DATA` points to an array or machine words, but WLA isn't that smart (and some people including me think this is the better solution).

Note that each CPU WLA supports contains opcodes that either generate an absolute reference or a relative reference to the given label. For example,

```
.org 20
DATA: ld a, DATA ; DATA becomes 20 (absolute)
      jr DATA    ; DATA becomes -4 (relative)
```

Check out section 14 for the list of opcodes that generate relative references.

You can also use `-`, `--`, `---`, `+`, `++`, `+++`, ... as un-named labels. Labels consisting of `-` are meant for reverse jumps and labels consisting of `+` are meant for forward jumps. You can reuse un-named labels as much as you wish inside your source code. Here's an example of this:

```
dec e
beq ++      ; jump -> ?
dec e
beq +       ; jump -> %
ld d, 14
--- ld a, 10 ; !
--  ld b, c  ; #
-   dec b    ; *
    jp nz, -  ; jump -> *
dec c
    jp nz, -- ; jump -> #
dec d
    jp nz, --- ; jump -> !
ld a, 20
-   dec a    ; $
    jp nz, -  ; jump -> $
+   halt     ; %
++  nop      ; ?
```

Note that `__` (that's two underline characters) serves also as a un-named label. You can refer to this label from both directions. Use `_b` when you are jumping backwards and `_f` when you are jumping forwards label `__`.

Example:

```
dec e
jp z, _f      ; jump -> *
dec e
__ ldi a, (hl) ; *
dec e
jp nz, _b     ; jump -> *
```

CAVEAT! CAVEAT! CAVEAT!

The following code doesn't work as it would if WLA would determine the distance lexically (but in practice it's WLALINK that does all the calculations and sees only the preprocessed output of WLA):

```
.macro dummy
-   dec a           ; #
    jp nz, -        ; jump -> #
.endm

...
-   nop             ; *
    dummy
    dec e
    jp nz, -        ; i'd like to jump to *, but i'll end up jumping
                        ; to # as it's closest to me in the output WLA produces
                        ; for WLALINK (so it's better to use \@ with labels inside
                        ; a macro).
```

WLALINK will also generate `_sizeof_[label]` defines that measure the distance between two consecutive labels. These labels have the same scope as the labels they describe. Here is an example:

```
Label1:
    .db 1, 2, 3, 4
Label2:
```

In this case you'll get a definition `_sizeof_Label1` that will have value 4.

WLA will skip over any child labels when calculating `_sizeof`. So, in this example:

```
Label1:
    .db 1, 2
@child:
    .db 3, 4
Label2:
```

The value of `_sizeof_Label1` will still have a value of 4.

2.5 Number Types

1000	decimal
\$100	hexadecimal
100h	hexadecimal
%100	binary
'x'	character

Remember that if you use the suffix `h` to give a hexadecimal value, and the value begins with an alphabet, you must place a zero in front of it so WLA knows it's not a label (e.g., `0ah` instead of `ah`).

2.6 Strings

Strings begin with and end to `"`. Note that no `0` is inserted to indicate the termination of the string like in e.g., ANSI C. You'll have to do it yourself. You can place quotation marks inside strings the way C preprocessors accept them.

Here are some examples of strings:

```
"Hello world!"  
"He said: \"Please, kiss me honey.\""
```

2.7 Mnemonics

You can give the operand size with the operand itself (and this is highly recommended) in WLA 6502/65C02/65CE02/6510/HUC6280/65816/6800/6801/6809:

```
and #20.b  
and #20.w  
bit loop.b  
bit loop.w
```

2.8 Brackets?

You can write

```
LDI (HL), A
```

or

```
LDI [HL], A
```

as both mean the same thing in the syntax of most of the supported CPUs. Yes, you could write

```
LDI [HL), A
```

but that is not recommended.

Note that brackets have special meaning when dealing with a 65816/SPC-700 system so you can't use

```
AND [$65]
```

instead of

```
AND ($65)
```

as they mean different things.

CHAPTER 3

Error Messages

There are quite a few of them in WLA, but most of them are not as informative as I would like them to be. This will be fixed in the future. Mean while, be careful. ;)

Supported ROM/RAM/Cartridge Types (WLA-GB)

4.1 ROM Size

GB-Z80 version of WLA supports the following ROM bank sizes. There's no such limit in the Z80/6502/65C02/65CE02/6510/65816/6800/6801/6809/8008/8080/HUC6280/SPC-700 version of WLA. Supply one of the following values to `.ROMBANKS`.

\$00	256Kbit	32KByte	2 banks
\$01	512Kbit	64KByte	4 banks
\$02	1Mbit	128KByte	8 banks
\$03	2Mbit	256KByte	16 banks
\$04	4Mbit	512KByte	32 banks
\$05	8Mbit	1MByte	64 banks
\$06	16Mbit	2MByte	128 banks
\$07	32Mbit	4MByte	256 banks
\$08	64Mbit	8MByte	512 banks
\$52	9Mbit	1.1MByte	72 banks
\$53	10Mbit	1.2MByte	80 banks
\$54	12Mbit	1.5MByte	96 banks

4.2 RAM Size

Supply one of the following hex values to `.RAMSIZE` in the GB-Z80 version of WLA.

\$00	None	None	None
\$01	16kbit	2kByte	1 bank
\$02	64kbit	8kByte	1 bank
\$03	256kbit	32kByte	4 banks
\$04	1Mbit	128kByte	16 banks
\$05	512kbit	64kByte	8 banks

4.3 Cartridge Type

It's up to the user to check that the cartridge type is valid and can be used combined with the supplied ROM and RAM sizes. Give one the the following values to `.CARTRIDGETYPE` in the GB-Z80 version of WLA.

\$00	ROM					
\$01	ROM	MBC1				
\$02	ROM	MBC1	RAM			
\$03	ROM	MBC1	RAM	BATTERY		
\$05	ROM	MBC2				
\$06	ROM	MBC2		BATTERY		
\$08	ROM		RAM			
\$09	ROM		RAM	BATTERY		
\$0B	ROM	MMM01				
\$0C	ROM	MMM01	SRAM			
\$0D	ROM	MMM01	SRAM	BATTERY		
\$0F	ROM	MBC3		BATTERY	TIMER	
\$10	ROM	MBC3	RAM	BATTERY	TIMER	
\$11	ROM	MBC3				
\$12	ROM	MBC3	RAM			
\$13	ROM	MBC3	RAM	BATTERY		
\$19	ROM	MBC5				
\$1A	ROM	MBC5	RAM			
\$1B	ROM	MBC5	RAM	BATTERY		
\$1C	ROM	MBC5				RUMBLE
\$1D	ROM	MBC5	SRAM			RUMBLE
\$1E	ROM	MBC5	SRAM	BATTERY		RUMBLE
\$20	MBC6					
\$22	MBC7					
\$BE	Pocket Voice					
\$FC	Pocket Camera					
\$FD	Bandai TAMA5					
\$FE	Hudson HuC-3					
\$FF	Hudson HuC-1					

CHAPTER 5

Bugs

If you find bugs, please let us know about them via GitHub: <https://github.com/vhelin/wla-dx/issues>

6.1 tests

The main purpose of the files in the `tests` directory is to test that WLA and WLALINK can assemble and link the tiny project correctly. You can also take a look at the code and syntax in the files, but beware: if you run the rom files you probably don't see anything on screen.

`include` directory under `gb-z80` could be very useful as the six include files there have all the Game Boy hardware register address and memory definitions you could ever need and more.

6.2 tests/gb-z80/lib

This folder holds few very useful libraries for you to use in your Game Boy projects. Instead of reinventing the wheel, use the stuff found in here. Remember to compile the libraries right after you've installed WLA by executing `make` in the `lib` directory.

6.3 memorymaps

Here you can find default memory maps (see `.MEMORYMAP`) for various computers and video game consoles.

CHAPTER 7

Temporary Files

Note that WLA will generate an temporary files in the current working directory while it works. On Windows and Unix-like systems, the file is called `.wla%PID%.a`, where `%PID%` is the PID of the process. For other system, it's just `wla_a.tmp`.

When WLA finishes its work these two files are deleted as they serve of no further use.

8.1 Compiling Object Files

To compile an object file use the `-o [OUT]` option on the command line.

These object files can be linked together (or with library files) later with WLALINK.

Name object files so that they can be recognized as object files. Normal suffix is `.o` (WLA default). This can also be changed with `.OUTNAME`.

With object files you can reduce the amount of compiling when editing small parts of the program. Note also the possibility of using local labels (starting with `_`).

Note: When you compile objects, group 1 directives are saved for linking time, when they are all compared and if they differ, an error message is shown. It is advisable to use something like an include file to hold all the group 1 directives for that particular project and include it to every object file.

If you are interested in the WLA object file format, take a look at the file `txt/wla_file_formats.txt` which is included in the release archive.

Here are some examples of definitions:

- `-D IEXIST`
- `-D DAY=10`
- `-D BASE = $10`
- `-D NAME=elvis`

And here's an WLA example creating definitions on the command line:

```
wla-gb -D DEBUG -D VERBOSE=5 -D NAME = "math v1.0" -o math.o math.s
```

DEBUG's value will be 0, VERBOSE's 5 and NAME is a string definition with value `math v1.0`. Note that `-D` always needs a space after it, but the rest of the statement can be optionally stuck inside one word.

8.2 Compiling Library Files

To compile a library file use the `-l [OUT]` option on the command line.

Name these files so that they can be recognized as library files. Normal suffix is `.lib` (WLA default).

With library files you can reduce the amount of compiling. Library files are meant to hold general functions that can be used in different projects. Note also the possibility of using local labels (starting with `_`). Library files consist only of `FREE` sections.

CHAPTER 9

Linking

After you have produced one or more object files and perhaps some library files, you might want to link them together to produce a ROM image / program file. WLALINK is the program you use for that. Here's how you use it:

```
wlalink [OPTIONS] <LINK FILE> <OUTPUT FILE>
```

Choose the option `-b [OUT]` for program file or `-r [OUT]` for ROM image linking. ROM image is all the data in the ROM banks. Program file is the data between the first used byte and the last used byte. You can also use `-bS [START ADDRESS]` and `-bE [END ADDRESS]` to specify the start and the end addresses of the program. Both are optional.

Link file is a text file that contains information about the files you want to link together. Here's the format:

1. You must define the group for the files. Put the name of the group inside brackets. Valid group definitions are

```
[objects]
[libraries]
[header]
[footer]
[definitions]
[ramsections]
[sections]
```

2. Start to list the file names.

```
[objects]
main.o
vbi.o
level_01.o
...
```

3. Give parameters to the library files:

```
[libraries]
bank 0 slot 1 speed.lib
```

(continues on next page)

(continued from previous page)

```
bank 4 slot 2 map_data.lib
...
```

Here you can also use `base` to define the 65816 CPU bank number (like `.BASE` works in WLA):

```
[libraries]
bank 0 slot 1 base $80 speed.lib
bank 4 slot 2 base $80 map_data.lib
...
```

You must tell WLALINK the bank and the slot for the library files.

4. If you want to use header and/or footer in your project, you can type the following:

```
[header]
header.dat
[footer]
footer.dat
```

5. If you have RAMSECTIONS inside the libraries, you must place the sections inside BANKs and SLOTs (ORG and ORGA are optional). Note that you can also change the type and priority of the section, and can use `appendto`:

```
[ramsections]
bank 0 slot 3 org $0 "library 1 vars 1"
bank 0 slot 3 orga $6100 priority 100 force "library 1 vars 2"
bank 0 slot 3 appendto "library 1 vars 2" "library 1 vars 3"
```

6. If you want to relocate normal sections, do as follows (ORG, ORGA, KEEP, PRIORITY and APPENDTO are optional, but useful):

```
[sections]
bank 0 slot 1 org $100 appendto "MusicPlayers" "MusicPlayer1"
bank 0 slot 1 orga $2200 semisubfree priority 100 keep "EnemyAI"
```

7. If you want to make value definitions, here's your chance:

```
[definitions]
debug 1
max_str_len 128
start $150
...
```

If flag `v` is used, WLALINK displays information about ROM file after a successful linking.

If flag `nS` is used, WLALINK doesn't sort the sections at all, so they are placed in the output in their order of appearance.

If flag `s` is used, WLALINK will produce a `NOGMB/NOSNES` symbol file. It's useful when you work under MSDOS (`NO$GMB` is a very good Game Boy emulator for MSDOS/Windows) as it contains information about the labels in your project.

If flag `S` is used, WLALINK will create a WLA symbol file, that is much like `NO$GMB` symbol file, but shows also symbols, defines, and breakpoints, not just labels.

If flag `d` is used, WLALINK discards all unreferenced `FREE`, `SEMIFREE`, `SEMISUBFREE`, `SUPERFREE` and `RAM` sections. This way you can link big libraries to your project and WLALINK will choose only the used sections, so you won't be linking any dead code/data.

If flag `D` is used, WLALINK doesn't create any `_sizeof_*` labels. Note that to disable fully `_sizeof_*` label creation, you'll also need to give WLA the `s` flag.

If flag `t` is used with `c64PRG`, WLALINK will add a two byte header to the program file (use with flag `b`). The header contains the load address for the PRG. Use the flag `a` to specify the load address. It can be a value or the name of a label.

If flag `i` is given, WLALINK will write list files. Note that you must compile the object and library files with `-i` flag as well. Otherwise WLALINK has no extra information it needs to build list files. Here is an example of a list file: Let's assume you've compiled a source file called `main.s` using the `i` flag. After you've linked the result also with the `i` flag WLALINK has created a list file called `main.lst`. This file contains the source text and the result data the source compiled into. List files are good for debugging. NOTE: list file data can currently be generated only for code inside sections. `.MACRO` calls and `.REPT` s don't produce list file data either.

If flag `L` is given after the above options, WLALINK will use the directory specified after the flag for including libraries. If WLALINK cannot find the library in the specified directory, it will then silently search the current working directory. This is useful when using WLA in an SDK environment where a global path is needed.

Make sure you don't create duplicate labels in different places in the memory map as they break the linking loop. Duplicate labels are allowed when they overlap each other in the destination machine's memory. Look at the following example:

```
...
.BANK 0
.ORG $150

    ...
    LD      A, 1
    CALL    LOAD_LEVEL
    ...

LOAD_LEVEL:
    LD      HL, $2000
    LD      (HL), A
    CALL    INIT_LEVEL
    RET

.BANK 1
.ORG 0

INIT_LEVEL:
    ...
    RET

.BANK 2
.ORG $0

INIT_LEVEL:
    ...
    RET
...
```

Here duplicate `INIT_LEVEL` labels are accepted as they both point to the same memory address (in the program's point of view).

Examples:

```
[seravy@localhost tbp]# wialink -r linkfile testa.sfc
[seravy@localhost tbp]# wialink -d -i -b linkfile testb.sfc
```

(continues on next page)

(continued from previous page)

```
[seravy@localhost tbp]# wlalink -v -S -L ../../lib linkfile testc.sfc  
[seravy@localhost tbp]# wlalink -v -b -s -t c64PRG -a LOAD_ADDRESS linkfile linked.prg
```

CHAPTER 10

Arithmetics

WLA is able to solve really complex calculations like

```
-( (HELLO / 2) | 3)
skeletor_end-skeletor
10/2.5
```

so you can write something like

```
LD HL, data_end-data
LD A, (pointer + 1)
CP (TEST + %100) & %10101010
```

WLALINK also has this ability so it can compute the pending calculations WLA wasn't able to solve.

The following operators are valid:

,	comma
	or
&	and
^	power
<<	shift left
>>	shift right
+	plus
-	minus
#	modulo
~	xor
*	multiply
/	divide
<	get the low byte
>	get the high byte

Note that you can do NOT using XOR:

- `VALUE_A ~ $FF` is 8-bit NOT
- `VALUE_B ~ $FFFF` is 16-bit NOT

Unary XOR (e.g., `~$FF`) is the same as NOT.

WLA computes internally with real numbers so $(5/2) * 2$ produces 5, not 4.

CHAPTER 11

Binary to DB Conversion

WLAB converts binary files to WLA's byte definition strings. Here's how you use it:

```
wlab -[ap]{bdh} <BIN FILE>
```

Give it the binary file and WLAB will output the WLA DB formatted data of it into stdout. Here's an example from real life:

```
wlab -da gayskeletor.bin > gayskeletor.s
```

WLAB has three command flags of which one must be given to WLAB:

- b** Output data in binary format.
- d** Output data in decimal format.
- h** Output data in hexadecimal format.

WLAB has also two option flags:

- a** Print the address (relative to the beginning of the data).
- p** Don't print file header.

Examples:

```
[seravy@localhost src]# wlab -bap iscandar.bin > iscandar.s  
[seravy@localhost src]# wlab -h starsha.bin > starsha.s
```


CHAPTER 12

Things you should know about coding for...

Please check out the source code examples (in `tests` directory) for quick target system specific information.

Please be aware that the source code files in there are mainly used to test that the compiler and linker work, they are not possibly good examples of how you should write code using WLA DX.

12.1 Z80

Check the Z80 specific directives. All SMS/GG coders should find `.SMSTAG`, `.SDSCTAG` and `.COMPUTESMSCHECKSUM` very useful...

There are shadow register aliases for opcodes that use registers A, F, BC, DE and HL. The shadow register versions are just for convenience, if the programmer wants to explicitly show that he is now using the shadow registers. For example:

AND A ; (original, assembles to 0xA7) AND A' ; (alias, assembles to 0xA7 and is in reality "AND A")

Opcodes that make relative label references:

```
JR *  
DJNZ
```

12.2 6502

For example mnemonics ADC, AND, ASL, etc... cause problems to WLA, because they take different sized arguments. Take a look at this:

```
LSR 11      ; $46 $0B  
LSR $A000   ; $4E $00 $A0
```

The first one could also be

```
LSR 11      ; $4E $0B $00
```

To really get what you want, use `.8BIT`, `.16BIT` and `.24BIT` directives. Or even better, supply WLA the size of the argument:

```
LSR 11.W    ; $4E $0B $00
```

Opcodes that make relative label references:

```
BCC
BCS
BEQ
BMI
BNE
BPL
BVC
BVS
```

12.3 65C02

Read the subsection [6502](#) as the information applies also to 65C02 coding...

Opcodes that make relative label references:

```
BCC
BCS
BEQ
BMI
BNE
BPL
BVC
BVS
BRA
BBR*
BBS*
```

12.4 65CE02

Read the subsection [6502](#) as the information applies also to 65CE02 coding...

Opcodes that make relative label references:

```
BCC
BCS
BEQ
BMI
BNE
BPL
BVC
BVS
BRA
BSR
```

(continues on next page)

(continued from previous page)

```
BBR*
BBS*
```

12.5 6510

Read the subsection [6502](#) as the information applies also to 6510 coding...

Opcodes that make relative label references:

```
BCC
BCS
BEQ
BMI
BNE
BPL
BVC
BVS
```

12.6 65816

Read the subsection [6502](#) as the information applies also to 65816 coding...

WLA-65816 has also few SNES specific directives which are all very helpful. Remember that when you use `.LOROM`, `.HIROM`, `.SLOWROM` and `.FASTROM` WLA automatically writes the information into the output. `.COMPUTESNESCHECKSUM`, `.SNESHEADER` and few others could also be useful.

Use `.BASE` to set the upmost eight bits of 24-bit addresses.

If possible, use operand hints to specify the size of the operand. WLA is able to deduce the accumulator/index mode to some extent from REP/SEP-mnemonics and `.ACCU` and `.INDEX`-directives, but just to be sure, terminate the operand with `.B`, `.W` or `.L`.

```
AND #10      ; can be two different things, depending on the size of the accu.
AND #10.B    ; forces 8-bit immediate value.
AND #10.W    ; forces 16-bit immediate value.
```

Or if you must, these work as well:

```
AND.B #10    ; the same as "AND #10.B".
AND.W #10    ; the same as "AND #10.W".
```

Opcodes that make relative label references:

```
BCC
BCS
BEQ
BMI
BNE
BPL
BVC
BVS
BRA
```

(continues on next page)

(continued from previous page)

```
BRL
PER
```

Use `.WDC` to start parsing WDC standard assembly code. `.NOWDC` sets the parser to parse WLA syntax assembly code.

12.7 HUC6280

Read the subsection [6502](#) as the information applies also to HUC6280 coding. . .

Opcodes that make relative label references:

```
BCC
BCS
BEQ
BMI
BNE
BPL
BVC
BVS
BSR
BBR*
BBS*
```

12.8 SPC-700

Note that you'll have to put an exclamation mark before a 16-bit value. For example,

```
CALL !Main
AND  A, !$1000
```

Opcodes that make relative label references:

```
BCC
BCS
BEQ
BMI
BNE
BPL
BVC
BVS
BRA
BBS
BBC
CBNE *
DBNZ *
```

12.9 Pocket Voice (GB-Z80)

Pocket Voice uses its own MBC. You can enable Pocket Voice mode by selecting Pocket Voice cartridge type (\$BE in \$0147) and defining correct .ROMBANKMAP and .MEMORYMAP. In PV mode bank 0 is 24KB and the rest are 8KB.

Note that WLA assumes that ROM offset is all the time 0. If you use something else as the offset, make sure to compute the jumps by hand as WLA cannot do that.

Check out `tests/gb-z80/include/pocket_voice.i` for more information.

12.10 GB-Z80

WLA outputs only \$10 when it decodes STOP. Often it's necessary to put an extra NOP (\$00) after a STOP, and sometimes something else, but that's left entirely to the user.

Opcodes that make relative label references:

JR *

CHAPTER 13

WLA Flags

Here are short descriptions for the flags you can give to WLA:

You can supply WLA with some (or all or none) of the following option flags.

-h	Assume that all label references are 16-bit by default (size hints still work). Without this flag it's assumed that label references are 8-bit unless otherwise specified.
-i	Add list file information. Adds extra information to the output so WLALINK can produce list files.
-M	WLA generates makefile rules describing the dependencies of the main source file. Use only with flags <code>o</code> and <code>l</code> .
-q	Quiet mode. <code>.PRINT*</code> -directives output nothing.
-s	Don't create <code>_sizeof_*</code> definitions.
-t	Test compile. Doesn't output any files.
-v	Verbose mode. Shows a lot of information about the compiling process.
-x	Extra compile time labels & definitions. WLA does extra work by creating few helpful definitions, and labels <code>SECTIONSTART_[section name]</code> and <code>SECTIONEND_[section name]</code> at the beginning and end of a section.
-D	Declare a definition.

One (and only one) of the following command flags must be defined.

-l	Output a library file.
-o	Output an object file.

You may also use an extra option to specify the include directory. WLA will search this directory for included files before defaulting to the specified `.INCDIR` or current working directory.

-I	Directory to include files.
-----------	-----------------------------

Examples:

```
[seravy@localhost tbp]# wla -D VERSION=255 -x -v -i -o testa.o testa.s
[seravy@localhost tbp]# wla -M -o testa.o testa.s
[seravy@localhost tbp]# wla -D VERSION=$FF -D MESSAGE=\"Hello world\" -l testb.lib_
↪testb.s
[seravy@localhost tbp]# wla -I ../../include -l testb.lib testb.s
[seravy@localhost tbp]# wla -M -I myfiles -l testa.lib testa.s
```

Note that the first example produces file named `testa.o`.

Extra compile time definitions

When you supply WLA with the flag `x` it will maintain few useful definitions and labels while compiling your source codes. Please use the enhanced error reporting engine (so don't use flag `f`) in conjunction with flag `x` as some of the definitions require extra information about the flow of the data which isn't available when using the old, crippled error reporting engine.

Here's a list of definitions you get when you use flag `x`:

WLA_FILENAME	A string definition holding the file name WLA is currently processing.
WLA_TIME	A string definition holding the calendar time (obtained using C's <code>ctime()</code>).
WLA_VERSION	A string definition holding the version number of WLA.

So you can do for example something like

```
.DB WLA_TIME
```

to store the time when the build process started into the ROM file you are compiling.

Definition `CADDR`, which is present without supplying the flag `x`, contains the current 16-bit memory address. So

```
LD HL, CADDR
```

will load the address of the operand data into registers H and L.

CAVEAT: Remember when using defines that `CADDR` gets the address of the place where the definition is used, not the address of the definition, which contains the `CADDR`.

Note that you'll also get all these definitions in lower case (e.g., `wla_filename`).

But that is not all. You will also get `SECTIONSTART_[section name]` labels that are inserted into the start of every section, and `SECTIONEND_[section name]` labels that are inserted into the end of every section.

CHAPTER 15

Good things to know about WLA

- Is 255 bytes too little for a string (file names, labels, definition labels, etc)? Check out `MAX_NAME_LENGTH` in `shared.h`.
- WLA preprocessor doesn't expand macros and repetitions. Those are actually traversed in the assembling phase.
- WLA's source code is mainly a huge mess, but `WLALINK` is quite well structured and written. So beware!
- To get the length of a string e.g. "peasoup", write "peasoup".length.
- Do not write `.E` into your sources as WLA uses it internally to mark the end of a file.

WLA DX's architectural overview

The two most important executables inside WLA DX are WLA (the assembler) and WLALINK (the linker).

16.1 WLA

WLA has four separate phases, called a little bit incorrectly passes:

1. `pass_1.c: pass_1()`:
 - The biggest data processor in WLA.
 - Includes the include files: every time the file is read in, white space is removed, lines formatted, etc.
 - Macros are processed along with directives
 - All textual data, code, etc. are transformed into WLA's internal byte code that gets written into a tmp (TMP) file, and after this phase the assembler or the linker has no idea of target CPU's opcodes - all is just pure WLA byte code.
 - The first and the only pass that handles the assembly source files supplied by the user.
 - The parser in this pass starts from the first byte of the first source file, then moves forward parsing everything that it encounters, but when a macro is called, the parser jumps to the beginning of the macro, and continues parsing from there.
2. `pass_2.c: pass_2()`:
 - If the user has issued directives like `.SDSCTAG`, here we generate the needed data and write that into TMP.
3. `pass_3.c: pass_3()`:
 - Here we read in TMP and do some sanity checks for the data, give labels addresses (if possible), generate internal structures for labels and sections.
4. `pass_4.c: pass_4()`:

- Again we read in TMP.
- Now we check that if there is a reference to a calculation, and that calculation has been successfully calculated, then we can replace the reference with the result.
- This phase writes out object and library files, i.e., transforms TMP to final output files (this write out could actually be `pass_5`)...

16.2 WLALINK

WLALINK is much simpler and more straight forward than WLA; WLALINK just reads in all the objects and library files, places the sections along with labels into the target memory map, solves pending calculations, calculates checksums, and writes out the final ROM/PRG files. `wlalink/main.c:main()` should quite clearly display all the higher level phases in the linking process.

CHAPTER 17

WLA Symbols

Symbols can be optionally generated as a part of the assembly and link steps. With a compatible emulator, this can provide extra information for debugging a ROM, or otherwise help in understanding how it operates.

The symbols file can be generated by `wlalink` by adding “-S” onto the command line. This will output labels, definitions, and some other rudimentary data. Most prominently, this can be used to understand where the ROM output various sections such as subroutines and data, and be able to look that up in the emulator’s ROM or RAM space.

Extra information for address-to-line mapping can be provided by adding the following command line arguments: - Run object generation (e.g. “`wla-65816`”) with “-i” to include list data in the output obj files - Run `wlalink` with “-S -A” to generate symbols with information related to address-to-line mapping

Address-to-line mappings includes information to relate lines in the source files to individual instructions in the generated ROM. This can be used to provide richer disassembly in the emulator, or allow for rich debugging in an external IDE.

17.1 Information For Emulator Developers

In order to properly support loading of WLA symbol files, it is recommended to follow this specification below, especially so as to gracefully support future additions to the symbol files.

- The file should be read one line at a time
- Any text on a line following a `;` should be ignored
- Lines matching `[\S+]` in regex or `[%s]` in scanf code are section headers, and represent a new section. Note that no section data will start with `.`
- Lines following the section header are the data for that section. If you’re acknowledging the section, utilize that section’s specific formatting. Read lines that match until a new section header is encountered.
- Unless otherwise specified, none of the data in any section should be assumed to be sorted in any particular way.

The following are the list of currently supported sections, what they mean, and how their data should be interpreted.

17.1.1 [labels]

This is a list of all Labels to sections of the ROM, such as subroutine locations, or data locations. Each line lists an address in hexadecimal (bank and offset) and a string associated with that address. This data could be used, for example, to identify what section a given target address is in, by searching for the label with the closest address less than the target address.

- Regex match: `[0-9a-fA-F]{2}:[0-9a-fA-F]{4} .*`
- Format specifier: `%2x:%4x %s`

17.1.2 [definitions]

This is a list of various definitions provided in code - or automatically during WLA's processing - and values associated with them. Most prominently, WLA outputs the size of each section of the ROM. Each line lists an integer value in hexadecimal, and a string (name) associated with that value.

- Regex match: `[0-9a-fA-F]{8} .*`
- Format specifier: `%8x %s`

17.1.3 [breakpoints]

This is a list of hexadecimal ROM addresses where the `.BREAKPOINT` directive was used in the source assembly. Each line lists an address in hexadecimal (bank and offset).

- Regex match: `[0-9a-fA-F]{2}:[0-9a-fA-F]{4}`
- Format specifier: `%2x:%4x`

17.1.4 [symbols]

This is a list of hexadecimal ROM addresses where the `.SYMBOL` directive was used in the source assembly. Each line lists an address in hexadecimal (bank and offset) and a string associated with that address.

- Regex match: `[0-9a-fA-F]{2}:[0-9a-fA-F]{4} .*`
- Format specifier: `%2x:%4x %s`

17.1.5 [source files]

These are used to identify what files were used during the assembly process, especially to map generated assembly back to source file contents. Each line lists a hexadecimal file index, a hexadecimal CRC32 checksum of the file, and a file path relative to the generated ROM's root. This could be used to load in the contents of one of the input files when running the ROM and verifying the file is up-to-date by checking its CRC32 checksum against the one generated during assembly.

- Regex match: `[0-9a-fA-F]{4} [0-9a-fA-F]{8} .*`
- Format specifier: `%4x %8x %s`

17.1.6 [rom checksum]

This is just a single line identifying what the hexadecimal CRC32 checksum of the ROM file was when the symbol file was generated. This could be used to verify that the symbol file itself is up-to-date with the ROM in question. This checksum is calculated by reading the ROM file's entire binary, and not by reading any platform-specific checksum value embedded in the ROM itself.

- Regex match: `[0-9a-fA-F]{8}`
- Format specifier: `%8x`

17.1.7 [addr-to-line mapping]

This is a listing of hexadecimal ROM addresses (bank and offset) each mapped to a hexadecimal file index and hexadecimal line index. The file index refers back to the file indices specified in the `source files` section, so that the source file name can be discovered. This information can be used to, for example, display source file information in line with disassembled code, or to communicate with an external text editor the location of the current Program Counter by specifying a source file and line instead of some address in the binary ROM file.

- Regex match: `[0-9a-fA-F]{2}:[0-9a-fA-F]{4} [0-9a-fA-F]{4}:[0-9a-fA-F]{8}`
- Format specifier: `%2x:%4x %4x:%8x`

CHAPTER 18

Legal Note

WLA DX (the whole package) was originally written by Ville Helin in 1998-2008. After that everybody has been able to take part in the development of WLA DX, and recently via GitHub. The authors are not responsible for anything the software does.

WLA DX is GPL software. For more information about GPL, take a look at the `LICENCE` file.

Game Boy and Game Boy Color are copyrighted by Nintendo.

Pocket Voice is copyrighted by Bung HK.

19.1 SYNOPSIS

wlalink [OPTIONS] LINK_FILE OUTPUT_FILE

19.2 OPTIONS

-b	Program file output
-d	Discard unreferenced sections
-i	Write list files (Note: WLA needs <code>-i</code> as well)
-r	ROM file output (default)
-s	Write also a NO\$GMB/NO\$SNES symbol file
-S	Write also a WLA symbol file
-A	Add address-to-line mapping data to WLA symbol file
-v	Verbose messages
-L LIBDIR	Look in LIBDIR for libraries before looking in CWD
-t TYPE	Output type (supported types: 'CBMPRG')
-a ADDR	Load address for CBM PRG

Choose one:

-b OUT	Program file linking
-r OUT	ROM image linking

19.3 DESCRIPTION

wlalink(1) is a part of WLA-DX. It links one or more object files (and perhaps some library files) together to produce a ROM image / program file.

LINK_FILE is a text file that contains information about the files you want to link together. Here's the format:

1. You must define the group for the files. Put the name of the group inside brackets. Valid group definitions are

```
[objects]
[libraries]
[header]
[footer]
[definitions]
```

2. Start to list the file names.

```
[objects]
main.o
vbi.o
level_01.o
...
```

3. Give parameters to the library files:

```
[libraries]
bank 0 slot 1 speed.lib
bank 4 slot 2 map_data.lib
...
```

Here you can also use `base` to define the 65816 CPU bank number (like `.BASE` works in WLA):

```
[libraries]
bank 0 slot 1 base $80 speed.lib
bank 4 slot 2 base $80 map_data.lib
...
```

You must tell WLALINK the bank and the slot for the library files.

4. If you want to use header and/or footer in your project, you can type the following:

```
[header]
header.dat
[footer]
footer.dat
```

5. If you want to make value definitions, here's your chance:

```
[definitions]
debug 1
max_str_len 128
start $150
...
```

If flag `-i` is given, *wlalink*(1) will write list files. Note that you must compile the object and library files with `-i` flag as well. Otherwise *wlalink*(1) has no extra information it needs to build list files.

Here is an example of a list file: Let's assume you've compiled a source file called `main.s` using the `-i` flag. After you've linked the result also with the `-i` flag *wlalink*(1) has created a list file called `main.lst`. This file contains

the source text and the result data the source compiled into. List files are good for debugging.

Make sure you don't create duplicate labels in different places in the memory map as they break the linking loop. Duplicate labels are allowed when they overlap each other in the destination machine's memory.

19.4 EXAMPLES

```
wlalink -r linkfile testa.sfc
wlalink -d -i -b linkfile testb.sfc
wlalink -v -S -L ../../lib linkfile testc.sfc
```


20.1 SYNOPSIS

```
wla-6502 [OPTIONS] SRC_FILE
wla-6510 [OPTIONS] SRC_FILE
wla-65816 [OPTIONS] SRC_FILE
wla-65c02 [OPTIONS] SRC_FILE
wla-6800 [OPTIONS] SRC_FILE
wla-6801 [OPTIONS] SRC_FILE
wla-6809 [OPTIONS] SRC_FILE
wla-8008 [OPTIONS] SRC_FILE
wla-8080 [OPTIONS] SRC_FILE
wla-gb [OPTIONS] SRC_FILE
wla-huc6280 [OPTIONS] SRC_FILE
wla-spc700 [OPTIONS] SRC_FILE
wla-z80 [OPTIONS] SRC_FILE
```

20.2 OPTIONS

-D DEFINE(=VAR) Define DEFINE with value VAR (VAR is optional)

-i	Add list file information
-I DIR	Add Include directory
-q	Quiet mode (.PRINT*-directives output nothing)
-v	Test compile (Don't output any files)
-x	Extra compile time definitions

Choose one:

-o OUT	Output an object file
-l OUT	Output an library file

20.3 DESCRIPTION

Assemble a BIN_FILE to an object file (-o) or to an library file (-l).

These object files can be linked together (or with library files) later with *wlalink(1)*.

Name object files so that they can be recognized as object files. Normal suffix is .o (WLA default). This can also be changed with .OUTNAME.

Name these files so that they can be recognized as library files. Normal suffix is .lib (WLA default).

With object files you can reduce the amount of compiling when editing small parts of the program. Note also the possibility of using local labels (starting with _).

With library files you can reduce the amount of compiling. Library files are meant to hold general functions that can be used in different projects. Note also the possibility of using local labels (starting with _). Library files consist only of FREE sections.

Note: When you compile objects, group 1 directives are saved for linking time, when they are all compared and if they differ, an error message is shown. It is advisable to use something like an include file to hold all the group 1 directives for that particular project and include it to every object file.

If you are interested in the WLA object file format, take a look at the file `txt/wla_file_formats.txt` which is included in the release archive.

20.4 EXAMPLES

```
wla-gb -D DEBUG -D VERBOSE=5 -D NAME = "math v1.0" -o math.o math.s
```

- -D IEXIST
- -D DAY=10
- -D BASE = \$10
- -D NAME=elvis

21.1 SYNOPSIS

```
wlab -[ap]{bdh} BIN_FILE
```

21.2 OPTIONS

- a** Print the address (relative to the beginning of the data).
- p** Don't print file header.

Choose one:

- b** Output data in binary format.
- d** Output data in decimal format.
- h** Output data in hexadecimal format.

21.3 DESCRIPTION

wla (1) converts binary files to WLA's byte definition strings and print it to the standard output.

21.4 EXAMPLES

```
wlab -da gayskeletor.bin > gayskeletor.s
wlab -bap iscandar.bin > iscandar.s
wlab -h starsha.bin > starsha.s
```