

LhTools

Version 0.7.5

**Toolbox, assembler, disassembler, BASIC compiler
and decompiler for
the SHARP PC-1500/A and TRS80 PC-2**

Copyright 1992 - 2015 - Christophe Gottheimer

The **lhTools** are a tool box for assembling, disassembling, build BASIC basic binaries and “*decompiling*” the programs BASIC or ML of the SHARP PC-1500/A and TANDY PC-2.

The **lhTools** are **copyright 1992-2015 Christophe Gottheimer <cgh750215@gmail.com>**

This code is distributed under the terms of the **GNU Public License (GPL) version 2.**

This version is still in pre-alpha release. It is not fully mature and bugs are present.

```
+-----+
|          DISCLAIMER          |
| YOU USE THIS CODE AT YOUR OWN RISK ! I AM NOT |
| RESPONSIBLE FOR ANY DAMMAGE OR ANY DATA LOST OR |
| CORRUPTED BY USING THIS SOFTWARE OR BY USING THE |
| BINARY IMAGES CREATED WITH THIS SOFTWARE WHILE |
| RUNNING THEM ON A SHARP PC-1500/A or TANDY PC-2. |
| BE SURE TO SAVE YOUR IMPORTANT DATA OR PROGRAMS |
| BEFORE LOADING AND RUNNING THE BINARY IMAGES. |
+-----+
```

It is composed of 4 utilities:

- **lhasm** The assembler and basic compiler to produce a binary image from BASIC assembler, or hexadecimal dump sources.
- **lhdump** The disassembler, BASIC decompiler and hexadecimal dumper, working on a binary image.
- **lhcom** The serial sender/receiver to transfer programs and data with the CE-158 serial interface.
- **lhpoke** The utility to transform a binary program into a BASIC program using **POKE**.

1/ Understanding the FRAGMENT concept

When building an image, all types of data are mixed: LM code, byte values, word values, text strings, BASIC code,... The fragments concept will organize these data to be assembled but also disassembled with good format.

For example, the following source **fr1.asm**. Do a

```
lhasm -T -F fr1.frag fr1.asm
```

```
.ORIGIN: 02FE
.CODE
CALL BEEP1
RET

.TEXT
"BIP BIP"

.BYTE
00 00
.END
```

If you try do disassemble with **lhdump -c 2fe fr1.bin** you get:

```
02FE BE E6 69          CALL BEEP1
0301 9A                RET
0302 42                DEC C
0303 49 50             AND (BC),50
0305 20                SBC L
0306 42                DEC C
0307 49 50             AND (BC),50
0309 00                SBC C
030A 00                SBC C
```

Of course, the code from **&02FE** to **&0301** is disassembled, but after, the text and bytes values are interpreted as code.

So, look to the fragment file **fr1.frag**:

```
.FRAGMENTS: 02FE
CODE 02FE
CODE 02FE
TEXT 0302
BYTE 0309
```

Now call the disassembler with the fragment file: **lhdump -F fr1.frag fr1.bin**

```
02FE BE E6 69          CALL BEEP1
0301 9A                RET

0302 "BIP BIP"

0309 00 00
```

The fragments are more less like the sections of the ELF files. In this version of **lhTools**, the following fragments are usable:

```
.BASIC BASIC program
.CODE Assembly instructions
.BYTE Byte (8-bits) values
```

.WORD	Word (16-bits) values [big endian]
.LONG	Long (32-bits) values [big endian]
.TEXT	Text strings of ASCII characters
.KEYWORD	BASIC keyword table
.VAR	Dynamic BASIC variables (not supported)
.XREG	Xreg registers (not supported)
.RESERVE	Reserve area (decoded by lhdump , but not encoded by lhasm)
.HOLE	Obscure data

When calling **lhasm**, the option **-F <fragfile>** will write the fragments descriptors to the file **<fragfile>**. This file may be given after to **lhdump** with the option **-F <fragfile>** to produce the disassembled listing.

The fragment file is a text file, with the header **.FRAGMENTS: <origin address>** and a descriptor **<fragment type> <start-address>** by line. Several descriptors may be specified. **<The fragment type>** is one of the following: **BASIC, CODE, BYTE, WORD, LONG, TEXT, KEYWORD, RESERVE, VARIABLE, XREG, HOLE**. The fragment begins at the **<start-address>** given up to the next fragment address or the end of the binary file.

For example, the following fragment file:

```
.FRAGMENTS: 40c5
CODE 40c5
BYTE 40f0
BASIC 4100
```

describe a LM program from **&40C5** to **&40EF**, a byte data area from **&40F0** to **&40FF** and the remain for a BASIC program. Note that BASIC fragment are decompiled.

Imagine the following binary **hello.bin**:

```
00 0A 10 F0 97 22 48 45 4C 4C 4F 20 57 4F 52 4C
44 22 0D 00 14 04 F1 82 31 0D 00 1E 03 F1 8E 0D FF
```

Create the fragment file **hello.frag** as follow:

```
.FRAGMENTS: 40c5
BASIC 40c5
```

Do **lhdump -F hello.frag hello.bin** and discover:

```
10 PRINT "HELLO WORLD"
20 BEEP 1
30 END
```

The original fragment may also be specified as argument for **lhdump**:

```
lhdump -B 40c5 hello.bin
```

will do the same as above.

2/ lhasm - Assembler and BASIC builder

Usage: lhasm [-h] [-v] [-d|-ddebug] [-dverbose] [-i] [-E[E]] [-W]
[-A argument=substitute...] [-D symbol=value...] [-a] [-na]
[-T|-L logfile] [-nc] [-ns] [-r resymfil] [-s resrcfile]
[-F fragfile] [-K[K][E] keywordfile] [-M macfile] [-S symfile]
[-m machine] [-O origin] [fragment] [-I includepath]
[-Z[header=type]] [-Z[name=headername]] [-Z[entry=startup]]
[-J[loop=n]] [-l] [-N|-no] [-x] [-o outfile] srcfile

where:

-l Do assembler pass 1 only and stop
-a All symbols treated as local except if .EXPORT: is specified
-d|-debug Show debug information
-dverbose Enable verbose mode
-h This help
-i Immediate one pass only assembler. Read from stdin
Incompatible with -J
-m machine Select the machine and modules
-m pc1500 RAM=&4000..&47FF ROM=&8000..&FFFF; This is the default
-m pc1500A RAM=&4000..&57FF ROM=&8000..&FFFF LM=&7C01..7FFF
-m pta4000+16 RAM=&0000..&47FF ROM=&8000..&FFFF
-m pc1560 RAM=&0000..&8000..&FFFF
-m ce151 Add &1000 at end of RAM; This is CE-151
-m ce155 Add &800 before RAM and &1800 at end of RAM; This is CE-155
-m ce159 Add &1000 before RAM; This is CE-159
-m ce161 Add &4000 before RAM; This is CE-161
-m ce163 Add &4000 before RAM; This is CE-163, bank1 is not supported
Only one -m ce... option may be given
-na Disable local symbols list into log file (with -T or -L)
-nc Disable comment copy into log file (with -T or -L)
-ns Disable symbols/variables list into log file (with -T or -L)
-o outfile Output binary code into outfile (.bin)
-r resymfile Input file of addresses to be re-symbol'ed
-s srcfile Output file (.asm) of the full source with re-symbols
-v Show version and exit
-x Output hexadecimal dump instead of binary into outfile (.hex)
-A argument=substitute Replace argument by substitute when defining a symbol
Several -A may be given if several symbols need to be defined
-D symbol=value Define the specified symbol to the given value
Several -D may be given if several symbols need to be defined
-E Warnings treated as errors
-EE Errors are fatal
-F fragfile Output fragments to fragfile (.frag)
-I includepath Add the <includepath> to the directories list where to search
the files to include
-J Replace JR by JP if displacement > 255. Incompatible with -i
-Jloop=n Like -J, but run only n optimization loops
-K keywordfile Output the BASIC keywords table to keywordfile (.keyw)
-KK keywordfile Output the BASIC keywords table to keywordfile (.keyw)
with old format < 0.6.0
-K[K]E keywordfile Export declared BASIC keywords to keywordfile (.keyw)
(with old format < 0.6.0 if option is -KKE)
-L logfile Output logs of the assembler processing into logfile (.log)
This option is exclusive with the -T option
-M macfile Output defined macros to macfile (.mac)
-N|-no Do not output generated binary code
-O origin Set origin address to specified value
-S symfile Output declared symbols to symfile (.sym)
-T Enable trace mode output to stderr.
This is exclusive with the -L log option
-W Enable low priority warnings
fragment Set original fragment
-B BASIC fragment; This is the default
-R RESERVE fragment
-X XREG fragment
-V dynamic VARIables fragment
-c CODE fragment
-b BYTE (8-bits) fragment
-w WORD (16-bits) fragment
-l LONG (32-bits) fragment
-t TEXT fragment

```
-k  KEYWORD fragment
-H  HOLE fragment
-Z          Add a CE158 CSAVE header
-Zname=name Set <name> as CSAVE name
-Zentry=entry Set <entry> as CSAVE startup routine
-Zheader=type Set <type> for CSAVE header
           with <type>: CSAVE CSAVEM CSAVER or PRINT
```

note:

```
If srcfile has a .bas extension, the BASIC mode is assumed.
If srcfile has a .asm, .as or .s extension, the CODE mode is assumed.
If srcfile has a .hex or .x extension, the BYTE mode is assumed.
If the -o outfile argument is not specified, <srcfile>.bin is used for output
```

A special option **-v#** is available for script. It return the version of the **lhTools** on the form **X.Y.Z[*PT*]**, i.e. **0.7.5** for this revision. If a patch level is present, the version will be **0.7.5p5**.

When **lhasm** exits, the following code are returned:

- **0** success,
- **1** error while parsing options or opening specified files,
- **127** several errors encountered inside the source code. The assembler has aborted and no binary code is generated,
- **255** fatal error raised. The assembler has asserted immediately. The fragment, symbol, keyword files are meaningless.

2.1/ Mnemonics

The following mnemonics are understood by **lhasm**:

ADC[#]	(R)	LDD[#]	(R)
ADC	r1	LDI[#]	(R)
ADC	rh	LDI	
ADC[#]	(mn)	LD	r1, n
ADC	n	LD	rh, n
ADD	R	LD	BC, R
ADD[#]	(R), n	LD	BC, PC
ADD[#]	(mn), n	LD	BC, SP
AND[#]	(R), n	LD	R, BC
AND[#]	(R)	LD	SP, BC
AND[#]	(mn), n	LD	PC, BC
AND[#]	(mn)	LD	SP, mn
AND	n	NOP	
AEX		OR[#]	(R), n
ATP		OR[#]	(R)
AMO		OR[#]	(mn), n
AM1		OR[#]	(mn)
BIT[#]	(R), n	OR	n
BIT[#]	(R)	OFF	
BIT[#]	(mn), n	POP	A
BIT[#]	(mn)	POP	R
BIT	n	PUSH	A
CALL	mn	PUSH	R
CDV		RCF	
CLA		RDP	
CPA[#]	(R)	RET	
CPA	r1	RL	
CPA	rh	RR	
CPA[#]	(mn)	RLD[#]	
CPA	n	RRD[#]	
CPI		RPU	
CP	r1, n	RPV	
CP	rh, n	RTI	
DADC[#]	(R)	SBC[#]	(R)
DEC	A	SBC	r1
DEC	r1	SBC	rh
DEC	rh	SBC[#]	(mn)
DEC	R	SBC	n
DI		SBR	(n)
DJC	d	SBR	cc, (n)
DSBC[#]	(R)	SCF	
EI		SDP	
HALT		SL	
INC	A	SR	
INC	r1	SPU	
INC	rh	SPV	
INC	R	STA[#]	(R)
INA		STA	r1
JR	cc, d	STA	rh
JR	d	STA	F
JP	mn	STA[#]	(mn)
LDA[#]	(R)	STD[#]	(R)

LDA	r1	STI[#]	(R)
LDA	rh	XOR[#]	(R)
LDA	F	XOR[#]	(mn)
LDA[#]	(mn)	XOR	n
LDA	n		

These mnemonics are aliases and provided as a standalone instruction to help in coding.

LDW		:= SBR (&C0)	ERRH	:= SBR (&E0)
LJNE	k, d	:= SBR (&C2)	RST	:= SBR (&E2)
JNE	k, d	:= SBR (&C4)	ERR1	:= SBR (&E4)
BKW		:= SBR (&C6)	LYX	:= SBR (&E6)
LJNES	d	:= SBR (&C8)	NORM	:= SBR (&E8)
STS	(n)	:= SBR (&CA)	SLX	:= SBR (&EA)
LDS	(n)	:= SBR (&CC)	CLX	:= SBR (&EC)
VAR	n, d	:= SBR (&CE)	ADN	:= SBR (&EE)
INTG	n, d	:= SBR (&D0)	SXY	:= SBR (&F0)
ARG	d, n	:= SBR (&D2)	CLS	:= SBR (&F2)
STB	n	:= SBR (&D4)	LDU	(mn) := SBR (&F4)
LDB	n	:= SBR (&D6)	STU	(mn) := SBR (&F6)
IFC		:= SBR (&D8)		
STVP		:= SBR (&DA)		
LDPT		:= SBR (&DC)		
EVAL	d	:= SBR (&DE)		

These instructions are aliases and are provided for backward compatibility:

OUTA		:= ATP	SWA	:= AEX
RSET		:= OFF	SWP	:= AEX
STA	T0	:= AM0	SLD[#]	:= RLD
STA	T1	:= AM1	SRD[#]	:= RRD
STI		:= LDI		

Convention for the mnemonics describe above:

- n** Byte 8-bits value, within **0..255 (&FF)**
- mn** Word 16-bits value, within **0..65535 (&FFF)**
- (n)** Indirect 8-bits value, within **0..255 (&FF)**
- (mn)** Indirect 16-bits value, within **0..65535 (&FFFF)**
- cc** Condition: **C, NC, V, NV, Z, NZ, V, NV, ==, !=, <, >=**
- d** 8-bits displacement, within **0..255**
- rh** High 8-bits register: **B, D, H, M**
- r1** Low 8-bits register: **C, E, L, N**
- R** Whole 16-bits register: **BC, DE, HL, MN**
- (R)** Indirect whole 16-bits register: **(BC), (DE), (HL), (MN)**
- A** Accumulator
- F** Flags (status)
- PC** Program counter

SP	Stack pointer
TO	Timer with 9th-bit to 0
T1	Timer with 9th-bit to 1
[#]	Optional second page access
k	BASIC keyword code if k \geq &E000 else a 8-bit value is assumed

Inside the **CODE** fragment, some special mnemonics are understood:

- **BYTE** *n1* [*n2...*] enter the 8-bits value(s) of *n1* [*n2...*]
- **WORD** *mn1* [*mn2...*] enter the 16-bits values of *mn1* [*mn2...*]
- **TEXT** "*string*" enter the normal string between double-quote
- **LENGTH** "*string*" enter the 8-bits length of the string
- **LENGTH** *arg* enter the 8-bits length of the *arg* parameter
- **STRINGIFY** *arg* enter a string image of the *arg* parameter
- **STRINGIFY VALUEOF'***arg* enter a string image of the *arg* parameter value, ie, the parameter *arg* is evaluated and the computed value is "*stringified*". Note that the string will be **nn** for a value \leq **255** and **mmnn** for other values
- **STRINGIFY VALUE8OF'***arg* enter a string image of the **8-bits** *arg* parameter value, ie, the parameter *arg* if evaluated and the computed value is "*stringified*"
- **STRINGIFY VALUE16OF'***arg* enter a string image of the **16-bits** *arg* parameter value, ie, the parameter *arg* if evaluated and the computed value is "*stringified*"
- **STRINGIFY KEYWORDOF'***code* enter a keyword image of the *code* parameter value. The string is the same as the keyword defined by **.DEFINE:** or the builtin BASIC keywords
- **STRINGIFY KEYWORD3OF'***code* enter a 3-characters keyword image of the *code* parameter value. The string is composed from; the first 3-characters of the keyword defined by **.DEFINE:** or the builtin BASIC keywords
- **STRINGIFY MONTHOF'***arg* enter a 3-characters month image of the *arg* parameter value for a value between **1** and **12**

Look to the following example **mac.as**:

```

.ORGIN: 40C5
.CODE

;; A variable SHOULD be initialized
%00h .EQU

.MACRO: DOIT
    WORD __#0
__#0 .EQU [-2].
    BYTE __#1
    LENGTH __#2
    STRINGIFY __#2
    __#2 .EQU .
.ENDMACRO

DOIT %00h 00 SetCRet
SCF
RET

DOIT %00h 00 ClrCRet
RCF
RET

```

.END

Calling **lhasm -T mac.as** will do:

```
1
2      .ORIGIN:      40C5
3      .CODE 40C5
4 40C5      ;;      A variable SHOULD be initialized
5 40C5      %00h    .EQU 0000
7          .MACRO:      DOIT
7          {
8 ; DOIT: 1          WORD ___#0
9 ; DOIT: 2          ___#0 .EQU  [-2].
10 ; DOIT: 3         BYTE ___#1
11 ; DOIT: 4         LENGTH ___#2
12 ; DOIT: 5         STRINGIFY ___#2
13 ; DOIT: 6         ___#2 .EQU .
14          }
14          .ENDMACRO      ; DOIT
16          DOIT  %00h 00 SetCRet
16          {
16 40C5      00 00          WORD  %00h
16 40C7      %00h .EQU 40C5
16 40C7      00          BYTE  00
16 40C8      07          LENGTH SetCRet
16 40C9      53 65 74 43 52  STRINGIFY  SetCRet
16          65 74
16 40D0      SetCRet:    .EQU 40D0
16          }
17 40D0      FB          SCF
18 40D1      9A          RET
20          DOIT  %00h 00 ClrCRet
20          {
20 40D2      40 C5          WORD  %00h
20 40D4      %00h .EQU 40D2
20 40D4      00          BYTE  00
20 40D5      07          LENGTH ClrCRet
20 40D6      43 6C 72 43 52  STRINGIFY  ClrCRet
20          65 74
20 40DD      ClrCRet:    .EQU 40DD
20          }
21 40DD      F9          RCF
22 40DE      9A          RET
24 40DF      .END
```

Inside the **TEXT** fragment, some special mnemonics are understood:

- "*string*" enter the normal string between double-quote. In the string, the following special characters are understood:
 - \\ for a backslash
 - \x<hex> for a character given in hexadecimal, ie **CHR\$ (&<hex>)**
 - c\+80 for setting the bit 7 (**&80**) of the character *c*
 - \xy for a character given by two hex digits, ie **CHR\$ (&xy)**
 - \ins the character **INSERT**, code **&27**, ie **CHR\$ (39)**
 - \pi the character **PI**, code **&5D**, ie **CHR\$ (93)**
 - \sqr the character **SQUAREROOT**, code **&5B**, ie **CHR\$ (91)**
 - \yen the character **YEN**, code **&5C**, ie **CHR\$ (92)**
- **VALUEOF'arg** enter a string image of the *arg* parameter value, ie, the parameter *arg* is evaluated and the computed value is "*stringified*". Note that the string will be **nn** for a value **<= 255** and **mmnn** for other values

- **VALUE8OF'arg** enter a string image of the **8-bits arg** parameter value, ie, the parameter *arg* if evaluated and the computed value is “stringified”
- **VALUE16OF'arg** enter a string image of the **16-bits arg** parameter value, ie, the parameter *arg* if evaluated and the computed value is “stringified”
- **KEYWORDOF'code** enter a keyword image of the *code* parameter value. The string is the same as the keyword defined by **.DEFINE:** or the builtin BASIC keywords
- **KEYWORD3OF'code** enter a 3-characters keyword image of the *code* parameter value. The string is composed fro; the first 3-characters oft the keyword defined by **.DEFINE:** or the builtin BASIC keywords
- **MONTHOF'arg** enter a 3-characters month image of the *arg* parameter value for a value between **1** and **12**

When entering text strings, some special characters may be specified as follow: The characters between the double quote " are interpreted as a text string. To enter a \, do \\. The characters **\pi \yen \sqr \ins** are the ASCII code **&5D, &5C, &5B** and **&39**. Writing **c\+80** will set the 8th bit (**&80**) to 1. The ASCII code may be directly entered by **\mn**, i.e, **\41** for A.

```
.TEXT
"ABcd"      ; This is a normal string
"\5d"      ; is the PI symbol
"C\+80"    ; is C with the 8th bit to 1
"\yen"     ; is the YEN
```

Running **lhasm -T tes.asm** gives:

```
.TEXT C5
C5  42 63 64      "ABcd"      ; This is a normal string
C9  D            "\5d" ; is the PI symbol
CA  C3           "C\+80"    ; is C with the 8th bit to 1
CB  C            "\yen"     ; is the YEN
CC  .END
    ;; 40C5      tes.asm$$._start
    ;; 40CC      tes.asm$$._end
    ;; 0007      tes.asm$$._length
```

2.2/ Base and character specifiers

When specifying an immediate value, the following specifiers are understood:

n	Hexadecimal 8-bits value (2-digits) within 00..FF
&n	Hexadecimal 8-bits value (1 to 2-digits) within &00..&FF
#n	Decimal 8-bits value (1 to 3-digits) within #0..#255
@n	Octal 8-bits value (1 to 3-digits) within @0..@377
\xn	Hexadecimal 8-bits value (1 to 2-digits) within &00..&FF
\un	Decimal 8-bits value (1 to 3-digits) within #0..#255
\on	Octal 8-bits value (1 to 3-digits) within @0..@377
\$c	Character ASCII code of c
mn	Hexadecimal 16-bits value (4-digits) within 0000..FFFF
&mn	Hexadecimal 16-bits value (1 to 4-digits) within &0000..&FFFF
#mn	Decimal 16-bits value (1 to 5-digits) within #0..#65535
@mn	Octal 16-bits value (1 to 5-digits) within @0..@177777
\Xmn	Hexadecimal 16-bits value (1 to 4-digits) within &0000..&FFFF
\Umn	Decimal 16-bits value (1 to 5-digits) within #0..#65535
\Omn	Octal 16-bits value (1 to 6-digits) within @0..@177777

The **.** (dot) means the current address (ie, the address on which the next byte will be written) and is assumed as 16-bits value.

The **..** (dot dot) means the current starting address and is assumed as a 16-bits value.

When **lhasm** assembles an instruction, it fills the pseudo symbols **.** and **..** as follow:

- .** is the address at which the current byte will be written:
JR . will produce a **JR +00 (8E 00)**.
- ..** is the address at which the currently assembled instruction starts:
JR .. will produce a **JR -02 (9E 02)**.

When specifying a character with **\$c**, it also possible to set a character by its full name with following syntax: **:\$<character>** where **<character>** is one of the following:

00	null	40	at
01	soh	41	upper.a
02	stx	42	upper.b
03	etx	43	upper.c
04	eot	44	upper.d
05	enq	45	upper.e
06	ack	46	upper.f
07	bel	47	upper.g
08	bs	48	upper.h
09	tab	49	upper.i
0A	lf	4A	upper.j
0B	vt	4B	upper.k
0C	ff	4C	upper.l
0D	cr	4D	upper.m
0E	so	4E	upper.n
0F	si	4F	upper.o
10	dle	50	upper.p

11	dc1		51	upper.q	
12	dc2		52	upper.r	
13	dc3		53	upper.s	
14	dc4		54	upper.t	
15	nak		55	upper.u	
16	syn		56	upper.v	
17	etb		57	upper.w	
18	can		58	upper.x	
19	em		59	upper.y	
1A	sub		5A	upper.z	
1B	esc		5B	openbracket	squareroot
1C	fs		5C	backslash	yen
1D	gs		5D	closebracket	pi
1E	rs		5E	circumflex	power
1F	us		5F	underscore	
20	space		60	backquote	
21	exclam		61	lower.a	
22	doublequote	stringmark	62	lower.b	
23	sharp		63	lower.c	
24	dollar		64	lower.d	
25	percent		65	lower.e	
26	ampersand	hexmark	66	lower.f	
27	quote	insert	67	lower.g	
28	openparenthesis		68	lower.h	
29	closeparenthesis		69	lower.i	
2A	star	multiply	6A	lower.j	
2B	plus	add	6B	lower.k	
2C	comma		6C	lower.l	
2D	minus	subtract	6D	lower.m	
2E	dot	period	6E	lower.n	
2F	slash	divide	6F	lower.o	
30	zero		70	lower.p	
31	one		71	lower.q	
32	two		72	lower.r	
33	three		73	lower.s	
34	four		74	lower.t	
35	five		75	lower.u	
36	six		76	lower.v	
37	seven		77	lower.w	
38	eight		78	lower.x	
39	nine		79	lower.y	
3A	colon		7A	lower.z	
3B	semicolon		7B	openbrace	
3C	less		7C	verticalbar	
3D	equal		7D	closebrace	
3E	greater		7E	tilde	
3F	question		7F	delete	fullcursor

Note that the characters **squareroot**, **yen**, **pi**, **backquote** and **insert** are specific to the SHARP PC-1500.

2.3/ Operators

When specifying an immediate value, i.e, **d**, **n**, or **mn**, unary operator may precede as follow:

+n	Positive displacement, PC+d
-n	Negative displacement, PC-d
*mn	Offset between current and mn
*_ *mn	Offset between address of the instruction and mn
*_ *mn [pq]	Offset between pq and mn
<mn	High 8-bit from the mn value, ie, m
>mn	High 8-bit from the mn value, ie, n
{mn	Shift 1 bit left
}mn	Shift 1 bit right
!mn	&FFFF XOR'ed 16-bits value
!n	&FF XOR'ed 8-bits value
^n	Set bit n to '1' if n = 1..16 or 0 if n = 0
'mn	First bit to '1' in mn starting from left
/mn	First bit to '1' in mn starting from right
~mn	Swap byte m and n for 16-bits value
~n	Swap digits nH and nL for 8-bits value
[opval]mn	Compute mn op val with the following op :
+	addition
-	subtraction
*	multiplication
/	division
&	logical AND
 	logical OR
^	logical XOR
=	Align on val bits frontier
<	Shift left val bits
>	Shift right val bits
:	Return value of val if val is not 0 , else return mn
?	Return value of val if val exist, else return mn

Note: If **val** start by a ' (**quote**), an expression is assumed and it is evaluated.

flags.f	Return the status flag mask where f is one of H, V, Z, I, C
BCDOF 'mn	Return the BCD value of mn
BCD80F 'n	Return the 8-bits BCD value of n
BCD160F 'mn	Return the 16-bits BCD value of mn
CODEOF "keyw"	Return the code value mn of the keyword " keyw "
OPCODE 'mnemo	Return the code value mn of the mnemonic mnemo . If the opcode stands in second page, the code value will be &FD<opcode> .

When specifying a register, unitary operator may be added as follow:

#<rR	High 8-bits register from rR , ie, B, D, H, M
#>rR	Low 8-bits register from rR , ie, C, E, L, N
#^rR	Whole 16-bits register from rR , ie, BC, DE, HL, MN
#*rR	Indirect 16-bits register from rR , ie, (BC), (DE), (HL), (MN)
	rR is any register: rh, r1, R, (R)

When specifying a condition, unitary operator may be added as follow:

#!cc Return the inverse condition as shown below:

#!cc becomes **Ncc**

#!Ncc becomes **cc**

#!!= becomes **==**

#!== becomes **!=**

#!>= becomes **<**

#!< becomes **>=**

2.4/ Symbols and variables

A symbol is a named value accessible in the source file and may be used at any time. The symbols defined in a source code may be saved into a **.sym** file by using **-S <symfile>** option.

A symbol is declared by setting its name followed by a **:** (colon). Note that no instruction is allowed after a symbol declaration.

With this, the immediate PC value is affected to the symbol.

To define a specific value to a symbol, use **.EQU <value>**.

The name of a symbol should not start with **.**, **** or **%** because these characters are reserved for special usage.

Example:

```
.ORIGIN:    40c5
.CODE
.LOCAL
.EXPORT: DOBEEP1 .EQU e669
    LDA    10
LOOP:
    PUSH  A
    CALL  DOBEEP1
    POP   A
    DEC  A
    JR   NZ,LOOP
    RET
.END
```

Running **lhasm -T te.asm** will give:

```
1
2      .ORIGIN:    40C5
3      .CODE 40C5
4 40C5 DOBEEP1:    .EQU E669
5 40C5 B5 10      LDA    10
6 40C7 LOOP: .EQU 40C7
7 40C7 FD C8      PUSH  A
8 40C9 BE E6 69   CALL  DOBEEP1
9 40CC FD 8A      POP   A
10 40CE DF        DEC  A
11 40CF 99 0A     JR   NZ LOOP
12 40D1 9A        RET
13 40D2 .END
      ;; 40C5      te.asm$$._start
      ;; 40D2      te.asm$$._end
      ;; 000D      te.asm$$._length

.SYMBOLS:
E669 DOBEEP1

FFFF      $$[.LOCAL] 000 te.asm
40C7      000$$LOOP
```


A symbol may be global to the whole source (even in included source) or local to a source file.

If **.EXPORT:** precedes a symbol declaration, this will be global. In a same way, if **.EXPORTALL** is specified in a source file, all symbols are global.

If **.LOCAL** is specified, all symbols defined after, until the **.END** of the file will be treated as local symbols. After **.LOCAL**, use **.EXPORT:** to force a symbol to be defined as global. When calling **lhasm**, the option **-a** force all symbols to be defined as local.

The scope of global symbols is all the source code. The scope of local symbols is only the source file, but not the files included from this source.

The assembler creates some special symbols. These symbols are global. These symbols are:

- `<source name>$$._start` The start address of the source file
- `<source name>$$._end` The end address of the source file
- `<source name>$$._length` The length of the source file

where `<source name>` is the file name of source given to **lhasm** or included through a **.INCLUDE:** directive.

To be independent from the file name, the assembler accepts two symbols in place of `<source name>`:

- `__MAIN__$` The main top source file, i.e, this specified on command line
- `__THIS__$` The current source file parsed

So, it is possible to get the top start address of the binary with `__MAIN__$._start`. The length of the current file parsed is retrieved by `__THIS__$._length`.

When a BASIC `<keyword>` is defined (**.DEFINE:**), the assembler creates some special symbols:

- `<keyword>\\._code` The 2-bytes code used to compile the keyword
- `<keyword>\\._jump` The “jump” address or entry point of the keyword
- `<keyword>\\._bits` The bits of the keyword: the high byte is the ASCII code of the letters (**NPC?**) and the low byte is the corresponding high 4-bits field **&CO, &AO, &80, &EO**. See **.DEFINE:** for an explanation of bits.

If the option **-ns** is given, the symbols are not printed into the `<logfile>`.

If the option **-na** is given, the local symbols are not printed into the `<logfile>`. Also, they will be not saved into the `<symfile>`.

Other special symbols for dealing with date and time (ex: **Oct. 31 2014 23:45:59**):

- `__DATE__$._year` The year of assembler start, ie, **2014**
- `__DATE__$._yy` The year modulo 100 of assembler start, ie, **14**
- `__DATE__$._month` The month of assembler start, ie, **10**
- `__DATE__$._day` The day of assembler start, ie, **31**
- `__DATE__$._ymd` The date on 16-bits: **(yy << 9) | (month<<5) | day**
- `__TIME__$._hour` The hour of assembler start, ie, **23**
- `__TIME__$._minute` The minute of assembler start, ie, **45**
- `__TIME__$._second` The second of assembler start, ie, **59**
- `__TIME__$._hm` The hour /minute on 16-bits: **(hour<<6) | minute**

Other special symbols for dealing with **lhTools** version (ex: version **0.6.99p4**):

- **__VERSION__\$\$.release** The version release, ie, **0**
- **__VERSION__\$\$.major** The version major, ie, **6**
- **__VERSION__\$\$.minor** The version minor, ie, **99**
- **__VERSION__\$\$.patch** The version patchlevel, ie, **4**

When defining a symbol, a part of the symbol value expression may be replaced by a substitution string before evaluating the expression. This feature is given to produce several images from the same file.

First, the substitution string should be declared before the symbol is defined. Two ways exist to deal with substitution strings:

- From the command line, with the option **-A <subname>=<subexpr>**,
- Inside the code with **.SUBSTITUTE: <subname> = <subexpr>**. Also, the existence of a substitution string may be checked with the condition **SUBSTITUTE?**.

The substitution strings are global to the whole source and should be defined only one time. The substitution strings are not values, but expressions to be evaluated when the substitution is performed into a symbol expression.

When the assembler evaluates the expression value for defining a symbol, it looks for the pattern **__/<subname>/** and if it is found, it will be replaced “in the text” by the **<subexpr>**. Note that only one pattern is allowed in an expression value.

For example, the source **tsub.asm**:

```
.IF SUBSTITUTE? LCDPORT
.ELSE
.SUBSTITUTE:      LCDPORT      = [>2]E2
.ENDIF

LCDCMD1      .EQU  __/LCDPORT/00
LCDCMD2      .EQU  __/LCDPORT/02
LCDCMD3      .EQU  __/LCDPORT/04
TEST  .EQU  [+1]~__/LCDPORT/DF

.CODE
LDA#  (LCDCMD1)
STA#  ([+1]LCDCMD2)
RET
.END
```

The substitution symbol is **LCDPORT**. If defined “in the source”, the substitution string is **.**

Running **lhasm -T -N asm/tsub.asm** gives:

```
1 40C5 +TRUE+      .IF  SUBSTITUTE? LCDPORT
2      /false/      .ELSE
3      /false/      .SUBSTITUTE:      LCDPORT      = [>2]E2
4      /false/      .ENDIF
6 40C5 LCDCMD1:    .EQU 3880
7 40C5 LCDCMD2:    .EQU 3880
8 40C5 LCDCMD3:    .EQU 3881
9 40C5 TEST: .EQU B739
11      .CODE 40C5
```

```

12
    .ORIGIN:      40C5
12 40C5 FD A5 38 80      LDA#   (LCDCMD1)
13 40C9 FD AE 38 81      STA#   ([+1]LCDCMD2)
14 40CD 9A              RET
15 40CE .END
    ;; 40C5      asm/tsub.asm$$._start
    ;; 40CE      asm/tsub.asm$$._end
    ;; 0009      asm/tsub.asm$$._length

    .SYMBOLS:
3880 LCDCMD1
3880 LCDCMD2
3881 LCDCMD3
B739 TEST

```

Another example with the substitution symbol **LCDPORT** is now passed from the command line by **lhasm -T -N -A LCDPORT=00 asm/tsub.asm**. With this command line, the option **-A LCDPORT=00** defines the substitution string.

```

1 40C5 +TRUE+          .IF   SUBSTITUTE? LCDPORT
2      /false/         .ELSE
3      /false/         .SUBSTITUTE:      LCDPORT      = [>2]E2
4      /false/         .ENDIF
6 40C5 LCDCMD1:        .EQU 0000
7 40C5 LCDCMD2:        .EQU 0002
8 40C5 LCDCMD3:        .EQU 0004
9 40C5 TEST: .EQU DF01
11      .CODE 40C5
12
    .ORIGIN:      40C5
12 40C5 FD A5 00 00      LDA#   (LCDCMD1)
13 40C9 FD AE 00 03      STA#   ([+1]LCDCMD2)
14 40CD 9A              RET
15 40CE .END
    ;; 40C5      asm/tsub.asm$$._start
    ;; 40CE      asm/tsub.asm$$._end
    ;; 0009      asm/tsub.asm$$._length

    .SYMBOLS:
0000 LCDCMD1
0002 LCDCMD2
0004 LCDCMD3
DF01 TEST

```

A variable is like a symbol but the value of a variable may change within the code. Variables are not saved by the **-S** *<symfile>* option. A variable name starts with **%** and has the form **%mnc** where *mn* is a 2 digits number from **00** to **99** and *c* is lowercase letter from **a** to **z**. A maximum of 2600 variables may be declared.

A variable **SHOULD BE INITIALIZED** before to use it. Referencing a variable without an affectation before will raise an error.

Example:

```
.ORIGIN:    40C5
.CODE
%10a  .EQU  10
      LD    L,%10a
%011:
      AND   (BC),00
      INC   BC
      DJC   %011
      RET
.END
```

Running **lhasm** will give:

```
1
      .ORIGIN:    40C5
2
      .CODE 40C5
3 40C5  %10a  .EQU  0010
4 40C5  6A 10          LD    L %10a
5 40C7  %011  .EQU  40C7
6 40C7  49 00          AND   (BC) 00
7 40C9  44          INC   BC
8 40CA  88 05          DJC   %011
9 40CC  9A          RET
10 40CD  .END
      ;; 40C5      tel.asm$$._start
      ;; 40CD      tel.asm$$._end
      ;; 0008      tel.asm$$._length

      .SYMBOLS:

40C7  %011
0010  %10a
```

The variables are cleared and erased between the passes 1 and 2. So it is not possible to retain a value of a variable from the pass 1 into the pass 2.

2.5/ Using macro

A macro is a part of code to be developed each time it is found in the source.

Imagine we want to have an instruction as **JR >** which does not exist. Just create a macro called **JR>** and when the assembler will find **JR> label** it will expand this code. The parameter label will be passed to the code and substituted according to the macro rules.

A macro is defined by the directive:

```
.MACRO: <name>
```

followed by any code, with the eventual substitution marker and is terminated by

```
.ENDMACRO
```

The substitution markers are on the form **__#n** where **n** is within **0..9**. When the macro is found in the code, the first parameter after the name is **__#0**, the second **__#1**, and so on, until the 10th and last parameter **__#9**.

An example with the macro JR>

```
.MACRO: JR>  
    JR ==,+02 ; If Z values are equal, test is false  
    JR >=, __#0 ; If C, the test is true  
.ENDMACRO
```

As the following source:

```
    LDA 10  
    LD B,09  
    JR> gt  
    RET ; test false  
gt: ; Greater than
```

Will give:

```
1  
1  
2 ; JR>: 1  
3 ; JR>: 2  
4  
4  
6 40C5 B5 10  
7 40C7 48 09  
8  
8  
8 40C9 8B 02  
false  
8 40CB 83 01  
8  
9 40CD 9A  
10 40CE gt: .EQU 40CE  
10 40CE .END
```

```
.MACRO: JR>  
    {  
    JR ==,+02 ; If Z values are equal, test is false  
    JR >=, __#0 ; If C, the test is true  
    }  
.ENDMACRO ; JR>  
    LDA 10  
    LD B 09  
JR> gt  
    {  
    JR == +02 ; If Z values are equal test is  
    JR >= gt ; If C the test is true  
    }  
    RET ; test false
```

By mixing the unary operators and substitution markers, some powerful macros may be defined:

```
.MACRO: LDR  
    LD #<__#0,<__#1 ; rH register loaded with high 8-bits  
    LD #>__#0,>__#1 ; rL register loaded with low 8-bits  
.ENDMACRO
```

The macro **LDR** is no define and will expand code to load the 16-bits value into a whole 16-bits register.

Now, write:

```
LDR HL,8899
```

And see:

```

1          .MACRO: LDR
1          {
2 ; LDR: 1      LD      #<__#0,<__#1 ; rH register loaded with high 8-bits
3 ; LDR: 2      LD      #>__#0,>__#1 ; rL register loaded with low 8-bits
4          }
4          .ENDMACRO ; LDR
5          LDR      HL 8899
5          {
5 40C5 68 88      LD      #<__#0 <__#1 ; rH register loaded with high 8-
bits          LD      #>__#0 >__#1 ; rL register loaded with low 8-bits
5 40C7 6A 99      LD      #>__#0 >__#1 ; rL register loaded with low 8-bits
5          }
5 40C9 .END
5          }
5 40C9 .END

```

When developing complex macros, it is also necessary to have some labels for jumps or addresses related into the macro. Because the macros are re-entrant, the labels should be available only inside the macro. To do this the 10 labels **0: 1: .. 9:** are available inside a macro. Note that the label **x:** should NOT be followed by an instruction.

The macro **XFER** will do a copy in reverse from **BC** to **DE** until **L** is not **&FF**, but stops if the bit 7 (**&80 := ^80**) is set.

```

.MACRO: XFER
LD      B <__#0
LD      C >__#0
LD      D <__#1
LD      E >__#1
LD      L >__#2
1:
LDI     (BC) ; Load A with (BC) and increment BC
STD     (DE) ; Store A to (DE) and decrement DE
BIT     ^08 ; Bit 7 of A is set
JR      C,2: ; Yes ! XFER is finished
DJC     1: ; Decrement L and jump to 1: if not C
2:
RET
.ENDMACRO

```

And to transfer the BASIC **A\$** variable to **&47FF**, do

```
XFER A$ 47FF \u15
```

And see:

```

1          .MACRO: XFER
1          {
2 ; XFER: 1      LD      B <__#0
3 ; XFER: 2      LD      C >__#0
4 ; XFER: 3      LD      D <__#1
5 ; XFER: 4      LD      E >__#1
6 ; XFER: 5      LD      L >__#2
7 ; XFER: 6      1:
8 ; XFER: 7      LDI     (BC) ; Load A with (BC) and increment BC
9 ; XFER: 8      STD     (DE) ; Store A to (DE) and decrement DE
10 ; XFER: 9     BIT     ^08 ; Bit 7 of A is set
11 ; XFER:10     JR      C,2: ; Yes ! XFER is finished

```

```

12 ; XFER:11          DJC   1:   ; Decrement L and jump to 1: if not C
13 ; XFER:12          2:
14 ; XFER:13          RET
15                    }
15                    .ENDMACRO ; XFER
17                    XFER  A$ 47FF \u15
17                    {
17 40C5  48 78          LD    B <__#0
17 40C7  4A C0          LD    C >__#0
17 40C9  58 47          LD    D <__#1
17 40CB  5A FF          LD    E >__#1
17 40CD  6A 0F          LD    L >__#2
17 40CF  1:
; #XFER__00__#1      .EQU 40CF
17 40CF  45            LDI   (BC) ; Load A with (BC) and increment
BC
17 40D0  53            STD   (DE) ; Store A to (DE) and decrement DE

17 40D1  BF 80          BIT   ^08 ; Bit 7 of A is set
17 40D3  83 02          JR    C 2: ; Yes ! XFER is finished
17 40D5  88 08          DJC   1: ; Decrement L and jump to 1: if not
C
17 40D7  2:
; #XFER__00__#2      .EQU 40D7
17 40D7  9A            RET
17                    }
17 40D8  .END

```

2.6/ Using structure

A **structure** is a way to organize the data. It is composed of a set of **fields**. A **field** may be a **byte**, a **word**, a **structure**, or an **array** of byte, word or structure.

The syntax to define a structure is:

```
.STRUCT:  <struct_name>
          <field1>  <type>[,<nb_of_element>]
          . . .
.ENDSTRUCT
```

Where *<type>* is:

text	A set of 8-bits character,
byte	A 8-bits value or a character,
word	A 16-bits value,
long	A 32-bits value,
struct ' <i><name></i>	A structure. Note that a structure can not reference itself.

If an array is needed, just follow the *<type>* by a comma and a value like **#n** for n items in decimal, **[+1]***<symbol>*, etc...

Up to **30** fields may be declared in a structure. Up to **100** structures may be defined. The structures are global to the source, even if they are declared in an included source.

Imagine that a program has to manage a header defined this way:

- A **name** : 11 characters,
- A **type** : 1 byte,
- A **length** : 2 bytes,
- A **pointer** to the previous header : 2 times 2 bytes.

So the structure **basfile_header** is defined as follow:

```
.STRUCT:      basfile_prev
              ptr      word,#2
.ENDSTRUCT

;; 11 bytes for the filename
FILENAMELEN:  .EQU     #11
.STRUCT:      basfile_header
              filename text,FILENAMELEN
              filetype byte
              filelen  word
              fileprev struct 'basfile_prev
.ENDSTRUCT
```

In the example above:

- The structure **basfile_prev** has only one field **ptr** define as an array of 2 words .
- The structure **basfile_header** has 4 fields: the first is an array of 11 bytes for the name, the second is a byte for the type, the third is a word for the length and the last is a structure to **basfile_prev**.

The following functions are available to deal with the structures and the fields:

SIZEOF' < <i>struct</i> <i>field</i> >	Return the whole size,
TYPEOF' < <i>struct</i> <i>field</i> >	Return the size of the base type,
ELEMENTOF' < <i>struct</i> <i>field</i> >	Return the size of the element,
ARRAYOF' < <i>struct</i> <i>field</i> >	Return the number of elements,
OFFSETOF' < <i>struct</i> <i>field</i> >	Return the offset of the field inside the structure.

For fields of type **byte**, **word**, **long** or **text**, **TYPEOF'** returns **1**, **2**, **4**, **1** respectively.
For fields of type **struct'**, **TYPEOF'** returns **1**.

For fields of type **byte**, **word**, **long** or **text**, **ELEMENTOF'** returns **1**. For fields of type **struct'**, **ELEMENTOF'** returns the **SIZEOF'**.

For all fields, **ARRAYOF'** returns the number #*n* specified when declaring the structure, or **1** if only one item is expected.

For all fields, **SIZEOF'** returns the whole size of the field, i.e. **ARRAYOF' * TYPEOF' * ELEMENTOF'**.

For all fields, **OFFSETOF'** returns the global offset inside the structure.

For structure, **SIZEOF'** and **ELEMENTOF'** return the whole size, **OFFSETOF'** returns always **0**, **TYPEOF'** and **ARRAYOF'** return **1**.

When a structure is declared, **lhasm** shows information in the log file:

```

1      .CODE 40C5
2          .STRUCT:   basfile_prev
2          {
3 ; basfile_prev: 1      +0000:0004.0002.0002.0002 ptr
4          }
4          .ENDSTRUCT ; 0004 basfile_prev
5 40C5 FILENAMELEN:   .EQU 000B
6          .STRUCT:   basfile_header
6          {
7 ; basfile_header: 1    +0000:000B.0001.0001.000B filename
8 ; basfile_header: 2    +000B:0001.0001.0001.0001 filetype
9 ; basfile_header: 3    +000C:0002.0002.0002.0001 filelen
10 ; basfile_header: 4   +000E:0004.0001.0004.0001 fileprev
11         }
11         .ENDSTRUCT ; 0012 basfile_header

```

The map information is:

```
+<offsetof>:<sizeof>.<typeof>.<elementof>.<arrayof>    <field>
```

These functions are available as immediate values, inside the mnemonics but also for symbols and variables assignment:

```

LD      C,OFFSETOF'basfile_header.filelen
LD      L,[-1]ARRAYOF'basfile_header.filename
HEADER_LEN .EQU  SIZEOF'basfile_header
ADC     HEADER_LEN

```

In our example, this will give the following code:

```

13 40C5 4A 0C          LD    C OFFSETOF'basfile_header.filelen
14 40C7 6A 0A          LD    L [-1]ARRAYOF'basfile_header.filename
15 40C9 HEADER_LEN: .EQU 0012
16 40C9 B3 12          ADC   HEADER_LEN

```

In this way, writing a source to set the filename field to zero will be:

```

    LDA    OFFSETOF'basfile_header.filename
    ADD    BC
    LD     L,SIZEOF'basfile_header.filename
    DEC   L
    CLA
loop:
    STI    (BC)
    DJC   loop

```

And the assembler code generated gives:

```

18 40CC B5 00          LDA    OFFSETOF'basfile_header.filename
19 40CE FD CA          ADD    BC
20 40D0 6A 0B          LD     L SIZEOF'basfile_header.filename
21 40D2 62              DEC   L
22 40D3 34              CLA
23 40D4 loop: .EQU 40D4
24 40D4 41              STI    (BC)
25 40D5 88 03          DJC   loop

```

Another way to manage the structure, is to use a register as a pointer to a structure or a field with the pseudo instruction **BIND** *<R>*,*<struct | field | R'>*, where register is **BC**, **DE** or **HL**. While a register is bound to a structure, the assembler will “follow” some instructions and update automatically the current field pointed by the register, but also warn if an instruction “breaks” the bind; these warnings are low priority and the option **-W** has to be set to show them. To bind a register to another, the source register should be already bound to a structure. In this case, both registers will point to the same field, but they will be followed separately. Note also the destination register should not be bound before **BIND**.

The “valid” instructions accepted and followed by the assembler are:

INC *<R>*, **DEC** *<R>*, **LDI** (*<R>*), **LDD** (*<R>*), **STI** (*<R>*), **STD** (*<R>*) and **LDI**, **CPI** if *<R>* is **BC**.

These instructions are considered to break the bind:

LD *<rh | rl>*, **INC** *<rh | rl>*, **DEC** *<rh | rl>*, **STA** *<rh | rl>*, **ADD** *<R>*, **POP** *<R>*, **LD** *<R>*,*<R'>* and **DJC** if *<R>* is **HL**.

To release a register, use **UNBIND** *<register>*. The assembler stops to follow the given register.

For example, with the structure `basfile_header` declared above:

```

BIND   BC,basfile_header      ; BC points to the base of basfile_header
BIND   DE,basfile_header.filelen ; DE points to the field filelen
LDI    (DE)
STA    H
LDI    (DE)                    ; Here DE points to fileprev
STA    L
; lhasm has updated DE to points to the next field: fileprev
LDA    OFFSETOF'DE

```

```

; If DE has now to points to filename, the offset should be substracted
BIND DE,basfile_header.filename
; But lhasm will genrate all the code for me ;)
LDA OFFSETOF'BC
BIND BC,basfile_header.filetype ; BC points to the field filetype
; So BC has now to points to filelen, the offset should be added
LDA OFFSETOF'BC
BIND HL,BC
LDA OFFSETOF'HL
DEC BC
LDA OFFSETOF'BC
BIND HL,basfile_header.fileprev.ptr
LDA OFFSETOF'HL
UNBIND BC
UNBIND DE
UNBIND HL

```

Running **lhasm** will give the following code:

```

27 40D7          BIND BC basfile_header
28 40D7          BIND DE basfile_header.filelen
29 40D7 B5 0C          LDA OFFSETOF'DE
30 40D9 55          LDI (DE)
31 40DA 28          STA H
32 40DB 55          LDI (DE) ; Here DE points to fileprev
33 40DC 2A          STA L
34 40DD          ; lhasm has updated DE to points to the next field: fileprev
35 40DD B5 0E          LDA OFFSETOF'DE
36 40DF          ; If DE has now to points to filename the offset should be
substracted
37 40DF          BIND DE basfile_header.filename
37 40DF FB 14 B1 0E 1A BIND DE basfile_header.filename
94 30 18
38 40E7 B5 00          LDA OFFSETOF'BC
39 40E9          ; But lhasm will genrate all the code for me ;)
40 40E9          BIND BC basfile_header.filetype
40 40E9 B5 0B FD CA BIND BC basfile_header.filetype ; BC points to the
field filetype
41 40ED          ; So BC has now to points to filelen the offset should be added
42 40ED B5 0B          LDA OFFSETOF'BC
43 40EF          BIND HL BC
44 40EF B5 0B          LDA OFFSETOF'HL
45 40F1 46          DEC BC
46 40F2 B5 0A          LDA OFFSETOF'BC
47 40F4          BIND HL basfile_header.fileprev.ptr
47 40F4 64 64 64 BIND HL basfile_header.fileprev.ptr
48 40F7 B5 0E          LDA OFFSETOF'HL
49 40F9          UNBIND BC
50 40F9          UNBIND DE
51 40F9          UNBIND HL

```

For subtracting, **lhasm** will use **DEC <R>** if the offset to subtract is less than **9**. Else it uses a register subtraction sequence. By the way, the assembler will use **DEC <R>** is the offset to add is less than 5. Else, it uses a addition sequence. Note that some code sequence will use the accumulator A and its current value is lost.

2.7/ Using DATA with structure

It may be useful to “declare” structure on a memory area. This help to organize the data inside the source code. When declaring a memory area as a DATA on a given structure, all the facilities of structures ate inherited by the DATA itself.

To declare a DATA area, the syntax is:

```
.DATA:    <Dname>    STRUCT <Sname>[,<Nelement>]
```

This will create a memory area called <Dname> mapped on a structure <Sname>. If <Nelement> is specified, the DATA will be an array of <Nelement>. If omitted, 0 is assumed for <Nelement>.

The following source **asm/d1.asm**:

```
.CODE  
  
.STRUCT: sd  
    b    byte  
    w    word  
    l    long  
    t    byte,#16  
.ENDSTRUCT  
  
.DATA:    d1    STRUCT sd  
  
.END
```

Running **lhasm** onto the source gives:

```
2          .CODE 40C5  
4          .STRUCT:    sd  
4          {  
5 ; sd: 1          +0000:0001.0001.0001.0001 b    {byte * #1}  
6 ; sd: 2          +0001:0002.0002.0002.0001 w    {word * #1}  
7 ; sd: 3          +0003:0004.0004.0004.0001 l    {long * #1}  
8 ; sd: 4          +0007:0010.0001.0001.0010 t    {byte * #16}  
9          }  
9          .ENDSTRUCT ; 0017 sd  
11  
11         .ORIGIN:    40C5  
11         .DATA:    d1  
11 40C5      {  
11         .BYTE 40C5 ; d1:#0.b    {byte * #1}  
11         .WORD 40C6 ; d1:#0.w    {word * #1}  
11         .LONG 40C8 ; d1:#0.l    {long * #1}  
11         .BYTE 40CC ; d1:#0.t    {byte * #16}  
11         .CODE 40DC  
11 40C5      }          ; 0017.0017.0001 d1  
11 40C5 00 00 00 00 00  .DATA:    d1 STRUCT sd  
          00 00 00 00 00  
          00 00 00 00 00  
          00 00 00 00 00  
          00 00 00  
13 40DC .END  
          .SYMBOLS:  
          40C5 d1
```

Note that the symbol **d1** is automatically created on address where **.DATA:** is performed.

If a fragment file is generated with the option **-F <fragfile>**, the DATA fragment will be put into this file. When disassembling the binary with **lhdump -F <fragfile>**, the DATA will be decoded following the fragments.

Because the structure has to be declared before the DATA, it is also possible to “initialize” the DATA according to the structure. For that, the syntax is:

```
.DATA:    <Dname>    STRUCT <Sname>[,<Nelement>]    INIT
            <field1 value>
            ...
.ENDDATA
```

For example, the source **asm/d2.asm**:

```
.CODE

.STRUCT: sd
    b    byte
    w    word
    l    long
    t    byte,#16
.ENDSTRUCT

.DATA:    d2    STRUCT sd INIT
            #8
            #16
            #32
            "ABCD" 00
.ENDDATA

.END
```

And running **lhasm** on this source gives:

```
2      .CODE 40C5
4          .STRUCT:    sd
4              {
5 ; sd: 1          +0000:0001.0001.0001.0001 b    {byte * #1}
6 ; sd: 2          +0001:0002.0002.0002.0001 w    {word * #1}
7 ; sd: 3          +0003:0004.0004.0004.0001 l    {long * #1}
8 ; sd: 4          +0007:0010.0001.0001.0010 t    {byte * #16}
9              }
9          .ENDSTRUCT ; 0017 sd
11
11      .ORIGIN:    40C5
11      .DATA:    d2
11 40C5          {
11      .BYTE 40C5 ; d2:#0.b    {byte * #1}
11      .WORD 40C6 ; d2:#0.w    {word * #1}
11      .LONG 40C8 ; d2:#0.l    {long * #1}
11      .BYTE 40CC ; d2:#0.t    {byte * #16}
11      .CODE 40DC
11 40C5          } ; 0017.0017.0001 d2
```

```

11          INIT
           {
12 40C5  08          #8
13 40C6  10          #16
14 40C7  20          #32
15 40C8  41 42 43 44 00  "ABCD"      00
16 40CD  00 00 00 00 00  ...
           00 00 00 00 00
           00 00 00 00 00
16          }
16          .ENDDATA      ; 0017 d2
18 40DC  .END
           .SYMBOLS:
           40C5  d2

```

When initializing the DATA, the last byte is repeated as filling pattern if the given initialization value does not fit fully into the last field of the structure. The values may be expressions, symbols, variables, text strings, "stringified" expressions, ...

The following code `asm/s2.asm` map the structure `basfile_header` on a memory area called `TopHeader` and on another memory area `PtrHeader`. `TopHeader` is array of 3 elements of type `basfile_header`.

```

.CODE
.STRUCT:      basfile_prev
              ptr      word,#2
.ENDSTRUCT
FILENAMELEN:  .EQU   #11
.STRUCT:      basfile_header
              filename  text,FILENAMELEN
              filetype  byte
              filelen   word
              fileprev  struct'basfile_prev,#3
.ENDSTRUCT

.DATA: TopHeader STRUCT basfile_header,#3 INIT
      aa
.ENDDATA

.DATA: PtrHeader STRUCT basfile_header INIT
      "Noname" 00 00 00 00 00
      ;01 02 03 04 05 06 07 08 09 0a 0b
      10 ; filetype
      ee dd
      ;<TopHeader >TopHeader
      >ADDRESSOF'TopHeader:#1.fileprev.ptr <ADDRESSOF'TopHeader:#1.fileprev.ptr
      OPCODE'NOP
      ;FF FF FF FF
.ENDDATA

```

Running `lhasm` will produce the initialization of `TopHeader` with the pattern `&AA` and this of `PtrHeader` with complex expressions:

```

1          .CODE  40C5
2          .STRUCT:      basfile_prev
2          {
3 ; basfile_prev: 1          +0000:0004.0002.0002.0002 ptr {word * #2}
4          }
4          .ENDSTRUCT      ; 0004 basfile_prev
5 40C5  FILENAMELEN:  .EQU  000B
6          .STRUCT:      basfile_header
6          {
7 ; basfile_header: 1      +0000:000B.0001.0001.000B filename  {text * #11}
8 ; basfile_header: 2      +000B:0001.0001.0001.0001 filetype  {byte * #1}
9 ; basfile_header: 3      +000C:0002.0002.0002.0001 filelen   {word * #1}

```

```

10 ; basfile_header: 4 +000E:000C.0001.0004.0003
fileprev {struct'basfile_prev * #3}
11 }
11 .ENDSTRUCT ; 001A basfile_header
13
13 .ORIGIN: 40C5
13 .DATA: TopHeader
13 40C5 {
13 .TEXT 40C5 ; TopHeader:#0.filename {text * #1}
13 .BYTE 40D0 ; TopHeader:#0.filetype {byte * #1}
13 .WORD 40D1 ; TopHeader:#0.filelen {word * #1}
13 .WORD 40D3 ; TopHeader:#0.fileprev {struct'basfile_prev *
#3}.ptr {word * #2}
13 .TEXT 40DF ; TopHeader:#1.filename {text * #1}
13 .BYTE 40EA ; TopHeader:#1.filetype {byte * #1}
13 .WORD 40EB ; TopHeader:#1.filelen {word * #1}
13 .WORD 40ED ; TopHeader:#1.fileprev {struct'basfile_prev *
#3}.ptr {word * #2}
13 .TEXT 40F9 ; TopHeader:#2.filename {text * #1}
13 .BYTE 4104 ; TopHeader:#2.filetype {byte * #1}
13 .WORD 4105 ; TopHeader:#2.filelen {word * #1}
13 .WORD 4107 ; TopHeader:#2.fileprev {struct'basfile_prev *
#3}.ptr {word * #2}
13 .CODE 4113
13 40C5 } ; 004E.001A.0003 TopHeader
13 INIT
{
14 40C5 AA aa
15 40C6 AA AA AA AA AA ...
AA AA AA AA AA
AA AA AA AA AA
AA AA AA AA AA
AA AA AA AA AA
AA AA AA AA AA
AA AA AA AA AA
AA AA AA AA AA
AA AA AA AA AA
AA AA AA AA AA
AA AA AA AA AA
AA AA AA AA AA
AA AA AA AA AA
AA AA
15 }
15 .ENDDATA ; 004E TopHeader
17 .DATA: PtrHeader
17 4113 {
17 .TEXT 4113 ; PtrHeader:#0.filename {text * #1}
17 .BYTE 411E ; PtrHeader:#0.filetype {byte * #1}
17 .WORD 411F ; PtrHeader:#0.filelen {word * #1}
17 .WORD 4121 ; PtrHeader:#0.fileprev {struct'basfile_prev *
#3}.ptr {word * #2}
17 .CODE 412D
17 4113 } ; 001A.001A.0001 PtrHeader
17 INIT
{
18 4113 4E 6F 6E 61 6D "Noname" 00 00 00 00 00
65 00 00 00 00
00
20 411E 10 10 ; filetype
21 411F EE DD ee dd
23 4121 ED
40 >ADDRESSOF'TopHeader:#1.fileprev.ptr <ADDRESSOF'TopHeader:#1.fileprev.ptr
24 4123 38 OPCODE'NOP
26 4124 38 38 38 38 38 ...
38 38 38 38
26 }
26 .ENDDATA ; 001A PtrHeader

```

The following functions are available to deal with the DATA:

SIZEOF'<Dname[.field]> Return the whole size,

TYPEOF '<Dname[.field]>	Return the size of the base type,
ELEMENTOF '<Dname[.field]>	Return the size of the element,
ARRAYOF '<Dname[.field]>	Return the number of elements,
OFFSETOF '<Dname[.field]>	Return the offset of the field inside the structure.

On DATA, the function **ADDRESSOF**' is available:

ADDRESSOF '<Dname[.field]>	Return address of the DATA or of the field inside the DATA.
-----------------------------------	---

As available on structure, the pseudo-instructions **BIND** and **UNBIND** are also working with the DATA.

Look the whole example in the source **asm/s2.asm**:

```
.CODE
.STRUCT:      basfile_prev
    ptr      word,#2
.ENDSTRUCT
FILENAMELEN: .EQU    #11
.STRUCT:      basfile_header
    filename  text,FILENAMELEN
    filetype  byte
    filelen   word
    fileprev  struct'basfile_prev,#3
.ENDSTRUCT

.DATA: TopHeader STRUCT basfile_header,#3 INIT
    aa
.ENDDATA

.DATA: PtrHeader STRUCT basfile_header INIT
    "Noname" 00 00 00 00 00
    ;01 02 03 04 05 06 07 08 09 0a 0b
    10 ; filetype
    ee dd
    ;<TopHeader >TopHeader
    >ADDRESSOF'TopHeader:#1.fileprev.ptr <ADDRESSOF'TopHeader:#1.fileprev.ptr
    OPCODE'NOP
    ;FF FF FF FF
.ENDDATA

. IF    STRUCT? basfile_header.filename
.PRINT2 "Length of filename " BCDOF'SIZEOF'basfile_header.filename
.ENDIF
. IF    STRUCT? basfile_hr.fname
.PRINT2 "Element of fname" ELEMENTOF'basfile_hr.fname
.ENDIF

LDA    OFFSETOF'basfile_header.filename
ADD    BC
LD     L,SIZEOF'basfile_header.filename
DEC    L
CLA

loop:  STI    (BC)
      DJC    loop

BIND   BC,basfile_header          ; BC points to the base of basfile_header
BIND   DE,basfile_header.filelen  ; DE points to the field filelen
LDI    (DE)
STA    H
LDI    (DE)
STA    L
BIND   BC,basfile_header.filetype
; Here lhasm does "pass" DE on the next field fileprev.
; But to point backward to filename, it needs to subtract...
BIND   DE,basfile_header.filename
; And lhasm will generate the good code ;)
```



```

UNBIND DE

BIND DE,BC
UNBIND DE

LDA OFFSETOF 'PtrHeader.filetype
LD B,<ADDRESSOF 'PtrHeader.filelen
LD C,>ADDRESSOF 'PtrHeader.filelen

BIND DE,PtrHeader.fileprev
BIND DE,PtrHeader.filelen

LD H,<ADDRESSOF 'TopHeader:#1.filetype
LD L,>ADDRESSOF 'TopHeader:#1.filetype

UNBIND BC
BIND BC,TopHeader
BIND HL,DE
UNBIND BC
UNBIND DE

BIND BC,TopHeader:#2.filetype
BIND BC,TopHeader:#2.fileprev.ptr
BIND BC,TopHeader:#2.fileprev.ptr:#1
BIND BC,TopHeader:#1
BIND BC,TopHeader:#0
BIND BC,TopHeader.filename

LDA SIZEOF 'PtrHeader
LDA ARRAYOF 'TopHeader
LDA ELEMENTOF 'TopHeader
LDA SIZEOF 'TopHeader
LDA SIZEOF 'PtrHeader.filelen
LDA ARRAYOF 'TopHeader.filetype
LDA ELEMENTOF 'TopHeader.fileprev.ptr
LDA SIZEOF 'TopHeader.fileprev
LDA ELEMENTOF 'TopHeader.fileprev
LDA ARRAYOF 'TopHeader.fileprev
LDA SIZEOF 'TopHeader:#1
LDA ELEMENTOF 'TopHeader:#1
LDA ARRAYOF 'TopHeader:#1

LDA OFFSETOF 'basfile_header.fileprev:#2.ptr:#1
LDA OFFSETOF 'TopHeader
LDA OFFSETOF 'TopHeader.filetype
LDA OFFSETOF 'TopHeader:#0
LDA OFFSETOF 'TopHeader:#0.filetype
LDA OFFSETOF 'TopHeader:#0.filelen
LDA OFFSETOF 'TopHeader:#0.fileprev
LDA OFFSETOF 'TopHeader:#0.fileprev:#1.ptr
LDA OFFSETOF 'TopHeader:#1
LDA OFFSETOF 'TopHeader:#1.filetype
LDA OFFSETOF 'TopHeader:#1.filelen
LDA OFFSETOF 'TopHeader:#1.fileprev
LDA OFFSETOF 'TopHeader:#1.fileprev:#1.ptr
LDA OFFSETOF 'TopHeader:#2
LDA OFFSETOF 'TopHeader:#2.filetype
LDA OFFSETOF 'TopHeader:#2.filelen
LDA OFFSETOF 'TopHeader:#2.fileprev
LDA OFFSETOF 'TopHeader:#2.fileprev:#1.ptr
CPA (ADDRESSOF 'TopHeader:#0.filelen)
CPA (ADDRESSOF 'TopHeader:#1.fileprev)
CPA (ADDRESSOF 'TopHeader:#2.fileprev:#2.ptr)
CPA (ADDRESSOF 'TopHeader:#2.fileprev:#0.ptr:#1)

.IF STRUCT? basfile_header.fileprev:#2.ptr
NOP
.ENDIF
.IF STRUCT? basfile_header
RET
.ENDIF
.IF STRUCT? basfile_header.top
OFF

```

```

.ENDIF

BIND    BC,TopHeader:#1                ; BC points to the base of basfile_header
BIND    DE,TopHeader:#2.filelen        ; DE points to the field filelen
LDA     OFFSETOF'DE
LDI     (DE)
STA     H
LDI     (DE)                            ; Here DE points to fileprev
STA     L
; lhasm has updated DE to points to the next field: fileprev
LDA     OFFSETOF'DE
; If DE has now to points to filename, the offset should be subtracted
BIND    DE,TopHeader:#2.filename
LDA     OFFSETOF'BC
; But lhasm will genrate all the code for me ;)
BIND    BC,TopHeader:#1.filetype       ; BC points to the field filetype
; So BC has now to points to filelen, the offset should be added
LDA     OFFSETOF'BC
BIND    HL,BC
LDA     OFFSETOF'HL
DEC     BC
LDA     OFFSETOF'BC
BIND    HL,TopHeader:#1.fileprev.ptr
LDA     OFFSETOF'HL
UNBIND  BC
UNBIND  DE
UNBIND  HL

```

Running **lhasm** on this source gives:

```

1          .CODE  40C5
2          .STRUCT:      basfile_prev
2          {
3  ; basfile_prev: 1          +0000:0004.0002.0002.0002 ptr {word * #2}
4          }
4          .ENDSTRUCT   ; 0004 basfile_prev
5 40C5     FILENAMELEN:  .EQU 000B
6          .STRUCT:      basfile_header
6          {
7  ; basfile_header: 1       +0000:000B.0001.0001.000B filename   {text * #11}
8  ; basfile_header: 2       +000B:0001.0001.0001.0001 filetype    {byte * #1}
9  ; basfile_header: 3       +000C:0002.0002.0002.0001 filelen    {word * #1}
10 ; basfile_header: 4       +000E:000C.0001.0004.0003
fileprev   {struct'basfile_prev * #3}
11          }
11          .ENDSTRUCT   ; 001A basfile_header
13
13          .ORIGIN:     40C5
13          .DATA:      TopHeader
13 40C5     {
13          .TEXT  40C5   ; TopHeader:#0.filename   {text * #11}
13          .BYTE  40D0   ; TopHeader:#0.filetype  {byte * #1}
13          .WORD  40D1   ; TopHeader:#0.filelen  {word * #1}
13          .WORD  40D3   ; TopHeader:#0.fileprev  {struct'basfile_prev *
#3}.ptr {word * #2}
13          .TEXT  40DF   ; TopHeader:#1.filename   {text * #11}
13          .BYTE  40EA   ; TopHeader:#1.filetype  {byte * #1}
13          .WORD  40EB   ; TopHeader:#1.filelen  {word * #1}
13          .WORD  40ED   ; TopHeader:#1.fileprev  {struct'basfile_prev *
#3}.ptr {word * #2}
13          .TEXT  40F9   ; TopHeader:#2.filename   {text * #11}
13          .BYTE  4104   ; TopHeader:#2.filetype  {byte * #1}
13          .WORD  4105   ; TopHeader:#2.filelen  {word * #1}
13          .WORD  4107   ; TopHeader:#2.fileprev  {struct'basfile_prev *
#3}.ptr {word * #2}
13          .CODE  4113
13 40C5     }          ; 004E.001A.0003 TopHeader
13          .INIT
13          {
14 40C5     AA          aa
15 40C6     AA AA AA AA AA  ...
          AA AA AA AA AA
          AA AA AA AA AA

```

```

AA AA AA AA AA
AA AA AA AA AA
AA AA AA AA AA
AA AA AA AA AA
AA AA AA AA AA
AA AA AA AA AA
AA AA AA AA AA
AA AA AA AA AA
AA AA AA AA AA
AA AA AA AA AA
AA AA AA AA AA
AA AA
15                                     }
15                                     .ENDDATA          ; 004E TopHeader
17                                     .DATA: PtrHeader
17 4113                               {
17                                     .TEXT 4113      ; PtrHeader:#0.filename      {text * #11}
17                                     .BYTE 411E      ; PtrHeader:#0.filetype    {byte * #1}
17                                     .WORD 411F      ; PtrHeader:#0.filelen {word * #1}
17                                     .WORD 4121      ; PtrHeader:#0.fileprev   {struct'basfile_prev *
#3}.ptr {word * #2}
17                                     .CODE 412D
17 4113                               }          ; 001A.001A.0001 PtrHeader
17                                     INIT
                                     {
18 4113 4E 6F 6E 61 6D                "Noname"          00 00 00 00 00
65 00 00 00 00
00
20 411E 10                            10          ; filetype
21 411F EE DD                          ee          dd
23 4121 ED
40                                     >ADDRESSOF'TopHeader:#1.fileprev.ptr <ADDRESSOF'TopHeader:#1.fileprev.ptr
24 4123 38                            OPCODE'NOP
26 4124 38 38 38 38 38                ...
38 38 38 38
26                                     }
26                                     .ENDDATA          ; 001A PtrHeader
28 412D +TRUE+                          .IF          STRUCT? basfile_header.filename
29 412D Length of filename 0011Length of filename 0011
30                                     /false/          .ENDIF
31 412D /false/                          .IF          STRUCT? basfile_hr.flname
32                                     /false/          .PRINT2 "Element of flname" ELEMENTOF'basfile_hr.flname
33                                     /false/          .ENDIF
35 412D B5 00                            LDA          OFFSETOF'basfile_header.filename
36 412F FD CA                            ADD          BC
37 4131 6A 0B                            LD           L SIZEOF'basfile_header.filename
38 4133 62                                DEC          L
39 4134 34                                CLA
40 4135 loop: .EQU 4135
41 4135 41                                STI         (BC)
42 4136 88 03                            DJC         loop
44 4138                                BIND        BC basfile_header
45 4138                                BIND        DE basfile_header.filelen
46 4138 55                                LDI         (DE)
47 4139 28                                STA         H
48 413A 55                                LDI         (DE)
49 413B 2A                                STA         L
50 413C                                BIND        BC basfile_header.filetype
50 413C B5 0B FD CA                        BIND        BC basfile_header.filetype
51 4140                                ;           Here lhasm does "pass" DE on the next field fileprev.
52 4140                                ;           But to point backward to filename it needs to subtract...
53 4140                                BIND        DE basfile_header.filename
53 4140 FB 14 B1 0E 1A                    BIND        DE basfile_header.filename
94 30 18
54 4148                                ;           And lhasm will generate the good code :)
55 4148                                UNBIND     DE
57 4148                                BIND       DE BC
58 4148                                UNBIND     DE
60 4148 B5 0B                            LDA          OFFSETOF'PtrHeader.filetype
61 414A 48 41                            LD           B <ADDRESSOF'PtrHeader.filelen
62 414C 4A 1F                            LD           C >ADDRESSOF'PtrHeader.filelen

```

```

64 414E          BIND    DE PtrHeader.fileprev
64 414E 58 41 5A 21 BIND    DE PtrHeader.fileprev
65 4152          BIND    DE PtrHeader.filelen
65 4152 5A 1F          BIND    DE PtrHeader.filelen
67 4154 68 40          LD      H <ADDRESSOF'TopHeader:#1.filetype
68 4156 6A EA          LD      L >ADDRESSOF'TopHeader:#1.filetype
70 4158          UNBIND  BC
71 4158          BIND    BC TopHeader
71 4158 48 40 4A C5      BIND    BC TopHeader
72 415C          BIND    HL DE
72 415C FD 98 FD 2A      BIND    HL DE
73 4160          UNBIND  BC
74 4160          UNBIND  DE
76 4160          BIND    BC TopHeader:#2.filetype
76 4160 48 41 4A 04      BIND    BC TopHeader:#2.filetype
77 4164          BIND    BC TopHeader:#2.fileprev.ptr
77 4164 4A 07          BIND    BC TopHeader:#2.fileprev.ptr
78 4166          BIND    BC TopHeader:#2.fileprev.ptr:#1
78 4166 4A 09          BIND    BC TopHeader:#2.fileprev.ptr:#1
79 4168          BIND    BC TopHeader:#1
79 4168 48 40 4A DF      BIND    BC TopHeader:#1
80 416C          BIND    BC TopHeader:#0
80 416C 4A C5          BIND    BC TopHeader:#0
81 416E          BIND    BC TopHeader.filename
83 416E B5 1A          LDA     SIZEOF'PtrHeader
84 4170 B5 03          LDA     ARRAYOF'TopHeader
85 4172 B5 1A          LDA     ELEMENTOF'TopHeader
86 4174 B5 4E          LDA     SIZEOF'TopHeader
87 4176 B5 02          LDA     SIZEOF'PtrHeader.filelen
88 4178 B5 01          LDA     ARRAYOF'TopHeader.filetype
89 417A B5 02          LDA     ELEMENTOF'TopHeader.fileprev.ptr
90 417C B5 0C          LDA     SIZEOF'TopHeader.fileprev
91 417E B5 04          LDA     ELEMENTOF'TopHeader.fileprev
92 4180 B5 03          LDA     ARRAYOF'TopHeader.fileprev
93 4182 B5 1A          LDA     SIZEOF'TopHeader:#1
94 4184 B5 1A          LDA     ELEMENTOF'TopHeader:#1
95 4186 B5 01          LDA     ARRAYOF'TopHeader:#1
97 4188 B5 18          LDA     OFFSETOF'basfile_header.fileprev:#2.ptr:#1
98 418A B5 00          LDA     OFFSETOF'TopHeader
99 418C B5 0B          LDA     OFFSETOF'TopHeader.filetype
100 418E B5 00          LDA     OFFSETOF'TopHeader:#0
101 4190 B5 0B          LDA     OFFSETOF'TopHeader:#0.filetype
102 4192 B5 0C          LDA     OFFSETOF'TopHeader:#0.filelen
103 4194 B5 0E          LDA     OFFSETOF'TopHeader:#0.fileprev
104 4196 B5 12          LDA     OFFSETOF'TopHeader:#0.fileprev:#1.ptr
105 4198 B5 1A          LDA     OFFSETOF'TopHeader:#1
106 419A B5 25          LDA     OFFSETOF'TopHeader:#1.filetype
107 419C B5 26          LDA     OFFSETOF'TopHeader:#1.filelen
108 419E B5 28          LDA     OFFSETOF'TopHeader:#1.fileprev
109 41A0 B5 2C          LDA     OFFSETOF'TopHeader:#1.fileprev:#1.ptr
110 41A2 B5 34          LDA     OFFSETOF'TopHeader:#2
111 41A4 B5 3F          LDA     OFFSETOF'TopHeader:#2.filetype
112 41A6 B5 40          LDA     OFFSETOF'TopHeader:#2.filelen
113 41A8 B5 42          LDA     OFFSETOF'TopHeader:#2.fileprev
114 41AA B5 46          LDA     OFFSETOF'TopHeader:#2.fileprev:#1.ptr
115 41AC A7 40 D1      CPA     (ADDRESSOF'TopHeader:#0.filelen)
116 41AF A7 40 ED      CPA     (ADDRESSOF'TopHeader:#1.fileprev)
117 41B2 A7 41 0F      CPA     (ADDRESSOF'TopHeader:#2.fileprev:#2.ptr)
118 41B5 A7 41 09      CPA     (ADDRESSOF'TopHeader:#2.fileprev:#0.ptr:#1)
120 41B8 +TRUE+          .IF    STRUCT? basfile_header.fileprev:#2.ptr
121 41B8 38              NOP
122          /false/          .ENDIF
123 41B9 +TRUE+          .IF    STRUCT? basfile_header
124 41B9 9A              RET
125          /false/          .ENDIF
126 41BA /false/          .IF    STRUCT? basfile_header.top
127          /false/          OFF
128          /false/          .ENDIF
130 41BA          BIND    BC TopHeader:#1
130 41BA 4A DF          BIND    BC TopHeader:#1 ; BC points to the base of
basfile_header
131 41BC          BIND    DE TopHeader:#2.filelen

```

```

131 41BC 58 41 5A 05          BIND    DE TopHeader:#2.filelen ; DE points to the field
filelen
132 41C0 B5 0C          LDA     OFFSETOF'DE
133 41C2 55          LDI     (DE)
134 41C3 28          STA     H
135 41C4 55          LDI     (DE) ; Here DE points to fileprev
136 41C5 2A          STA     L
137 41C6          ; lhasm has updated DE to points to the next field: fileprev
138 41C6 B5 0E          LDA     OFFSETOF'DE
139 41C8          ; If DE has now to points to filename the offset should be
substracted
140 41C8          BIND    DE TopHeader:#2.filename
140 41C8 58 40 5A F9        BIND    DE TopHeader:#2.filename
141 41CC B5 00          LDA     OFFSETOF'BC
142 41CE          ; But lhasm will genrate all the code for me ;)
143 41CE          BIND    BC TopHeader:#1.filetype
143 41CE 4A EA          BIND    BC TopHeader:#1.filetype ; BC points to the field
filetype
144 41D0          ; So BC has now to points to filelen the offset should be added
145 41D0 B5 0B          LDA     OFFSETOF'BC
146 41D2          BIND    HL BC
146 41D2 FD 6A          BIND    HL BC
147 41D4 B5 0B          LDA     OFFSETOF'HL
148 41D6 46          DEC     BC
149 41D7 B5 0A          LDA     OFFSETOF'BC
150 41D9          BIND    HL TopHeader:#1.fileprev.ptr
150 41D9 6A ED          BIND    HL TopHeader:#1.fileprev.ptr
151 41DB B5 0E          LDA     OFFSETOF'HL
152 41DD          UNBIND  BC
153 41DD          UNBIND  DE
154 41DD          UNBIND  HL
154 41DD          .END
          .SYMBOLS:
000B  FILENAMELEN
4113  PtrHeader
40C5  TopHeader
4135  loop

```

2.8/ JR and JP

A special feature is supported by the assembler. With the option **-J**, **lhasm** will replace the instructions **JR cc,nnnn** by a **JR !cc,+03 JP nnnn**, and the **JR nnnn** by a **JP nnnn**. After, the “optimizer” will run, and only the **JR cc,d** or the **JR d** with $d > 255$ will remain with a **JP**.

For example, the following code

```
.CODE
top:
    JR Z,end
    JR end
    JR top
    JR H top
    JR c,&3F00
    JP 4000
    JR NV 4321
    JR 5000
```

end:

will be rewritten:

40C5	89 03	JR	NZ,40CA
40C7	BA 40 E5	JP	40E5
40CA	BA 40 E5	JP	40E5
40CD	BA 40 C5	JP	40C5
40D0	85 03	JR	NH,40D5
40D2	BA 40 C5	JP	40C5
40D5	81 03	JR	NC,40DA
40D7	BA 3F 00	JP	3F00
40DA	BA 40 00	JP	4000
40DD	8F 03	JR	V,40E2
40DF	BA 43 21	JP	4321
40E2	BA 50 00	JP	5000

After the optimizer as run, finally, the code will be:

40C5	8B 16	JR	Z,40DD
40C7	8E 14	JR	40DD
40C9	9E 06	JR	40C5
40CB	97 08	JR	H,40C5
40CD	81 03	JR	NC,40D2
40CF	BA 3F 00	JP	3F00
40D2	BA 40 00	JP	4000
40D5	8F 03	JR	V,40DA
40D7	BA 43 21	JP	4321
40DA	BA 50 00	JP	5000

If the option **-Jloop=N** is set, the optimizer will stop after N loops. If $N=1$, no optimization is performed by the assembler. **-Jloop=0** is the same as **-J**.

Note that only the **JR** and **JR cc** are processed by the assembler. The jumps of the other instructions, like **DJC** or the **SBR** are kept as written in the source.

2.9/ BASIC program and assembly in-lining inside

To write a BASIC program, the directive **.BASIC** will start a **BASIC** fragment. If the name of the source file is ending by **.bas**, this fragment is assumed by default.

The syntax of a BASIC line is:

```
<basiclinenum> ["label"]<inst>[:...<inst>]
```

The valid *<basiclinenum>* are from **1** to **65279**. A space should follow the *<basiclinenum>* to separate it from the rest of the line:

```
10 PRINT I
```

or

```
10 PRINTI
```

will be compiled as the **PRINT** instruction and the variable **I**, but:

```
10PRINTI
```

will not be understood by the assembler.

The characters following the instruction **REM** are not compiled are kept as is.

All the keywords defined in the built-in ROM could be encoded, as these from the CE-150 and the CE-158 interfaces.

For example the source **asm/bas.bas**:

```
10 CLS:WAIT 0
20 FOR I=0 TO 10:PRINT I
30 NEXT I
40 BEEP 1:END
```

Running **lhasm -T asm/bas.bas** gives:

```
1
      .ORIGIN:      40C5
1 40C5 00 0A 07 F0 88      10  CLS:WAIT 0
      3A F1 B3 30 0D
2 40CF 00 14 0E F1 A5      20  FOR I=0 TO 10:PRINT I
      49 3D 30 F1 B1
      31 30 3A F0 97
      49 0D
3 40E0 00 1E 04 F1 9A      30  NEXT I
      49 0D
4 40E7 00 28 07 F1 82      40  BEEP 1:END
      31 3A F1 8E 0D
40F1 FF [END BASIC MARKER]
```

It is also possible to “enter” instruction not present inside the ROM or requiring external modules or software.

The escape sequence **\<code>** will enter the BASIC instruction by its code, where *<code>* is a 4 digits hexadecimal number between **&E000** and **&FEFF**.

```
10 \F097 "Hello"
```

gives:

```
1
      .ORIGIN:      40C5
1 40C5 00 0A 0A F0 97      10  \F097 "Hello"
      22 48 65 6C 6C
      6F 22 0D
```

New BASIC instructions created with the assembler are also available, if the keywords are exported by the option **-KE** *<keywfile>*. It could be in this way imported into the source file to be compiled properly.

For example, do:

```
lhasm -N -KE asm/erner1.keyw asm/erner1.asm
lhasm -T -N asm/erner1.bas
```

and see:

```

2          .IMPORT:      asm/erner1.keyw
4
4 40C5     .ORIGIN:      40C5
          00 0A 09 F1 9C      10    ON ERROR GOTO 99
          F1 B4 F1 92 39
          39 0D
5 40D1     00 14 06 F0 80      20    RAISE 100
          31 30 30 0D
6 40DA     00 1E 03 F1 8E      30    END
          0D
7 40E0     00 63 28 F1 82      99    BEEP1: PRINT "Error ";ERL;" in line ";ERL:RESUME
          31 3A F0 97 22
          45 72 72 6F 72
          20 22 3B F0 20
          3B 22 20 69 6E
          20 6C 69 6E 65
          20 22 3B F0 20
          3A 52 45 53 55
          4D 45 0D
410B     FF          [END BASIC MARKER]
```

The source code of **asm/erner1.bas** is:

```
.IMPORT:  asm/erner1.keyw
```

```

10 ON ERROR GOTO 99
20 RAISE 100
30 END
99 BEEP1: PRINT "Error ";ERL;" in line ";ERL:RESUME
```

Inside BASIC string or line, some special characters may be entered, if they follow the escape sequence **\<char>** or **\<code>**., To enter a ****, do ****. The characters **\pi** **\yen** **\sqr** **\ins** are the ASCII code **&5D**, **&5C**, **&5B** and **&39**. The ASCII code may be directly entered by **\<code>**, i.e, **\41** for A.

For example:

```
10 PRINT "\pi\yen\7c\7e\\"
```

gives:

```

          .ORIGIN:      40C5
1 40C5     00 0A 0A F0 97      10    PRINT "\pi\yen\7c\7e\\"
          22 5D 5C 7C 7E
          5C 22 0D
```

Even in BASIC, it is still possible to access to the symbols, the BASIC line addresses and to evaluate some expressions.

The BASIC compiler understands the following instructions:

```
\addr[<linenum>] Returns the address of the first instruction (i.e the
address of the line + 3) of the BASIC <linenum>
specified and compile it into the BASIC line,
```


\get [<symbol>]	Returns the value of the symbol <symbol> and compile it into the BASIC line,
\eval8 [<expr>]	Returns the 8-bits value of the expression <expr> and compile it into the BASIC line,
\eval16 [<expr>]	Returns the -bits value of the expression <expr> and compile it into the BASIC line.

For example, the BASIC source **asm/bas2.bas**:

```

10 REM ABCDEF
20 POKE \addr[10]+2,\eval8[opcode'RET]
30 POKE \get[A$],&01,&02,&03,\eval8[>1234]
40 CALL \eval16[[+3]A$]

```

Running **lhasm** (at **40C5**) will produce the following code:

```

10 REM ABCDEF
20 POKE &40C8+2,&9A
30 POKE &78C0,&01,&02,&03,&34
40 CALL &78C3

```

Each time a BASIC line is compiled, the assembler defines a new symbol containing the absolute address of the first instruction in this BASIC line (i.e. + 3). The symbol is named:

<source name>\$\$**._addr:***<linenum>*

where *<linenum>* is the BASIC line number compiled.

In the example above, the symbols are:

```

40C8 asm/bas2.bas$$._addr:00010
40D4 asm/bas2.bas$$._addr:00020
40E5 asm/bas2.bas$$._addr:00030
4100 asm/bas2.bas$$._addr:00040

```

These symbols are global and exported.

To simplify the introduction of assembly code inside BASIC instructions like **REM**, **POKE** and **DATA** or when assigning a \$ variable, it is now possible to call the assembler while a BASIC fragment is active.

The syntax is the following:

```

<basiclinenum> ...<inst>:...<inst> \asm[
    assembly code, with symbols, variables and macros
\]end <inst>...

```

Note the **\asm[** should be at the end of the source line and **\]end** at the beginning of a source line followed by a space.

A small example below:

```

        .MACRO:      LDDB_nn
LD      B,<_#0
LD      C,>_#0
        .ENDMACRO
; .BASIC
10 REM  \asm[
    %80h .EQU ^08
        LDA 00
        LDDB_nn 7750
        LD L,%80h
loop:

```

```

                STI    (BC)
                DJC    loop
                RET
        \]end
20 POKE A, \asm[
                SBR    (F2)
                CALL  &ED00
                RET
        \]end
30 E$="\asm[
                LDBC_nn    str
                RET
        str:  .EQU  .
                \$A \$B \$C
        \]end EFGH"
40 DATA  \asm[
        PUSH  HL
        PUSH  BC
        CALL  BEEP1
        POP   BC
        POP   HL
        RET
        \]end
50 END

```

Running **lhasm** on this source **te5.bas** will give:

```

1          .MACRO:    LDBC_nn
1          {
2 ; LDBC_nn: 1          LD    B,<_#0
3 ; LDBC_nn: 2          LD    C,>_#0
4          }
4          .ENDMACRO ; LDBC_nn
5 40C5      ;.BASIC
6 40CA      10      REM  \asm[
7 40CA  %80h  .EQU  0080
8 40CC      LDA    00
9          LDBC_nn  7750
9          {
9 40CE      LD    B <_#0
9 40D0      LD    C >_#0
9          }
10 40D2     LD    L %80h
11 40D2  loop: .EQU  40D2
12 40D3     STI    (BC)
13 40D5     DJC    loop
14 40D6     RET
15 40C5     00 0A 0F F1 AB  \]end
           B5 00 48 77 4A
           50 6A 80 41 88
           03 9A 0D
16 40DE     20      POKE A, \asm[
17 40E5     SBR    (F2)
18 40F1     CALL  &ED00
19 40F5     RET
20 40D7     00 14 1C F1 A1  \]end
           41 2C 26 43 44
           2C 26 46 32 2C
           26 42 45 2C 26
           45 44 2C 26 30

```

```

30 2C 26 39 41
0D
21 40FD          30    E$="\asm[
22              LDBC_nn  str
22              {
22 40FF          LD     B <__#0
22 4101          LD     C >__#0
22              }
23 4102          RET
24 4102  str:   .EQU 4102
25 4105          \$A   \$B   \$C
26 40F6  00 1E 12 45 24  \jend EFGH"
           3D 22 48 41 4A
           02 9A 41 42 43
           45 46 47 48 22
0D
27 4110          40    DATA  \asm[
28 4117          PUSH  HL
29 411F          PUSH  BC
30 412B          CALL  BEEP1
31 4133          POP   BC
32 413B          POP   HL
33 413F          RET
34 410B  00 28 32 F1 8D  \jend
           26 46 44 2C 26
           41 38 2C 26 46
           44 2C 26 38 38
           2C 26 42 45 2C
           26 45 36 2C 26
           36 39 2C 26 46
           44 2C 26 30 41
           2C 26 46 44 2C
           26 32 41 2C 26
           39 41 0D
35 4140  00 32 03 F1 8E   50    END
0D
4146  FF          [END BASIC MARKER]
;; 40C5          te5.bas$$._start
;; 4147          te5.bas$$._end
;; 0082          te5.bas$$._length

.SYMBOLS:
40D2  loop
4102  str

0080  %80h

```

and the following BASIC program:

```

10 REM \B5\00HwJPj\80A\88\03\9A
20 POKE A,&CD,&F2,&BE,&ED,&00,&9A
30 E$="HAJ\02\9AABCEFGH"
40 DATA &FD,&A8,&FD,&88,&BE,&E6,&69,&FD,&0A,&FD,&2A,&9A
50 END

```

2.10/ Creating and registering BASIC keywords

The assembler knows how to work with new BASIC keywords. So, it possible to create the assembly code for an new BASIC instruction or function, and to define a BASIC keyword and finally to register this new BASIC keyword in the user's keyword table.

Please refer to other documentation to learn how to deal with BASIC instructions.

In this example, the new BASIC instruction **RAISE** is created. We first do a define of the new keyword "**RAISE**" and write the code for the instruction:

```
.ORIGIN: 47C5
.CODE
.DEFINE:    "RAISE"      = FOEO N
            EVAL doerrH
            INTG 08,doerrH
            STA  H
            LDA  H
            SBR  Z,(E2)
doerrH:
            ERRH
.END
```

At this time, a new BASIC keyword is defined by the assembler:

- The entry point, **RAISE\ \. _start** is automatically declared by the assembler at the current address of **.DEFINE:**,
- The name is **RAISE**. The keyword name is specified between two double quotes "
- The code for the BASIC compiler is **&FOEO**. For automatic code allocation, see below,
- The bits **N** means that this instruction is available in NORMAL and in a BASIC program, like **PRINT**.

The new keyword is fully global; it is visible in whole source and all included files, but also including files.

If keyword table is created in the source by the fragment **.KEYWORD**, just write "**RAISE**" in this fragment to register the **RAISE** instruction in the table. Of course, some specifics initializations (**POKE**) have to called before to have this instruction understood by the original BASIC ROM.

In our example, we will write:

```
; Keyword table should be aligned on a 2Kbytes frontier
.ALIGN:    0800
; Do not care of the &54 bytes from &xx00 to &xx53
.HOLE
.SKIP 054
; The keyword table starts. Enter into a KEYWORD fragment
.KEYWORD
"RAISE"    ; our keyword RAISE
```

The option **-K <keywfile>** gives the opportunity to write all keywords in a file. This may be very useful for the dumper. If another source needs a reference to this keyword, it is possible to export the keyword with the option **-KE <keywfile>**. For backward compatibility with older versions (< **0.6.0**), use **-KK** or **-KKE** options respectively.

Running **lhasm** on the source **raise.asm** will output:

```

1
      .ORIGIN:      47C5
2      .CODE 47C5

3 47C5  .DEFINE:      "RAISE"      = FOEO N
      ;; FOEO      RAISE\\._code
      ;; 47C5      RAISE\\._jump
      ;; 4ECO      RAISE\\._bits
4 47C5  DE 07          EVAL  doerrH
5 47C7  D0 08 04      INTG   08 doerrH
6 47CA  28            STA   H
7 47CB  A4            LDA   H
8 47CC  CB E2        SBR   Z (E2)
9 47CE  doerrH:      .EQU 47CE
10 47CE  E0           ERRH
12 47CF          ;    Keyword table should be aligned on a 2Kbytes frontier
13
      .ALIGN:      4800
14 4800          ;    Do not care of the &54 bytes from &xx00 to &xx53
15      .HOLE 4800
17 4854          ;    The keyword table starts. Enter into a KEYWORD fragment
18      .KEYWORD 4854
19 4855  .KEYWORD: "RAISE"  FOEO 47C5 N
19 4855  52 41 49 53 45  "RAISE"      ; our keyword RAISE
      FO E0 47 C5 D0
21 485F  .END
      ;; 47C5      raise.asm$$._start
      ;; 485F      raise.asm$$._end
      ;; 009A      raise.asm$$._length

      .SYMBOLS:
47CE  doerrH

```

It is also possible to let the assembler automatically fetching a code for a keyword. This is useful to write a code with keyword assembly routines gotten on the flow.

To do this, the following code syntax is expected:

```
AUTO.t?<code>
```

Where *t* (type) is one of **I F** or **V**, and <code> is a 4-hexadecimal code.

- **I** stands for **INSTRUCTION**, like **PRINT**, **IF** or **NEW**. The valid codes are from **&F080** to **&F0FF**.
- **F** stands for **FUNCTION**, like **CHR\$**, **SIN** or **LEN**. The valid codes are from **&F060** to **&F07F**.
- **V** stands for **VARIABLE**, like **MEM**, **PI** or **TIME**. The valid codes are from **&F020** to **&F05F**.

If the <code> already exists, the assembler will automatically choose the next code available in the type range.

Finally, the assembler may automatically fetching a code for a keyword. This is useful to write a code with keyword assembly routines gotten on the flow.

To do this, the following code syntax is expected:

```
AUTO.t
```

Where *t* (type) is one of **I F** or **V**, as described above. In this last case, the next available code for the type range will be allocated by the assembler up to all codes are busy.

Look the source of **erner1.asm**:

```
.CODE
```

```

.DEFINE: "ERN"      = AUTO.V?F054 N
      LDA  (ERRORNUM)
      JP   D9E4

.DEFINE: "ERL"      = AUTO.V N
      LDU  (ERRORLINE)
      JP   DA6C

.DEFINE: "RAISE"    = AUTO.I?F097 N
      EVAL doerr
      INTG 00,doerr
      LDA  H
      STA  H
      SBR  Z,(&E2)
doerr:
      ERH

.DEFINE: "PRINTERR" = F097 N
      JP   PRINT\\._jump

.END

```

Running **lhasm** on the source **erner1.asm** will output:

```

1      .CODE 40C5

3 40C5  .DEFINE:      "ERN" = F054 N
      ;; F054      ERN\\._code
      ;; 40C5      ERN\\._jump
      ;; 4ECO      ERN\\._bits
4
      .ORIGIN:      40C5
4 40C5  A5 78 9B      LDA  (ERRORNUM)
5 40C8  BA D9 E4      JP   D9E4

7 40CB  .DEFINE:      "ERL" = F020 N
      ;; F020      ERL\\._code
      ;; 40CB      ERL\\._jump
      ;; 4ECO      ERL\\._bits
8 40CB  F4 78 B4      LDU  (ERRORLINE)
9 40CE  BA DA 6C      JP   DA6C

11 40D1  .DEFINE:      "RAISE" = F080 N
      ;; F080      RAISE\\._code
      ;; 40D1      RAISE\\._jump
      ;; 4ECO      RAISE\\._bits
12 40D1  DE 07      EVAL doerr
13 40D3  D0 00 04      INTG 00 doerr
14 40D6  A4      LDA  H
15 40D7  28      STA  H
16 40D8  CB E2      SBR  Z (&E2)
17 40DA  doerr: .EQU 40DA
18 40DA  E0      ERH

20 40DB  .DEFINE:      "PRINTERR" = F097 N
      ;; F097      PRINTERR\\._code
      ;; 40DB      PRINTERR\\._jump
      ;; 4ECO      PRINTERR\\._bits
21 40DB  BA E4 EB      JP   PRINT\\._jump
23 40DE  .END

```

Because **PRINT** already use the code **&F097**, the assembler automatically choose **&F080** for **RAISE**. The keyword **PRINTERR** force the use the code **&F097**. In a same way, like the code **&F054** is free, **ERN** may use it. Finally, the assembler choose itself the code for **ERL** and it takes **&F020**.

2.11/ RESERVE area

It is also possible to encode a memory as it is a RESERVE area. To do so, the fragment RESERVE has to be activated by the directive **.RESERVE**.

Inside this fragment, the syntax is:

```
<page>.F<key> <reservedata>
```

Where <page> is **I II** or **III** and <key> is **1** to **6** or **! " # \$ % &** respectively, mapping the 6 'keys' below the screen.

The <reservedata> may be any BASIC instruction, a string between double-quote, a character, a byte value or an expression.

For example the source **asm/reserve.asm**:

```
.RESERVE  
II.F# "ABC" &40  
I.F5 &F0 &97 $@  
III.F1 <CODEOF' "INPUT" >CODEOF' "INPUT"  
II.F$ BEEP "1"
```

gives the following encoding of a RESERVE area:

```
0  
      .ORIGIN:      40C5  
1      .RESERVE     40C5  
3 40C5 13 41 42 43 40  II.F# "ABC" &40  
4 40CA 05 F0 97 40    I.F5  &F0 &97 $@  
5 40CE 09 F0 91      III.F1      <CODEOF' "INPUT" >CODEOF' "INPUT"  
6 40D1 14 F1 82 31  II.F$ BEEP "1"  
8 40D5 .END
```


2.12/ Assembler directives

.ORIGIN: *<base addr>*

Set *<base addr>* as new origin address.

.ALIGN: *<frontier>*

Compute the next address to be aligned on the given *<frontier>*. The bytes value between the current address and the next aligned address is set to **&00**. The new aligned address is taken as current assembler address.

.JUMPTO: *<addr>*

Set *<addr>* as new origin address. Note that *<addr>* may be an expression, a symbol or a variable.

.SKIP: *<nbytes>*

.SKIP *<nbytes>*

Skip *<nbytes>* and set new origin address.

.END

End the assembler and update pointers for saving binary file. If **-ns** is not specified and **-T** or **-L** *<logfile>* are given, the symbols and variables defined are listed after a **.SYMBOLS:** banner. If **-ns** is set, the symbols and variables are not listed. If **-na** is specified, the local symbols are not listed.

.COMMENT: *<comment>*

Set a comment to the current fragment.

.BASIC

Enter into BASIC fragment. BASIC lines are compiled. A BASIC line start with a line number **1..65529** followed by a space and one or several BASIC keywords or expression.

.CODE

Enter into CODE fragment. LH5801 mnemonics are assembled.

.BYTE

Enter into BYTE fragment. Bytes 8-bits values are compiled. Text strings may be entered between " .

.WORD

Enter into WORD fragment. Words 16-bits values are compiled.

.LONG

Enter into LONG fragment. Longs 32-bits values are compiled.

.TEXT

Enter into TEXT fragment. Text between " are compiled.

.KEYWORD

Enter KEYWORD fragment. The BASIC keyword table is built. The word pointers area is updated. Note that **.KEYWORD** is expected to be specified on a 2048 bytes frontier + **&54**, i.e, **&0054**, **&0854**, **&1054**, etc...

.HOLE

Enter into HOLE fragment. Obscure area. Only **.SKIP** *<n>* is expected to skip *<n>* bytes.

.EXPORTALL

All symbols in the current source are treated as global symbols.

.EXPORT: *<name>* [**.EQU** *<value>*]

Define a global symbol *<name>* with the given *<value>*. If **.EQU** *<value>* is omitted, the current assembler address is taken.

<name>: [**.EQU** *<value>*]

Define a global or a local symbol *<name>* with the given *<value>*. If **.EQU** *<value>* is omitted, the current assembler address is taken. The scope of global is forced if **.EXPORTALL** is specified, or if **.LOCAL** is not given before in the source.

[**.EXPORT:**] *<name>*: **.ARRAYOF** *<item size>* *<base>* *<end>*

Define a global or local symbol *<name>* with the number of elements computed inside the array starting at *<base>* and ending at *<end>* and composed by items of the given size. The *<item size>* may be an immediate value, **BYTE** or **WORD**.

.LOCAL

All symbols defined after will be declared as local, except if preceded by **.EXPORT:** or if **.EXPORTALL** is specified in the source. If the option **-a** is given to **lhasm**, all symbols are assumed as local.

%mnc [**.EQU** *<value>*]

Define the variable **%mnc** with the given *<value>*. If **.EQU** *<value>* is omitted, the current assembler address is taken. The variable name is on the form **%mnc** where *m* and *n* are a digit from **0** to **9**, and *c* is lowercase letter from **a** to **z**. A variable is always global.

%mnc **.ARRAYOF** *<item size>* *<base>* *<end>*

Define the variable **%mnc** with the number of elements computed inside the array starting at *<base>* and ending at *<end>* and composed by items of the given size. The *<item size>* may be an immediate value, **BYTE** or **WORD**.

.SUBSTITUTE: *<subname>* = *<subexpr>*

Define a substitution string *<subname>* with the given *<subexpr>*. When a symbol is defined (local or global) and contains the pattern **__**/*<subname>*/**_**, the pattern is replaced by the *<subexpr>* of the substitution string. The substitution strings are global, and should be defined only once. Note that the *<subexpr>* are not values, but are a string which is evaluated when the symbol is defined. The substitution strings scope is global to whole source (and included files) from its definition to the end of the assembler work.

.DEFINE: "<keyword>" = <code> <bits>

Define <keyword> with the <code> as a new BASIC keyword. The entry point is fixed to the current PC address. The <bits> parameter is one of the following letters:

- **N** normal usage, like **PRINT** or **SIN**,
- **P** programmable only in a BASIC program like **FOR**,
- **C** command only like **NEW**,
- **?** unsupported mode.

.DEFINE: "<keyword>" = **AUTO.**<t>?<code> <bits>

Like **.DEFINE:** above, but let the assembler choose the code if this specified by <code> is already taken. In the syntax, <t> (type) is one of **I F** or **V**, and <code> is a 4-hexadecimal code.

- **I** stands for **INSTRUCTION**, like **PRINT**, **IF** or **NEW**. The valid codes are from **&F080** to **&FOFF**.
- **F** stands for **FUNCTION**, like **CHR\$**, **SIN** or **LEN**. The valid codes are from **&F060** to **&F07F**.
- **V** stands for **VARIABLE**, like **MEM**, **PI** or **TIME**. The valid codes are from **&F020** to **&F05F**.

.CHECKSUM [[+](<code>)] [<start-address> [<end-address>]]

Perform a checksum computation and write checksum value as a 16-bits word at the current address. The checksum is computed from the first **.ORIGIN:** and up to the current address.

If (<code>) is given, the checksum will be stored after putting <code>.

If + is given before (<code>), the <code> will be added to the checksum computed.

If <start-address> is given, it will be taken as start address for checksum computation. Also if <end-address> is specified, it will be taken as end address for the checksum computation. If <end-address> is . and +(<code>) is written, . will reference the address after the <code>.

.CHECKSUM [() | +<expr>] [<start-address> [<end-address>]]

This second syntax is supported starting **lhTools-0.7.5**. Idem as the *old* **.CHECKSUM** but the code is filled with <expr> which may be any expression, like **OPCODE'**.

The *old* syntax of **.CHECKSUM** is still accepted.

Like the <expr> is optional, if it is not used, but the <start-address> [<end-address>] is expected, a () should be put as first argument.

.DATESTAMP [<expr>]

.TIMESTAMP [<expr>]

Add a TIME BCD-value (**hhmmss**- hourminutesecond) or a DATE BCD-value (**YYMMDD**- yearmonthday) at the current address.

hhmmss is in 24-hour clock format. **YY** is year modulo **100**.

If <expr> is given, the DATE or TIME BCD-value will be stored after putting the code computed by <expr>. Like **.CHECKSUM**, the <expr> may be any expression, like **OPCODE'**.

.MACRO: *<name>*

Define a new macro *<name>*. All code given is assumed to be part of the macro until **.ENDMACRO** is encountered.

.ENDMACRO

End the current macro.

.STRUCT: *<name>*

Define a new structure *<name>*.

.ENDSTRUCT

End the current structure definition.

.INCLUDE: *<file>*

Include the file *<file>*. If the file is already included nothing is done. If *<file>* is not found in the current directory, it will be searched first with the same directory as the source file which includes it. After, it will be searched in all the directories specified by the options **-I** *<includepath>*.

.IMPORT: *<symfile>* | *<keywfile>*

Include the symbol or keyword file *<file>*. This file is one of the “output” file generated by **lhasm** with the option **-S** *<symfile>* or **-K[K]** *<keywfile>*.

.IF [NOT] <test> [*<val1>* [*<val2>*]]

[.ELSE]

.ENDIF

Evaluate the *<test>* and if **TRUE**, execute the lines between **.IF** and **.ELSE** if specified or **.ENDIF**. When the assertion is **FALSE**, execute the lines between the **.ELSE** and **.ENDIF**. If no **.ELSE** is specified, nothing is done. The following *<test>* are understood:

- **VERSION?** *x.y.z.t* is **TRUE** if the current **lhasm** version is greater or equal to version *x.y.z.t* specified. Note *x* or *x.y* or *x.y.z* or *x.y.z.t* are valid.
- **INCLUDED?** is **TRUE** if the source is executed in a **.INCLUDE:** directive.
- **ORIGIN?** is **TRUE** if an origin is already set by the directive **.ORIGIN:** or by the option **-O**.
- **MACHINE?** is **TRUE** if a machine is declared.
- **MODULE?** is **TRUE** if a module is declared.
- **MACHINE?** *<machine>* is **TRUE** if a machine is declared and if it is equal to the *<machine>* specified. Valid *<machine>* are **PC1500 PC1500A PTA4000+16** and **PC1560**.
- **MODULE?** *<module>* is **TRUE** if a module is declared and if it is equal to the *<module>* specified. Valid *<module>* are **CE151 CE155 CE159 CE161** and **CE163**.
- **EXIST?** *<name>* is **TRUE** if *<name>* is defined in the current scope.
- **SUBSTITUTE?** *<subname>* is **TRUE** if *<subname>* is defined.
- **KEYWORD?** “*<name>*” is **TRUE** if *<name>* is defined as a keyword by the directive **.DEFINE:** or by **.IMPORT:**.

- **STRUCT?** *<name>* is **TRUE** if *<name>* is an existing structure or field.
- **EQUAL?** *<val1>* *<val2>* is **TRUE** if *<val1>* is equal to *<val2>*.
- **LESS?** *<val1>* *<val2>* is **TRUE** if *<val1>* is less than *<val2>*.
- **GREATER?** *<val1>* *<val2>* is **TRUE** if *<val1>* is greater than *<val2>*.
- **PASS?** *<num>* is **TRUE** if the current assembler pass is equal to *<num>*. Note that valid *<num>* values are **1** or **2**.
- **0** is always **FALSE**. **1** is always **TRUE**. This is a simple way to “remove” or “insert” code.

If **NOT** *<test>* is specified, the result of *<test>* is “negated”: If *<test>* is **TRUE**, the result of **.IF** will be **FALSE**; If *<test>* is **FALSE**, the result of **.IF** will be **TRUE**.

.NOP IF [NOT] <test> [<val1> [<val2>]]
.ENDNOP

The assembly source enclosed between **.NOP** and **.ENDNOP** is replaced by **NOP** opcode (**&38**) if the *<test>* is **TRUE**. Else the source is assembled normally. The *<test>* assertions are the same as the directive **.IF** described above.

.WARNING "*string*"
.ERROR "*string*"
.FATAL "*string*"

Raise a warning, an error or a fatal error and print the "*string*" specified. A fatal error will abort **lhasm** and a non-null error code is returned.

.PRINT "*string*"|*value* ["*string*"|*value* ...]

Print the message composed by all strings and/or values to **stdout** and to log file if one.

.PRINT2 "*string*"|*value* ["*string*"|*value* ...]

Same as **.PRINT** above, but it is executed only when the assembler is running the pass 2.

.DEBUG "*string*"|*value* ["*string*"|*value* ...]

Print the message composed by all strings and/or values to **stdout** and to log file if one, but only if the assembler is running with debug mode enabled (option **-d**).

.DEBUG2 "*string*"|*value* ["*string*"|*value* ...]

Same as **.DEBUG** above, but it is executed only when the assembler is running the pass 2.

.PC1500

Declare a **PC1500** machine.

.PC1500A

Declare a **PC1500A** machine.

.PC1560

Declare a **PC1560** machine.

.PTA4000+16

Declare a **PTA4000+16** machine.

.CE151

Declare a **CE151** module.

.CE155

Declare a **CE155** module.

.CE159

Declare a **CE159** module.

.CE161

Declare a **CE161** module.

.CE163

Declare a **CE163** module. The memory scheme is like the **CE161**. The bank1 is not supported.

.RAM

If a machine is declared, check the current section to be a RAM section.

.ROM

If a machine is declared, check the current section to be a ROM section.

.SYS

If a machine is declared, check the current section to be a SYS variable section (area **&7000..&7FFFF**).

.LM

If a **PC1500A** is declared, check the current section to be a LM section (area from **&7C01..&7FFFF**).

.FILL: *<n time>* **WITH** *<val1>* [*<val2> ...*]

.FILLTO: *<address>* **WITH** *<val1>* [*<val2> ...*]

.FILLALIGN: *<frontier>* **WITH** *<val1>* [*<val2> ...*]

Fill from the current assembler address to the *<address>* specified, up to the *<frontier>* specified or a number of time *<n time>* with the pattern *<val1>* [*<val2> ...*].

.CHECKSUM:

.FULLCHECKSUM:

.DECLARE:

.SYMBOLS:

Dummy directives handled for backward compatibility with **lhdump**.

2.13/ Immediate assembler

The standard assembler has two-passes. But it is also possible to generate code immediately by calling the immediate assembler, ie, one-pass only with the option **-i**. In this case, the source code is read from **stdin** and if the trace mode is redirected to **stderr** (option **-T**), the immediate code and informations are printed.

To exit from immediate assembler, use **CTRL+D**. Exiting by **CTRL+C** will not write a binary file, and the generation of symbols, fragments and macros files may be disturbed by **CTRL+C**.

If no **-o <binfile>** option is given, **stdin.bin** is used as output binary file.

Note that when running with immediate assembler, variables and symbols should be defined to correct value BEFORE assembling, else an error may generated due to bad value or undefined. But macro definition and expansion are usable with the immediate assembler

An example: Type **lhasm -T -i -O 40C5 -c**

```

.!.
  1 40C5  .CODE
CLA
  2 40C5  34                CLA
LD B,79
  3 40C6  48 79            LD   B 79
LD C,00
  4 40C8  4A 00            LD   C 00
loop:
  5 40CA  loop: .EQU 40CA
STI
  6 40CA  F5                STI
CP C,C0
  7 40CB  4E C0            CP   C C0
JR NC loop
  8 40CD  91 05            JR   NC loop
RET
  9 40CF  9A                RET
.END
```

<use CTRL+D to exit from immediate assembler>

Written 11 bytes (40C5:40D0) to stdin.bin

When started in immediate mode, the assembler accepts the directives below:

!.!

Displays the current fragment and address.

.NOOUTPUT

Set the **-N** option. No binary output will be done.

2.14/ Structured sources and programation

The assembler provides structured proclamation. This will reduce the number of symbols, but also help into source maintenance, visibility and development.

Imagine the following result:

```
40C5 45          LDI   (BC)
40C6 8B 05       JR    Z,40CD
40C8 B9 7F       AND   7F
40CA 51          STI   (DE)
40CB 9E 08       JR    40C5
40CD 49 00       AND   (BC),00
40CF 44          INC   BC
40D0 4E 80       CP    C,80
40D2 89 01       JR    NZ,40D5
40D4 44          INC   BC
40D5 4E FF       CP    C,FF
40D7 91 0C       JR    NC,40CD
40D9 25          LDA   (HL)
40DA BF 80       BIT   80
40DC 89 04       JR    NZ,40E2
40DE BD FF       XOR   FF
40E0 8E 02       JR    40E4
40E2 B9 7F       AND   7F
40E4 34          CLA
40E5 41          STI   (BC)
40E6 88 03       DJC  40E5
40E8 9A          RET
```

No symbols are defined and the source is:

```
.CODE

begin                ; 1:
    LDI   (BC)        ;    ldi (bc)
while NZ             ;    jr z,2:
    AND   &7F         ;    and 7f
    STI   (DE)        ;    sti (de)
repeat              ;    jr 1:
                    ; 2:
begin                ; 3:
    AND   (BC),&00    ;    and (bc),&00
    INC   BC          ;    inc bc
    CP   C,&80        ;    cp c,&80
    if   Z            ;    jr nz,31:
        INC   BC      ;    inc bc
    endif            ; 31:
    CP   C,&FF        ;    cp c,&ff
until >=            ;    jr nc,3:

LDA   (HL)          ;    lda (hl)
BIT   &80            ;    bit &80
if   =              ;    jr nz,4:
    XOR   &FF        ;    xor &ff
else                ;    jr 5:
    AND   &7F        ; 4: and 7f
endif              ; 5:
```



```

CLA                ;   cla
begin              ; 6:
    STI    (BC)    ;   sti (bc)
until DJC          ;   djc 6:

RET

.END

```

The new pseudo-instructions are introduced in CODE fragment. In the following, *<test>* is any condition *<cc>*, as for **JR** *<cc>*.

The assembler will automatically “optimize” the jumps. If the displacements are too far to use a **JR**, the assembler will use a **JP** instead:

```

JR [<cc>,]03
JP <label>

```

In case of **until DJC**, the assembler will use:

```

DEC L
JR NC,03
JP <label>

```

2.14.1/ if ... else ... endif

```

if <test>                ; JR !<cc>,toendif
    <TRUE-clause>          ;   code executed if <cc>...
endif                    ; toendif:

```

If the *<test>* assertion is **TRUE**, the *<TRUE-clause>* is executed, else a jump to the **endif** is performed.

```

if <test>                ; JR !<cc>,toelse
    <TRUE-clause>          ;   code executed if <cc>...
else                    ; JR toendif
                            ; toelse:
    <FALSE-clause>        ;   code executed if !<cc>...
endif                    ; toendif:

```

If the *<test>* assertion is **TRUE**, the *<TRUE-clause>* is executed and a jump to **endif** is performed, else a jump to the *<FALSE-clause>* is performed to execute it.

2.14.2/ begin ... while ... repeat

```

begin                    ; tobegin:
    <BEGIN-clause>        ;   ...
while <test>              ; JR !<cc>,torepeat
    <TRUE-clause>          ;   code executed if <cc>...
repeat                   ; JR tobegin
                            ; torepeat:

```

Always execute the code between **begin** and **while**. If the *<test>* assertion is **TRUE**, the *<TRUE-clause>* is executed and a jump to **begin** is performed, else a jump to the instruction following the **repeat** is performed to exit from the loop.

2.14.3/ begin ... until

```

begin                                ; tobegin:
    <loop-clause>                       ;   code executed if !<cc>...
until <test>                           ; JR !<cc>,tobegin

```

Execute the code between **begin** and **until**. If the *<test>* assertion is **TRUE**, a jump to the instruction following the **until** is performed to exit from the loop, else a jump to **begin** is performed.

With the **begin .. until** loop, the *<test>* may be specified with **DJC**. In this case, the **JR !<cc>** is replaced by the instruction **DJC**:

```

begin                                ; tobegin:
    <loop-clause>                       ;   code executed if !<cc>...
until DJC                             ; DJC tobegin

```

2.14.4/ Force JR for displacement

These pseudo-instructions also work with the immediate assembler, but the code will be never optimized. To force the use of **JR** instead of **JP**, a **!** (exclamation) has to be added after **if!**, **else!** or **while!**. The pseudo-instructions **repeat!** and **until!** will be always optimized. If the computed displacement is over 255 bytes, an error is raised.

For example, launch **lhasm -c -T -i** and type:

```

LDA (HL)
IF! Z
INC HL
ELSE!
DEC HL
ENDIF

```

You will see:

```

1
      .ORIGIN:    40C5
1 40C5 25                LDA    (HL)

2 40C6 89 00            IF!    Z
3 40C8 64                INC    HL
4 40C9 8E 00            ELSE!
5 40CB 66                DEC    HL
4(1) 40C9 8E 03        1:else
2(1) 40C6 89 03        0:if

```

2.15/ Conditional or NOP'ed code

To write assembly sources as generic as possible, it may be useful to have some part of code to be “*assembled*” only if some conditions are **TRUE**.

The assembler provides several ways to handle “*conditional*” source code.

The directive:

```
.IF <test>
    <TRUE-code>
[ .ELSE
    <FALSE-code> ]
.ENDIF
```

will assemble *<TRUE-code>* if the *<test>* assertion is **TRUE**. If the *<test>* assertion is **FALSE** and a directive **.ELSE** is given, the *<FALSE-code>* is assembled.

The following will

```
.TEXT
.IF  MODULE?    CE163
    "BK1"
.ELSE
    "?07"
.ENDIF
```

enters a string **"BK1"** if the module is a **CE163**, else it enters the string **"?07"**.

The mnemonic:

```
EXPAND <val> <asmcode>
```

will assemble *<asmcode>* if *<val>* is not **0**.

The following:

```
EXPAND    iserror    RCF
RET
```

produce **RCF RET** if **iserror** is not **0**, else it produce only **RET**.

The directive:

```
.NOP IF <test>
    <asmcode>
.ENDIF
```

replace *<asmcode>* by **NOP** opcode (**&38**) if the *<test>* assertion is **TRUE**. If the *<test>* assertion is **FALSE** *<asmcode>* is normally assembled.

The following will

```
.NOP IF    NOT EXIST? USEBEEP
    JP     BEEP1
.ENDIF
```

produce **&38 &38 &38** if the symbol **USEBEEP** does not exist, else it produce **&BA &E6 &69**.

2.16/ Opcodes

The operator **OPCODE'**<mnemo> gives the feature to load an immediate value with the opcode of the mnemonic <mnemo>. This is useful to write sources dealing with **LH5801** opcode, like an assembler, for example.

The syntax of **OPCODE'** is:

OPCODE'<mnemo>[:<arg1>[:<arg2>]]

Where <mnemo> is a **LH5801** mnemonic, like **RET**, **STA**, **POP**, and the optional <arg1> and <arg2> may be:

&n a 8-bits value,
_ a 8-bits value (one 'underscore'),
&mn a 16-bits value,
mm a 16-bits value,
__ a 16-bits value (two 'underscores'),
(&n) a 8-bits address,
(_) a 8-bits address (one 'underscore'),
(&mn) a 16-bits address,
(mm) a 16-bits address,
(__) a 16-bits address (two 'underscores'),
R a 16-bits register: **BC**, **DE**, **HL** or **MN**,
(R) a 16-bits indirect register: **(BC)**, **(DE)**, **(HL)** or **(MN)**,
rh a high 8-bits register: **B**, **D**, **H** or **M**,
rl a low 8-bits register: **C**, **E**, **L** or **N**,
A the accumulator,
F the Flags status register,
BC the register **BC**,
PC the register **PC**,
SP the register **SP**,
+d a forward displacement,
-d a backward displacement,
cc a conditon.

Look the code of **asm/op.asm**:

```
.CODE

.ORIGIN:    40C5

LDA    OPCODE' RET
LDA    OPCODE' LDA: _
LDA    OPCODE' STA: (&mn)
LDA    OPCODE' OR: (__) : _
LDA    OPCODE' AND#: (mm) : &n
LDA    OPCODE' JR: +d
LDA    OPCODE' JR: NZ: -d
LDA    OPCODE' JR: cc: +d
LDA    OPCODE' JR: ==: -d
LDA    OPCODE' SBR: cc: (__)
LDA    OPCODE' SBR: (&n)
LDA    OPCODE' LD: SP: &mn
LDA    OPCODE' CALL: mm
```

```

LDA  OPCODE'LD:BC:SP
LDA  OPCODE'POP:A
LDA  (OPCODE'PUSH:A)
LDA  OPCODE'PUSH:R
LDA  OPCODE'INC:R
LDA  OPCODE'DEC:r1
LDA  OPCODE'DEC:rh
LDA  OPCODE'LDA:L
LDA  OPCODE'STA:C
LDA  OPCODE'LD:HL:BC
LDA  OPCODE'EVAL:+d

```

```

BYTE OPCODE'NOP
BYTE OPCODE'CPA:h

```

```

; Non-ambiguous mnemonics may be given without arguments
BYTE OPCODE'JP
BYTE OPCODE'DSBC
BYTE OPCODE'DJC

```

```

.MACRO:      DOOPCODE
             %00o .EQU OPCODE'__#0
             ; EXPAND will generate code only if <%00o is not 0
             EXPAND <%00o LDA <%00o
             EXPAND <%00o STI (BC)
             LDA >%00o
             STI (BC)
.ENDMACRO

```

```

DOOPCODE POP:HL
DOOPCODE STA:H
DOOPCODE PUSH:HL

```

```

.END

```

Note that if a mnemonic is not ambiguous, the eventual arguments may be omitted. For example, **OPCODE'JP** will return **&BA** because only one mnemonic 'JP' exists.

If a generic argument representing a register or a condition is given, the base opcode will be returned. For example, **PUSH:HL** return **&FD &A8**; but **POP:R** return **&FD &0A**, which represents the base mnemonic. In a same way **JR:cc:+d** return **&81** and **LDA:r1** return **&0A**.

Because opcodes may be 1 or 2 bytes (if it is located into the second table), the special mnemonic **EXPAND <val> <asmcode>** will assemble the **<asmcode>** only if its first argument **<val>** is not **0**. So the macro **DOOPCODE <mnemo>** declared as:

```

.MACRO:      DOOPCODE
             %00o .EQU OPCODE'__#0
             ; EXPAND will generate code only if <%00o is not 0
             EXPAND <%00o LDA <%00o
             EXPAND <%00o STI (BC)
             LDA >%00o
             STI (BC)
.ENDMACRO

```

deals properly with the mnemonics from the second table.

For example:

```
DOOPCODE    POP:HL
gives:
47           DOOPCODE    POP:HL
47           {
47 40FB %00o .EQU FD2A
47           +TRUE+      EXPAND    00FD
47 40FB B5 FD           EXPAND    <%00o LDA <%00o
47           +TRUE+      EXPAND    00FD
47 40FD 41             EXPAND    <%00o STI (BC)
47 40FE B5 2A          LDA    >%00o
47 4100 41             STI    (BC)
47           }
```

and

```
DOOPCODE    STA:H
gives:
           DOOPCODE    STA:H
           {
%00o .EQU 0028
/false/     EXPAND
/false/     EXPAND
B5 28       LDA    >%00o
           STI    (BC)
           }
```

2.17/ CSAVE headers for the CE-158 interface

When using the CE-158 interface, some headers are needed when sending or receiving a file on the SHARP PC-1500. See more informations in the **CE-158 instruction manual**, page 29.

The headers are built by the commands **CSAVE (BASIC)**, **CSAVEr (RESERVE)**, **CSAVEM (CODE)** and **PRINT** (variables assumed to **BYTE**). The same headers are expected when calling the commands **CLOAD/MERGE**, **CLOADr**, **CLOADM** or **INPUT**. Note that the commands **CSAVEa**, **CLOADa** or **MERGEa** do not need a header.

The assembler is able to build and fill properly the headers for the CE-158. This is performed by the option **-Z**.

If **-Z** is only specified, the name is filled with the source file name (up to 16 characters, without / and up to . of extension), the type is chosen according of the original fragment, and in case of **CODE**, the startup address is filled if a symbol **STARTUP** is defined inside the assembly source.

If **-Zname=<myname>** is specified, *<myname>* will be filled into the header as file name (up to 16 characters).

If **-Zentry=<startaddr>** is specified, *<startaddr>* will be filled into the header as startup address.

If **-Zheader=<type>** is specified, *<type>* will be used as header string. The valid *<type>* are:

- **CSAVE** to build the string "**@COM**",
- **CSAVEr** to build the string "**ACOM**",
- **CSAVEM** to build the string "**BCOM**",
- **PRINT** to build the string "**HCOM**".

An incorrect header type is rejected.

The base address and the length are automatically filled according to the symbols **__MAIN__\$._start** and **__MAIN__\$._length**.

For example:

lhasm -Z asm/erner1.asm will build the following header:

```
01 42434f4d 65726e65726c00000000000000000000 40c5 0018 ffff
```

lhasm -Zname="ERNERL ROUTINES" asm/erner1.asm will build the following header:

```
01 42434f4d 45524e45524c20524f5554494e455300 40c5 0018 ffff
```

WARNING: Note that a bad usage of a CE-158 header may raise some unexpected results on the SHARP PC-1500 computer !

2.18/ Full examples

The directory **asm/** contains some examples and tests sources.

Note that the example **asm/tall.as** should be assembled by this command:

```
lhasm -T -A DODO=BE -A MYSYM=E24A tall.asm
```

The option **-A DODO=BE** and **-A MYSYM=E24A** define substitute symbols. In the assembler code, the lines

```
mondodo .EQU &__ /DODO/00
```

and

```
thissym .EQU __ /MYSYM/
```

will be parsed as follow:

```
236 4145 mondodo: .EQU BE00  
251 4155 thissym: .EQU E24A
```

This will set dynamically the values of these symbols. This is useful to write a source and building several images by changing some symbols values.

To build all the tests examples, simply enters into the directory **asm** and do:

```
make tests
```

A sample binary may be built by calling:

```
make <example>.bin
```

To build the binary from the source **strgfy.asm**:

```
make strgfy.bin
```

To obtain the listing **<example>.lst** file when generating a binary file, just pass the variable **LSTFILE=yes** to **make**.

```
make LSTFILE=yes strgfy.bin
```

The listing file is named **strgfy.lst**.

3/ Re-symbol'ing' and re-sourcing

The assembler offers a facility for rebuilding source file, by adding missing symbols and rewrite the source. This is useful when dumping a binary image into a source file (**lhdump -s**) and adding symbols later.

Image the following code in **ra.bin**:

```
34 28 2A 61 6C 48 91 04 9A
```

Running **lhdump -c 40c5 ra.bin** gives:

```
40C5 34          CLA
40C6 28          STA  H
40C7 2A          STA  L
40C8 61          STI  (HL)
40C9 6C 48       CP   H,48
40CB 91 04       JR   NC,40C9
40CD 9A          RET
```

Also with the **-s** option:

```
.ORIGIN: 40C5
.CODE
    CLA
    STA  H
    STA  L
    STI  (HL)
    CP   H,48
    JR   NC,40C9
    RET

.END
.SYMBOLS:
```

Now, use the following **rs.sym** file

```
.SYMBOLS:
40c5  START
40c9  loop
40cd  END
```

to **lhdump** as follow: **lhdump -s -c 40c5 -S rs.sym ra.bin**

```
0003 symbol(s) read
    .ORIGIN: 40C5
    .CODE
START:
    CLA
    STA  H
    STA  L
    STI  (HL)
loop:
    CP   H,48
    JR   NC,loop
END:
    RET

.END
.SYMBOLS:
40CD  END
```

```

40C5  START
40C9  loop

```

This is very simple when working on the binary. But what to do with a pretty source file ?

See the code of **ra1.asm**:

```

;; Standard origin for all PC-1500
;; without module
.ORIGIN: 40C5

;; Assembly code
.CODE

;; This load accumulator with 0
CLA

;; Copy 0 to H and L
STA  H
STA  L

;; Store 0 into the address pointed
;; by HL and increment HL
STI  (HL)

;; Until H greater or equal to &48
CP   H,&48
JR   NC,-05      ;; five bytes back

;; Finish. Back to BASIC
RET

.END

```

Just put some addresses into a special symbols file **ra1.sym**:

```

40c5
40c9
40cd

```

And run the assembler with the “*re-symbol*” option (**-r ra.sym**) and “*re-source*” option (**-s raS.sym**):

```
lhasm -T -r ra.sym -s raS.asm ra1.asm.
```

This will produce a **raS.asm** file:

```

tmp_4_0c5:
;; Standard origin for all PC-1500
;; without module
.ORIGIN: 40C5

;; Assembly code
.CODE

;; This load accumulator with 0
CLA

;; Copy 0 to H and L
STA  H
STA  L

;; Store 0 into the address pointed

```

```

        ;; by HL and increment HL
        STI    (HL)
tmp_4_0c9:

        ;; Until H greater or equal to &48
        CP    H,&48
        JR    NC,-05    ;; five bytes back
tmp_4_0cd:

        ;; Finish. Back to BASIC
        RET

        .END

```

The symbols **tmp_n_xyz** are created from the *nxyz* address listed into the re-symbol file. Note that the symbols are added and the whole source is kept. Of course, the file **raS.asm** may be assembled by **lhasm**.

This feature is deprecated. To build a source file, it is better to use **lhdump -s**.

4/ lhdump - Universal dumper and sourcer

```
Usage: lhdump [-h] [-v] [{-s [-inline]|-d}] [-a] [-g]
          [-D:<dis>] [-C=[start:]addr]] [-Z]
          [-F infile] [-K infile] [-S infile] [-O addr] [fragment, ...]
          [-o outfile] infile
```

where:

```
-a          BYTE fragments are printed in HEX and ASCII
-d          Produce listing file; This is the default
-g          Use graphical character for &27 &5B &5C &5D and &7F
-h          This help
-o outfile  Write dump or source to outfile, else use stdout
-s          Produce source file; exclusive with -d
-inline     Produce \asm[ .. \]end directive if -s is active
-v          Show version and exit
-C          Compute full CHECKSUM
-C=addr     Compute CHECKSUM to addr-1 and compare to addr
-C=start:addr Compute CHECKSUM from start to addr-1 and compare to addr
-D:<inst>    In BASIC fragment <inst> are disassembled
            where <inst> is DATA, POKE, REM, VAR
            REM and VAR are disassembled only when code is found
-F fragfile Read fragment description from <fragfile>
-K kywfile  Read keyword from <kywfile>
-O addr     Origin address, else start at 0000
-S symfile  Read symbols from <symfile>
-Z          Expect and use a CE158 header if valid
```

with fragment:

```
-B [addr]   BASIC fragment; This is the default
-R [addr]   RESERVE fragment
-X [addr]   XREG fragment
-V [addr]   dynamic VARIables fragment
-c [addr]   CODE fragment
-b [addr]   BYTE (8-bits) fragment
-w [addr]   WORD (16-bits) fragment
-l [addr]   LONG (32-bits) fragment
-t [addr]   TEXT fragment
-k [addr]   KEYWORD fragment
-H [addr]   HOLE fragment
```

lhdump is the full dumper, decoder, decompiler and disassembler. It works from a binary image (created by **lhasm**) and prints the dumped source according to the options.

A special option **-v#** is available for script. It return the version of the **lhTools** on the form *x.y.z*, i.e. **0.7.5** for this revision.

-B <addr> : A BASIC image is expected. So the BASIC decompiler is called. When a BASIC image contains some ML code inside, in **REM** lines, **POKE**, variables or **DATA**, the **-D:<inst>** may be specified. Depending of the processed BASIC instruction, the LH5801 disassembler is called. Running **lhdump -B 40c5 -D:POKE te5.bin** will give:

```
10 REM \B5\00HwJPj\80A\88\03\9A
20 POKE A,&CD,&F2,&BE,&ED,&00,&9A
    ; POKE+0000 CD F2          SBR (F2)
    ; POKE+0002 BE ED 00      CALL CURMOVNCHAR
    ; POKE+0005 9A          RET
    ;
30 E$="HAJ\02\9AABCEFGH"
40 DATA &FD,&A8,&FD,&88,&BE,&E6,&69,&FD,&0A,&FD,&2A,&9A
50 END
```

Note that **-D:<inst>** may be specified several times: **-D:POKE -D:REM ...**

-c <addr> : An assembly image is expected, the LH5801 disassembler is called. Running **lhdump -c c5 te.bin** gives:

```
00C5 B5 10          LDA    10
00C7 FD C8          PUSH   A
00C9 BE E6 69       CALL  BEEP1
00CC FD 8A          POP    A
00CE DF             DEC    A
00CF 99 0A          JR     NZ,00C7
00D1 9A             RET
```

-x <addr>, **-v <addr>**, **-r <addr>** : XREGS, dynamic VARIables, RESERVE image is expected. So the decoder is called. For example, **lhdump -R 40c5 ter.bin** gives:

```
40C5 I.F5          CALL  &C5@
40CC I.F3          CALL  &30C0@
```

-b <addr>, **-w <addr>**, **-l <addr>**, **-t <addr>** : A data image is expected. So the disassembler is called. Look the call with **-b** and **-t** on the binary **te6.bin**. First as a byte fragment:

```
lhdump -b 40c5 te6.bin
40C5 48 65 6C 6C 6F 20 57 6F 72 6C 64 21 00
```

And now, as a text fragment:

```
lhdump -t 40c5 te6.bin
40C5 "Hello World!\00"
```

-k <addr> : A BASIC keyword table image is expected. Here is an example on the keyword table extracted from the **BASFILE** utility. Running **lhdump -k 4054 tek.bin** to decode the keywords:

```
4054 C7 "FCREATE"    FOB0 46EA
4060 C6 "FCLOSE"     FOB1 472C
406B C5 "FOPEN"      FOB3 4752
4075 C6 "FWRITE"     FOBE 49CF
4080 C5 "FREAD"      F06D 4A80
408A C5 "FTELL"      F06E 4C56
4094 C4 "FEOF"       F06F 4CA9
409D C5 "FSEEK"      FOBD 4B9F
40A7 D5 "GSAVE"      FOAF 4CD8
40B1 C5 "GLOAD"      FOAE 4D13
40BB D5 "MINIT"      FOA0 476C
40C5 C4 "MMEM"       F06C 455D
40CE C4 "MDIR"       FOA3 45F9
40D7 C5 "MNAME"      FOA2 45C4
40E1 C5 "MKILL"      FOA1 459B
40EB D5 "PSAVE"      FOA5 4817
40F5 C5 "PLOAD"      FOA6 485C
40FF C7 "PENDALL"    FOA4 489D
410B C5 "PCALL"      FOA7 4921
4115 C6 "PENVRN"     FOA8 48BE
4120 C7 "PRETURN"    FOA9 48DF
412C C6 "PSTACK"     FOAA 491B
4137 D4 "HEX$"       F06A 4FD5
4140 D0 ""
```

-H *<addr>*: A HOLE, i.e. an obscure area for stack, or volatile data. This area will be skip by the dumper.

-C: Computes and prints the code checksum on the whole code.

-C=[start:]end: Computes and prints the code checksum starting from *<start>* if specified, else the base of the code is taken. When *<end>* is given, the computed checksum and this stored into the ML code at the address *<end>* is compared.

-K *<keywordfile>*: Read the BASIC keyword file to produce the BASIC decompiled source. This is useful to decompile BASIC programs written with some BASIC extensions. A keyword file has the following syntax:

```
.KEYWORD:
D6 "DELETE"      F080 38C5
D4 "DISP"        F081 3930
D5 "RENUM"       F082 396E
D3 "SET"         F083 39AF
D5 "RESET"       F086 39CD
D3 "ASK"         F060 39F0
D0 ""            0000 0000
```

When calling the **lhdump** with a keyword file, the keyword information are printed:

```
39AF    [F083]  "SET"
39AF    BE 3D C5          CALL  3DC5
39B2    FD 98           PUSH  DE
39B4    BE 39 90          CALL  3990
.      .      .
39CA    FD 1A           POP   DE
39CC    E2             RST
.
39CD    [F086]  "RESET"
39CD    BE 3D C5          CALL  3DC5
39D0    FD 98           PUSH  DE
39D2    BE 39 90          CALL  3990
.      .      .
39ED    FD 1A           POP   DE
39EF    E2             RST
.
39F0    [F060]  "ASK"
39F0    D0 00 00          INTG  00,39F3
39F3    AE 7B 01          STA   (7B01)
.      .      .
```

-F *<fragfile>*: Read the FRAGMENT description from the given file. The let a mixed segment of code, data, BASIC, ... in the same binary image. Refer to the chapter 1/ **Understanding the FRAGMENT concept** for an full explanation about fragments.

-s: Produce a source file, immediately usable by **lhasm**. The symbols given by the option **-S** *<symfile>* file are fetched and disassembled within the mnemonics. If an address is referenced in the code address space without any corresponding symbol, a temporary symbol, named **lbl_<n>_<xyz>** (where *&nxyz* is the referenced address) is created and will be defined inside the source file. This gives the opportunity to re-assemble the same file later to another origin address or to modify it. Note that addresses outside the code space are kept unchanged to symbols. Note that structures or macros are not re-"sourced" by **lhdump -s**.

If **-inline** is given with **-s** and some options **-D:<inst>** are also specified and BASIC binary contains assembly code into **POKE**, **DATA**, **REM** or string variables, the assembly source will be dumped between **\asm[** and **\]end** directives for inlining.

As example:

lhdump -s -inline -D:POKE -D:REM -D:VAR -D:DATA asm/inasm.bin
will produce:

```

10 REM                \asm[
                        LDA    00
                        LD     B,77
                        LD     C,50
                        LD     L,80
                        STI    (BC)
                        DJC    -03
                        \]end
20 POKE A,            \asm[
                        SBR    (F2)
                        CALL  CURMOVNCHAR
                        RET
                        \]end
30 E$="\asm[
                        LD     B,41
                        LD     C,02
                        RET
                        STI    (BC)
                        DEC    C
                        STD    (BC)
                        LDI    (BC)
                        DEC    BC
                        LDD    (BC)
                        LD     B,22
                        \]end "
40 DATA              \asm[
                        PUSH   HL
                        PUSH   BC
                        CALL  BEEP1
                        POP    BC
                        POP    HL
                        RET
                        \]end
50 END
```

-d (default) : Produce a simple listing. If no fragment are specified, the BASIC is assumed by default. When listing code fragment, the addresses, bytes and mnemonics are printed.

Running **lhdump -c 40c5 asm/c.bin** will show:

```

40C5                A7 00 A7                CPA (00A7)
```

Note that **-d** and **-s** are exclusive.

-z : Check for a CE-158 CSAVE header, and if valid use the header information for fragment type (**CSAVE** for **BASIC**, **CSAVEr** for **RESERVE**, **CSAVEM** for **CODE** or or **PRINT** for **BYTE**), base address, and startup address (**CSAVEM**) if present.

5/ lhcom - Serial send or receive utility

Usage: ./lhcom [-h] [-v] [-d|-ddebug] [-dverbose]] [-m interface]
[-Y {[line]=[speed,size,parity,stopb]]]
[-Z[header=type]] [-Z[start=addr]] [-Z[name=headername]] [-Z[entry=addr]]
[-S symfile] binfile

where:

-d|-debug Show debug information
-dverbose Enable verbose mode
-h This help
-m interface Select the interface type
-m ce158 Use the CE158 serial interface setting and discipline
Only one -m ce... option may be given
-F fragfile Read fragments from <fragfile>
-S symfile Read symbols from <symfile>
-Y line Use <line> as serial device
-Y =speed,size,parity,stopb Set the serial settings
with <speed> : 75 100 110 200 300 600 1200 or 2400
with <size> : 5 6 7 or 8
with <parity> : N E or O
with <stopb> : 1 or 2
-Z Add a CE158 CSAVE header
-Zname=name Set <name> as CSAVE header file name
-Zstart=addr Set <addr> as CSAVE header start base address
-Zentry=addr Set <addr> as CSAVE header startup routine
-Zheader=type Set <type> for CSAVE header magic
with <type> : CSAVE CSAVEM CSAVEr or PRINT

lhcom is a transfer program to send (*upload*) or receive (*download*) programs or data using the CE-158 serial interface. **lhcom** is in charge to configure the serial line, to build if necessary the **CSAVE** header for sending and write or read data.

One of **-s** (send) or **-r** (receive) action should be specified when calling **lhcom**.

When called for receiving, the **CSAVE** header is expected to be received from the remote PC-1500. If **-z** is specified, the header is kept into the binary file received. Else, the binary file is saved without the **CSAVE** header. Note that the **CSAVE** header is useful when calling **lhdump** or to send the binary file again.

When called for sending, the **CSAVE** header has to be built, if the **-z** option is specified. If not, the **CSAVE** header is expected inside the binary file to send. When specifying a **-z** option, the same options as **lhasm** are supported by **lhcom** (see 2.17/). The start address is retrieved from the first fragment if a **-F <fragfile>** is given. The length is filled with the length of the binary file. If a symbol file is given by **-S <symfile>**, and a symbol **STARTUP** exists and the **CSAVE** header is **CSAVEM** (magic **BCOM**), the entry address is filled with the **STARTUP** address found.

The default serial line device is **/dev/ttyS0** for Unix/Linux/*BSD platforms. The default serial port is **\\.\COM1** for Window32 platforms. To specify another serial device, use the option **-Y <serial line>**.

The default line settings are **300** bauds, **8** bits, **No Parity**, **1** stop bit (**300, 8, N, 1**). These are the same as the default CE-158 parameters. To specify others line settings use the options **-Y =<speed>,<wordsize>,<parity>,<stopbit>**.

- The supported values for **speed** are **100 110 200 300 600 1200** or **2400**.

- The supported values for **wordsize** are **5 6 7** or **8**.
- The supported values for **parity** are **N** (no) **E** (even) or **O** (odd).
- The supported values for **stopbit** are **1** or **2**.

binfile is the binary file to read for sending (**-s**) or to write for receiving (**-r**).

To send a **BASIC** program from a PC-1500 to a host computer, do on the host computer:

```
lhcom -r myprog.bin
```

And on the PC-1500:

```
SETDEV CO
```

```
OUTSTAT 0
```

```
CSAVE "MYPROGBASIC"
```

In the example above, we use the default line settings.

To receive a **ML** program starting at **&40c5** from a host computer at the speed **2400**, do on the PC-1500:

```
SETCOM 2400
```

```
SETDEV CI
```

```
CLOAD M
```

And on the host computer, do

```
lhcom -s -Y =2400 -Zheader=CSAVEM -Zstart=&40c5 myml.bin
```

6/ lhpoke - Binary to BASIC converter

Usage: ./lhpoke [-h] [-v] [-x] [-xx] [-Z] [-O origin] [-A[A] appendline]
[-B byteperline] [-L linenum] [-I lineincr] [-V variable]
[-S symfile] [-o basfile] binfile

where:

-h This help
-x Values in POKE are in hexadecimal
-xx Values in POKE are in hexadecimal aligned
-A appendline Append <appendline> on the first line
-AA appendline Append a new line after the first with <appendline>
-B byteperline Write <byteperline> bytes on each line. Default 10
-I lineincr Use <lineincr> as line number increment. Default 10
-L linenum Use <linenum> as first line number. Default 10
-O address Use <address> as origin base address. Default &40C5
-S symfile Read symbols from <symfile>
-V variable Use <variable> as base address. Default A
-Z Expect and use a CE158 CSAVE header
-o outfile Output BASIC code into basfile (.bas)

lhpoke is a small utility to convert a binary file into a BASIC program using **POKE**. The aim is to offer more freedom to relocate an invariant code.

Image the following assembly program **ern.asm**:

```
.CODE
LDA (ERRORNUM)
JP &D9E4
.END
```

This code is fully invariant and could be installed at any location.

Do a **lhasm -c ern.asm** and after running **lhpoke -o ern.bas ern.bin** produce the following BASIC source:

```
10 A=197+256*PEEK &7863
20 POKE A+0,165,120,155,186,217,228
```

To enter the bytes and the offsets in hexadecimal, use the option **-x**:

```
10 A=197+256*PEEK &7863
20 POKE A+&0,&A5,&78,&9B,&BA,&D9,&E4
```

To enter the bytes and the offsets into a hexadecimal formatted form, use the option **-xx**:

```
10 A=197+256*PEEK &7863
20 POKE A+&0000,&A5,&78,&9B,&BA,&D9,&E4
```

By default, **lhpoke** uses the BASIC variable **A** for the base address. To use another variable, use the option **-V var** where *var* is a one-letter variable.

So **lhpoke -xx -V U ern.bin** gives:

```
10 U=197+256*PEEK &7863
20 POKE U+&0000,&A5,&78,&9B,&BA,&D9,&E4
```

By default, **lhpoke** start numbering the line at **10** and use an increment of **10**. To change the first line, use the option **-L linenum** and to change the increment, use the option **-I increment**.

By default, **lhpoke** prints **10** bytes for each line of **POKE**. To change this, use the option **-B nbyte** wher *nbyte* is from **1** to **16**.

By default, **lhpoke** uses origin base address as the **RAM base + 197** (i.e **&mmC5**) where **&mm** is given by the value of **&7863**. To set another base, use the option **-O address**.

To add some instruction after the variable assigned to the origin base address, use the option **-A inst1[:inst2[:...]]**. The *inst1[:inst2[:...]]* are inserted after a colon **:** on the first line. If you prefer to add them on a new line, use the option **-AA inst1[:inst2[:...]]** instead.

For example, **lhpoke -A 'INPUT "BASE ADDRESS?",A' -x ern.bin** produces:

```
10 A=197+256*PEEK &7863:INPUT "BASE ADDRESS?",A
20 POKE A+&0,&A5,&78,&9B,&BA,&D9,&E4
```

In the same way,

```
lhpoke -AA 'PRINT "BASE=";A:INPUT "BASE ADDRESS?",A' \
-O 4100 -x ern.bin
```

produces:

```
10 A=&4100
20 PRINT "BASE=";A:INPUT "BASE ADDRESS?",A
30 POKE A+&0,&A5,&78,&9B,&BA,&D9,&E4
```

If the binary contains a CE-158 **CSAVE** header, use the option **-Z**. With it, some information like the base address will be retrieved from this header.

7/ Various files formats

lhasm and **lhdump** write or read some files: fragments files, symbols files, and keywords files. These files are all text and could be created manually, for example, to explore or source a binary image with **lhdump**. In a same way, **lhasm** produces some files on request for different usages, as a dump by **lhdump**, or an export to another source assembled by **lhasm** (**.IMPORT:**).

lhasm writes the files if the following options are given on the command line:

- **-F** *<fragfile>* Write the fragment to the file *<fragfile>*,
- **-S** *<symfile>* Write all globals symbols to the file *<symfile>*,
- **-K** *<keywfile>* Write all BASIC keyword to the file *<keywfile>*. If the option is **-KK**, **lhasm** uses the old format for compatibility with the **lhTools** version **<0.6.0**.

lhdump reads the files if the following options are given on the command line:

- **-F** *<fragfile>* Reads the fragment from the file *<fragfile>*,
 - **-S** *<symfile>* Reads all symbols from the file *<symfile>*,
 - **-K** *<keywfile>* Reads all BASIC keyword from the file *<keywfile>*.
- lhdump** accepts both new or old (version **<0.6.0**) formats.

lhasm is also able to reads the symbols or keywords files with the directive **.IMPORT:**. The file names for *<fragfile>*, *<symfile>*, *<keywfile>* and are free. By convention only the following extension may be used: **.frag** for fragments files, **.sym** for the symbols files and **.keyw** for the keywords files. **lhasm** and **lhdump** does not work with the file extensions, but with the **MAGIC**.

7.1/ Fragments file

The fragments file is written each time a new fragment is created under **lhasm** with the **.BASIC**, **.CODE**, **.BYTE**, **.WORD**, **.LONG**, **.TEXT**, **.KEYWORD**, **.VAR**, **.XREG**, **.HOLE** or **.RESERVE** directives. The first **.ORIGIN:** directive set the **MAGIC** with the base origin of the binary image. The format of the fragment file is:

```
.FRAGMENTS:    <addr>
                 <fragment1>    <addr1>
                 <fragment2>    <addr2>
                 . . .
                 <fragmentN>    <addrN>
```

where **.FRAGMENTS:** is the **MAGIC**, *<fragmentN>* is one of **BASIC**, **CODE**, **BYTE**, **WORD**, **LONG**, **TEXT**, **KEYWORD**, **HOLE**, **XREG**, **VAR** or **RESERVE**, and the *<addrN>* is a hexadecimal number on 4 characters, like **40c5** or **E33F**. The file content is not case sensitive. No comment are allowed in this file.

As an example, the fragment file generated on the source test file **asm/tall.asm**:

```
.FRAGMENTS: 00C5
              CODE 00C5
```

```

CODE 00C5
BYTE 01A9
WORD 01AF
CODE 01C1
HOLE 0800
KEYWORD      0854
BYTE 0860
BYTE 4142
CODE 4145

```

Note that the fragments are printed in increasing order by the assembler, and **the fragments should be declared in increasing order**, even if this file is written manually.

7.2/ Symbols file

The symbols file is written at the end of the assembler. All symbols declared, globals AND locals, will be saved into the symbols file. The format of this file is:

```

.SYMBOLS:
  <value1>  <symbolname1>
  <value2>  <symbolname2>
  . . .
  <valueN>  <symbolnameN>

```

where **.SYMBOLS:** is the **MAGIC**, <symbolnameN> is the name of the symbol and the <addrN> is a hexadecimal number on 4 characters, like **40c5** or **E33F**. The file content is not case sensitive. No comment are allowed in this file.

As an example, a part of the symbol file generated on the source test file **asm/tall.asm**:

```

.SYMBOLS:
  4800  A
  4200  B
  0153  BC
  00EB  BIBI
  . . .
  01CA  theloop
  1234  thissym
  01C8  top

```

Note that the symbols are printed in alphabetic order when written by the assembler. But, they may be declared in any order, if the symbol file is written manually.

7.3/ Keywords file

The keywords file is written at the end of the assembler. All BASIC keywords defined will be saved into the keywords file. The format of this file is:

```

.KEYWORD:
  "<keyword1>"  <code1> <jump1> <bits1>
  "<keyword2>"  <code2> <jump2> <bits2>
  . . .

```

"<keywordN>" <codeN> <jumpN> <bitsN>

where **.KEYWORD:** is the **MAGIC**, "<keyword1>" is the name of the BASIC keyword, <codeN> is a hexadecimal number on 4 characters representing the BASIC compiled code for the instruction, <jumpN> is a hexadecimal number on 4 characters representing the "jump address" of the instruction, and <bitsN> is one of the following letter: **N**, **P**, **C** or **?**. The file content is not case sensitive, except for the "<keywordN>". Note that "<keywordN>" should be defined **between double-quote**. The keywords are printed in the order they appear in the keyword table when written by the assembler. But, they may be declared in any order, if the keyword file is written manually. No comment are allowed in this file.

As an example, the keywords file generated by the source test file **asm/tall.asm**:

```
.KEYWORD:  
  "ZWORD"      FOEF 00FD N
```

Note that **lhasm** and **lhdump** deal with both new and old format (version < **0.6.0**). The old file format is given just for compatibility with the old **lhTools** version, but the new format should always be preferred.

```
.KEYWORD:  
  <dummyhex>   "<keyword1>"   <code1> <jump1>  
  <dummyhex>   "<keyword2>"   <code2> <jump2>
```

Note that the <dummyhex> field is meaningless.

As an example, the keywords file generated on the source test file **asm/tall.asm** with option **-KK**:

```
.KEYWORD:  
  D6      "ZWORD"      FOEF 00FD
```

8/ Installation

You need **gmake**, **coreutils**, **binutils** and **gcc** to compile the **lhTools-0.7.5**.
Unzip the **lhTools-0.7.5.zip** archive and change directory to **lhTools-0.7.5/**.
Type **make install**. This will produce two executables **lhasm** and **lhdump** and install them into your **bin** directory.

Be sure that the installation directory is in your **PATH**. Call them by **lhdump** and **lhasm**.

Do not hesitate to report bugs, problems, suggestions and ask support to me. Send a email to me (**cgh75015@gmail.com**).

It is possible now to build the **lhTools** for **Windows32** with the **MinGW**. To do that, call **make** as follow:

```
make WIN32CC=<mingw32-gcc-name> win32
```

where **<mingw32-gcc-name>** is the name of the **MinGW** compiler. But you need to install the **MinGW** suite. For example, if the **MinGW** compiler is **i586-mingw32-gcc**, do

```
make WIN32CC=i586-mingw32-gcc win32
```

After, execute the **lhasm_win32.exe**, **lhdump_win32.exe**, **lhcom_win32.exe** and **lhpoke_win32.exe** under your **Windows32** platform.

NOTE TO Windows USERS

I provide this feature of cross-building the **lhTools** for **Windows32**, but, like I do not own and I do not have access to Windows platforms, **it is NOT tested**. These executables are provided as is, **WITHOUT ANY GUARANTY OF WORKING**.

Also, because the **lhTools** are developed, tested and designed to run on *nix/Linux platforms, some features may not compile or not work on **Windows32**.

Do not hesitate to contact me or to request support from myself if you are interested to do a port of the **lhTools** under **Windows32/64** platforms.

9/ Incompatibilities with older versions

With this release **0.6.x**, some incompatibilities with older versions are introduced:

- The option **-e** (verbose) is replaced by **-dverbose**.
- The option **-K <keywfile>** produces a keyword file in a new format not understandable by the old versions < **0.6.0**. To keep backward compatibility, use **-KK <keywfile>**. The assembler and the dumper are able to deal with both formats.
- The **.DEFINE:** directive has changed, and now a *<bits>* parameter may be specified. Note that the assembler still understood the old directive format.
- The **.KEYWORD** fragment has also changed, and only the BASIC keyword has to be specified to be introduced into the keyword table. Note that the assembler still accepts the old **.KEYWORD** fragment syntax.

Of course, due to new directives, features and special symbols introduced, the sources written for the **0.6.0** version may be not assembled by older **lhTools** versions.

With releases older than **0.7.2**, the **RLD** and **RRD** mnemonics are not assembled if a # (page **&FD**) is specified. So the following code is not correctly generated:

```
.CODE
RRD
RLD
RRD#
RLD#
.END
```

should now give:

```
1      .CODE 40C5
2
      .ORIGIN: 40C5
2 40C5  D3      RRD
3 40C6  D7      RLD
4 40C7  FD D3   RRD#
5 40C9  FD D7   RLD#
6 40CB  .END
```

In the same way, for older releases, **lhdump** did not correctly disassemble these instructions when found in page **&FD**:

```
40C5  D3      RRD
40C6  D7      RLD
40C7  FD D3   RRD#
40C9  FD D7   RLD#
```

The **-z** flag and **-z<type=value>** options are introduced in the version **0.7.2**.

The utilities **lhcom** and **lhpoke** are introduced in the version **0.7.4**.

The directives **.TIMESTAMP** and **.DATETAMP** are introduced with **0.7.5**. Also, the directive **.CHECKSUM <expr>** is introduced within this release.

10/ License

Copyright 1992-2015 Christophe Gottheimer <cgh75015@gmail.com>

lhTools is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License version 2 as published by the Free Software Foundation. Note that I am not granting permission to redistribute or modify **lhTools** under the terms of any later version of the General Public License.

This program is distributed in the hope that it will be useful (or at least amusing), but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program (in the file "COPYING"); if not, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.