MACBAS'2014

December 2014 - April 2015

Macro-assembler, monitor, debugger and sourcer the SHARP PC-1500/A and TRS80 PC-2

MACBAS2014 is a macro-assembler, monitor, debugger and sourcer for the SHARP PC-1500/A and TANDY PC-2.

This version is still in beta release. It is fully mature but bugs may be still 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.     |
+---------------------------------------------------+
```

MACBAS2014 is written in assembly language. It was firstly developed directly on a SHARP PC-1500 with the software **`XMON`**.

MACBAS2014 requires the **lhTools** version **`0.7.2`** or higher to be assembled from the sources.

IMPORTANT NOTE

MACBAS2014 is **NOT** compatible with the **CE-150** printer or the **CE-158** serial interface because **MACBAS2014** uses the same memory area for its pointers than these peripherals. Also some BASIC commands have the same internal code between a **CE-158** and **MACBAS2014**. However note that **MACBAS2014** is **fully compatible** with the **CE-150** or **CE-162** audio interfaces. Historically, this was because I do not own such interface on the first development.

- If you expect to use a **CE-150** or a **CE-158**, you should first *"exit"* from **MACBAS2014**. To do so, you may *"reset"* the computer by a **`CALL &E000`** or just by a **`POKE &79D4,0`** and a **POWER-OFF/ON**.
- The line settings of the **CE-158** should be reinitialized by a **`SETCOM`**.
- The **CE-150** settings should also be reinitialized by a **POWER-OFF/ON**.
- To return to **MACBAS2014**, just do

```
POKE &79D1,&8
POKE &79D4,&55
```

When returning to **MACBAS2014**, its pointers and LM registers are corrupted. Be sure to reinitialize them with **`REGS`** and **`DEF+(up-arrow)`**.

1/ Installation

MACBAS2014 requires a ROM version **A02** or higher. Try a **PEEK &E2B9**. If you get **56**, you have a good ROM to run **MACBAS2014**. Else, it will be not possible to run the software. Note that all PC-1500A have always the good ROM.

MACBAS2014 needs **8 Kbytes** of amount of memory. The remaining memory is free for assembler sources and binaries. **MACBAS2014** is provided with an image for each modules: **CE-151** (requires a PC-1500A), **CE-155**, **CE-159** and **CE-161/163** (same image).

Before loading the image of **MACBAS2014**, be sure to save your work and all programs or variables.

To load **MACBAS2014**, switch to **PRO** mode and do:

- **CE-151:** **NEW &6000**
- **CE-155:** **NEW &4800**
- **CE-159:** **NEW &4000**
- **CE-161/163:** **NEW &2000**

Note that the address after the **NEW** will set the base address for the BASIC program and reserve some space free. You may specify another address greater than this specified, but not below, else the BASIC program you enter will overwrite **MACBAS2014**. It is highly recommended to reserve also some space for binary code and the symbols table. If you want to reserve **&800** (2kbytes) for assembly code and symbols, change the values to **&6800**, **&5000**, **&3800** or **&2800** respectively.

The range addresses reserved for **MACBAS2014** is shown below (full images):

- **CE-151:** **&40C5..&5FFF**
- **CE-155:** **&38C5..&47FF**
- **CE-159:** **&20C5..&3FFF**
- **CE-161/163:** **&00C5..&1FFF**

The range addresses reserved for **MACBAS2014** is shown below (reduced images):

- **CE-151:** **&40C5..&5D7F**
- **CE-155:** **&38C5..&557F**
- **CE-159:** **&20C5..&3D7F**
- **CE-161/163:** **&00C5..&1D7F**

1.1/ Images naming

MACBAS2014 is delivered with 2 kinds of images: **`.wav`** for audio download with the **CE-150** or **CE-162** cassette interface and **`.bin158`** for a serial download with the **CE-158** serial interface.

The full images come with the **`SRCS`** command. The reduced images without; this give 640 bytes more free for small memory module.

The 2 keyboard layouts are also provided. Choose **`k1`** or **`k2`** according the layout you prefer.

The images naming is **`Images/full/macbas2014-k`***`N`***`-ce`***`MMM`***`.`***`XXX`** for full image and **`Images/reduced/macbas2014-k`***`N`***`-ce`***`MMM`***`.`***`XXX`** where *`N`* is the keyboard layout (**`1`** or **`2`**), *`MMM`* is the module (**`151`**, **`155`**, **`159`** or **`161`** [also for **`CE-163`**]), and *`XXX`* is **`.wav`** for audio or **`.bin158`** for serial.

The **`.bin158`** are binary images with the **CE-158** header included to a download with the **`CLOAD M`** command.

1.2/ Audio download

Connect the **PC-1500** to a **CE-150** or **CE-162** audio cassette interface and plug the audio jack wire.

After, enter a **`CLOAD M`** command and start to play the WAV file. After 9 minutes, **MACBAS2014** is loaded. See **1.4/** to start **MACBAS2014**.

- When loading the **full** WAV image, **`MACBAS"14:K`***`k`***`F`***`ccc`* is displayed where *`k`* is **`1`** or **`2`** depending of the keyboard layout and *`ccc`* is the module: **`151 155 159`** or **`161`**.
- When loading the **reduced** WAV image, **`MACBAS"14:K`***`k`***`r`***`ccc`** is displayed where *`k`* is **`1`** or **`2`** depending of the keyboard layout and *`ccc`* is the module: **`151 155 159`** or **`161`**.

1.3/ Serial download

Connect the **PC-1500** to a **CE-158** interface and plug the serial wire between the host computer and the interface.

On host, configure the serial line parameters with **2400** bauds, **8 bits**, **No parity** and **1-bit stop**: **`2400/8/N/1`**.

Switch the **CE-158** interface ON and after the **PC-1500**. Configure the serial parameters and set the device in input mode:

```
SETCOM 2400,8,N,1
SETDEV CI
OUTSTAT 0
CLOAD M
```

Start the transfer on the host PC. After less than 1 minute, **MACBAS2014** is loaded. See **1.4/** to start **MACBAS2014**.

1.4/ Initialization

Depending of the images you have loaded, the following steps need to be done to initialize **MACBAS2014**, but the **reduced** and **full** images share the same initializations steps.

CE-151:

```
POKE &785B,&4F,&FD
POKE &79D1,&28
POKE &79D4,&55
```

CE-155:

```
POKE &785B,&47,&FD
POKE &79D1,&24
POKE &79D4,&55
```

CE-159:

```
POKE &785B,&2F,&FD
POKE &79D1,&18
POKE &79D4,&55
```

CE-161 and **CE-163**:

```
POKE &785B,&0F,&FD
POKE &79D1,&08
POKE &79D4,&55
```

Type the **POKE**'s addresses and values very carefully, because a mistake may crash the **PC-1500** and the whole memory may be corrupted or lost !

Once done, just do **DEF+OFF** :

- if the computer switch OFF, the keyboard is not installed or is not working (**ROM A01**, see warning at **1**/),
- if the computer resets (see **NEW0? :CHECK**), the check sum is not correct, so the loaded image is corrupted,
- else:

Welcome to MACBAS2014 !

2/ Symbolic macro-assembler

MACBAS2014 is a symbolic macro-assembler. It is able to assemble a complete source in from a BASIC program or to assemble directly the mnemonics from the command line.

When called from keyboard input, **MACBAS2014** will assemble directly and display the code.

Be careful to initialize first the assembler pointers, else MACBAS2014 will work at a unknown address.

To do so, switch to **PRO** mode, type an address followed by **DEF+(up-arrow)**. If you assemble a BASIC program, the assembler pointers are initialized by the **asm** directive (see **2.2**/).

Because the mnemonics are in **lowercase**, be sure to follow the case hereafter.

For example, switch to **PRO** mode, do **CLEAR**, enter the following address **&7900** and do **DEF+(up arrow)**. You will see:

```
7900:sbc c
```

The current instruction at **&7900** (i.e **&00**) is disassembled.
Now, type **nop ENTER** and you see:

```
7900:nop
```

This is the command or the immediate mode. Note that it is not possible to enter assembly in **RUN** mode (**MASBAS2014** will raise an **ERROR 101**).

Enter the following program:

```
10 asm &7900,&790F,&7910,&791F
20 call &E669:ret
30 end
```

Now switch to **RUN** mode and do **RUN**. When the **>** is back, the program is assembled. Try a **CALL &7900** and you ear a beep.
Switch back to **RUN** mode and do **DEF+(down arrow)** and you see:

```
7900:call &e669
```

Press **(down arrow)**, and

```
7903:ret
```

will be displayed. Press **(up arrow)** will return to the **&7900**.

Now, modify the program as follow:

```
10 asm &7900,&790F,&7910,&791F
15 defs "BEEP1"=&E669
20 call "BEEP1":ret
30 end
```

Now switch to **RUN** mode and do **RUN**. When the **>** is back, the program is assembled. Try a **CALL &7900** and you ear a beep. But you have defined the symbol **"BEEP1"** with the value **&E669**. Do **SMBL LIST** and you see:

```
BEEP1                    &E669
```

Press (**down arrow)** to continue or **CL** to exit.
Switch back to **RUN** mode and do **DEF+<down arrow>** and you see:

```
7900:call "BEEP1"
```

Now, switch back to **PRO** mode and do **"BEEP1" DEF+(up arrow)** and you discover:

```
E669="BEEP1":ld l,&08
```

You are inside the ROM routine **BEEP1** at the address **&E669**. In a same way, you may also call the routine by a **CALL GETS "BEEP1"** !

The full images comes with the sourcer command SRCS. This will convert an assembly code into a BASIC source. For example, the following will source the the routine **&ED95** (convert the hexadecimal string into a 16-bits number). To do so, switch to **PRO** mode and do:

```
NEW
SRCS &ED95,&EDAA
```

After a couple of seconds, the following BASIC source code is available:

```
push hl
ldi (bc)
call &ED7D
jr nc,"L001"
swp
sta h
ldi (bc)
call &ED7D
jr nc,"L001"
rcf
adc h
scf
"L001":pop hl
ret
```

2.1/ Mnemonics

The following mnemonics are understood by **MACBAS2014**:

```
adc[#]      (R)
adc         rl
adc         rh
adc[#]      (&mn)
adc         &n
add         R
add[#]      (R),&n
add[#]      (&mn),&n
and[#]      (R),&n
and[#]      (R)
and[#]      (&mn),&n
and[#]      (&mn)
and         &n
atp
am0
am1
bit[#]      (R),&n
bit[#]      (R)
bit[#]      (&mn),&n
bit[#]      (&mn)
bit         &n
call        &mn
cdv
cpa[#]      (R)
cpa         rl
cpa         rh
cpa[#]      (&mn)
cpa         &n
cpi
cp          rl,&n
cp          rh,&n
dadc[#]     (R)
dec         A
dec         rl
dec         rh
dec         R
di
djc         &d
dsbc[#]     (R)
ei
halt
inc         A
inc         rl
inc         rh
inc         R
ita
jr          cc,&d
jr          &d
jp          &mn
lda[#]      (R)
lda         rl
lda         rh
```

```
ldi[#]      (R)
ldi
ld          rl,&n
ld          rh,&n
ld          BC,R
ld          BC,PC
ld          BC,SP
ld          R,BC
ld          SP,BC
ld          PC,BC
ld          SP,&mn
nop
or[#]       (R),&n
or[#]       (R)
or[#]       (&mn),&n
or[#]       (&mn)
or          &n
off
pop         A
pop         R
push        A
push        R
rcf
rdp
ret
reti
rl
rr
rld
rrd
rpu
rpv
sbc[#]      (R)
sbc         rl
sbc         rh
sbc[#]      (&mn)
sbc         &n
sbr         (&n)
sbr         cc,(&n)
scf
sdp
sl
sr
spu
spv
sta[#]      (R)
sta         rl
sta         rh
sta         F
sta[#]      (&mn)
std[#]      (R)
sti[#]      (R)
```

```
lda         F           |   swp
lda[#]      (&mn)       |   xor[#]      (R)
lda         &n          |   xor[#]      (&mn)
ldd[#]      (R)         |   xor         &n
```

The syntax of all **sbr** subroutines is also supported by **MACBAS2014**.

```
sbr (&00),&n,&n',&d     |   sbr (&C2),&k,&d
sbr (&02),&n,&n',&d     |   sbr (&C4),&k,&d
sbr (&04),&d            |   sbr (&C8),&d
sbr (&08),&d            |   sbr (&CA),&n
sbr (&0E),&n,&d         |   sbr (&CC),&n
sbr (&10),&n            |   sbr (&CE),&n,d
sbr (&1A),&d            |   sbr (&D0),&n,d
sbr (&1C),&n            |   sbr (&D2),&d,n
sbr (&26),&d            |   sbr (&D4),&n
sbr (&28),&d            |   sbr (&D6),&n
sbr (&2A),&n,&n'        |   sbr (&DE),&d
sbr (&2C),&d            |   sbr (&F4),&mn
sbr (&2E),&d            |   sbr (&F6),&mn
sbr (&34),&n,&n',&d',...,&n,&d
```

*Note: The **sbr** calls not listed above do not take arguments.*

Some instructions are specific to **MACBAS2014**:

ld BC,&mn Assembled as **ld C,&n:ld B,&m**
ld DE,&mn Assembled as **ld E,&n:ld D,&m**
ld HL,&mn Assembled as **ld L,&n:ld H,&m**

The conditions **<=** and **>** are supported:

jr <=,&d Assembled as **jr Z,&d:jr NC,&d**
jr >,&d Assembled as **jr Z,+2:jr C,&d**

Convention for the mnemonics described above:

&n Byte 8-bits value, within **0..255 (&FF)**
&mn Word 16-bits value, within **0..65535 (&FF)**
(&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**
rl Low 8-bits register: **C**, **E**, **L**
R Whole 16-bits register: **BC**, **DE**, **HL**
(R) Indirect whole 16-bits register: **(BC)**, **(DE)**, **(HL)**
A Accumulator
F Flags (status)
PC Program counter
SP Stack pointer
[#] Optional second page access
&k BASIC keyword code if **k >= &E000** else a 8-bit value is assumed

*Note: The undocumented register **MN** is not supported.*
Note: Conditions and registers may be entered in both uppercase or lowercase.

The values **&n**, **&mn**, **&d** or **&k** may be any expressions of the BASIC evaluator.

```
lda PEEK (&764E) AND 3:jr "loop":call PEEK (&785B)*256+PEEK (&785C)
```

The mnemonics and the commands **byte**, **word** and **text** are callable directly from keyboard in **PRO** mode (command or immediate mode). The mnemonics are assembled in **PRO** mode or execute a **RUN** in **RUN** mode to starts the assembler, if a **asm ... end** block is present in the **BASIC** program. When the assembler is working from a BASIC program, more functionalities, like define symbols or the structured programmation, are available.

The registers **A**,**B**,**C**,**D**,**E**,**H**,**L**,**F**,**BC**,**DE**,**HL**,**PC**,**SP**,**(BC)**,**(DE)**,**(HL)** are different from the corresponding BASIC variables. If you want to use the value of BASIC variable instead of a register, you may do it by writing **0+***RRR* to avoid an incorrect instruction.

For example:

```
A=10
lda A
```

will be rejected by the assembler. Do instead:

```
lda 0+A
```

2.2/ Assembler directives

Note that these directives are only usable while running the assembler inside a BASIC program and should be entered in **PRO** mode.

asm <*start-code*>,<*end-code*>**[,[**<*start-symb*>,<*end-symb*>**]**
[,<*store-code*>**]**

Activate the assembler. Instructions after this directives will be assembled. If some arguments are not specified, the previous values are used.
The assembler will write the assembled code starting <*start-code*> and until <*end-code*> is reached. Code can not be written outside the code area.
If symbols need to be defined, the symbol area is between <*start-symb*> and <*end-symb*>.
The <*store-code*> is address at which the code will be stored. This gives the possibility to assemble a code for another place; for example, a code running at the place of **MACBAS2014**.

asm CONT

Activate the assembler reusing the previously fixed addresses. This provided to add code and symbols.

end

Finish the assembler. When running a program, instructions outside the block **asm .. end** will raise an error.

defs "*symbol-name*"**[=**&*mn***]**

Defines a symbols named "*symbol-name*" to the given value &*mn*. If no value is specified the current assembler address (i.e **THIS**) is taken.

2.3/ Assembler variables

`THIS` Function returns the current assembler address.

`PASS` Function returns the current pass of the assembler, i.e, **`1`** for the first pass, **`2`** for the second, **`0`** if the assembler is not active. This is very useful to write a source with actions performed only in pass 1 or 2.

2.4/ Assembler data

byte *&n*|**&mn*[,*&n*|**&mn*[,...]]
Store 8-bits values. If ***&mn** is given, compute the displacement between the current address of the byte and the given address *&mn*.

word *&mn*[,*&mn*[,....]]
Store 16-bits values.

text "<*text*>"[,"<*text*>"[,...]]
Store text strings

2.5/ Values

HEX$ *&mn*
Returns the hexa-string of the *&mn* 16-bits value.

HIGH *&mn*
Returns the high 8-bits value of the *&mn* 16-bits value.

LOW *&mn*
Returns the low 8-bits value of the *&mn* 16-bits value.

2.6/ Structured code

The following instructions are designed to write structured code, and also to economize symbols. The condition **cc** is working on the status register F, as performed by **JR cc**.

Note that these instructions are only usable while running the assembler inside a BASIC program and should be entered in **PRO** mode.

if [#]cc <*TRUE-code*> **[else** <*FALSE-code*>**] endif**
Execute <*TRUE-code*> if the condition **cc** is **TRUE**, if not, execute <*FLASE-code*>. Note that the <*FALSE-code*> section may be omitted.

begin <*LOOP-code*> **until [#]cc|djc**
Execute <*LOOP-code*> until the condition **cc** is TRUE.

begin <*CONDITION-code*> **while [#]cc** <*LOOP-code*>**> repeat**
Execute <*CONDITION-code*> and while the condition **cc** is TRUE, execute <*LOOP-code*>.

If **#** precedes the condition **cc**, this will inform the assembler that the jump will go over the **255** bytes limit and tell the assembler to use absolute jumps (**jp**) instead of relatives (**jr**).

See some examples in the chapter **4/**.

2.7/ Symbols

With **MACBAS2014**, a symbol is a string **"<*symbol-name*>"** and a value. All symbols are 16-bits values. When defining a new symbol, a name (**"<*symbol-name*>"**) should be given, and optionally a 16-bits value. If the value is omitted, the current assembler address (**THIS**) is assigned to the symbol.

If a symbol already exists, it is not possible to change its value. Symbols are defined during the pass **1** of the assembler. The pass **2** generates the code with the good values.

A symbol name may contain any characters in the range **&20** to **&7F**. The minimum length for a symbol name is 2 characters, and the maximum length is **80** characters.
Note the two exceptions below:

1. Symbol names starting by a space (**&20**) are not treated as symbols and so may be used for BASIC or for indentation,
2. Symbol names of only one character are skipped and are so reserved for direct BASIC call (**DEF+**<*letter*>).

The sourcer **SRCS** creates symbols of the form **"L***nnn***"** where *nnn* is a 3 digits decimal number. Avoid to use symbols with these name if you expect to run **SRCS**.

GETS "*symbol-name*"
: Return the value of the symbol "*symbol-name*". If the symbol does not exist, **-1** is returned.

SMBL ON [*&mn***]**
: Active (or re-activate after a **SMBL OFF**) the symbol table defined before with **asm**. If *&mn* is given, the symbol table starting at *&mn* is used.

SMBL OFF
: Deactivate the symbol table.

SMBL LIST [*&mn***]**
: List all symbols on the screen using the working symbol table. Press **(down-arrow)** to list the next symbol, and **CL** or **ON/BREAK** to exit. If *&mn* is given, the symbol table starting at address *&mn* is used.

2.8/ ML execution

REGS *RRR*=*&mn*|*RRR* **TO** *var*[,*RRR*=*&mn*|*RRR* **TO** <*var*>[,...]]
Load register *RRR* with the value *&mn* or save the register *RRR* value to the BASIC variable <*var*>. *RRR* is one of the register **A**, **F**, **BC**, **DE**, **HL**, **PC** or **SP**.

EXEC [TRON] *&mn*
[,<*valSP*>][;[<*valA*>][,[<*valBC*>][,[<*valDE*>][,[<*valHL*>]]]]
[;[<*varA*>][,[<*varBC*>][,[<*varDE*>][,[<*varHL*>]]]]]]
Launch execution of the ML program at the address *&mn*, with optionally setting the stack **SP** from <*valSP*>, and optionally storing the given value into **A**, **BC**, **DE** and **HL**. from <*valA*>, <*valBC*>, <*valDE*> and <*valHL*> respectively.
When the execution ends, optionally the values of the registers **A**, **BC**, **DE** and **HL** may be saved into the BASIC variables <*varA*>, <*varBC*>, <*varDE*> and <*varHL*>.
If **TRON** is given, the execution does not start, but the debugger is activated. When the debugger is active, pushing **ON/BREAK** while the execution will break the program, showing **BREAK AT** *&mn*. Also, if the stack goes outside its space, the program will abort by **ERROR AT** *&mn*.
See the Keyboard driver section 3. below for further explanation about the debugger.

EXEC CONT [*&mn*]
Continue execution after a break. Start at the address *&mn* if it is specified.

EXEC CONT
Deactivate the ML debugger.

EXEC RUN *&mn*[;*args,....*]
Launch execution of the ML program at the address *&mn*, but the eventual arguments *args,....* may be parsed as any BASIC commands, like **POKE**. This is useful to implement new BASIC commands. Note that is not possible to create a function, like **PEEK** for example.

bkp
Install a special **call** instruction to the debugger entry. When LM code is executing, this call will enter the debugger and stop; this is *"Break Point"*. This is very useful to debug LM programs. When the debugger is entered, the registers are saved, and it is possible to execute step-by-step with the embedded debugger (see chapter **3.3**).

2.9/ Sourcing code into BASIC program (full images only)

SRCS <*start-code*>,<*end-code*>**[,**<*start-data*>,<*end-data*>
[,<*start-symb*>**[,<**_line-num_**>[,<**_increment_**>]]]]**
Build a BASIC program using the ML code from <*start-code*> to <*end-code*> using the symbol table at <*start-symb*> (or the current working symbol table if omitted), create a byte or text area from <*start-data*> to <*end-data*> if specified. The first line has the number specified by **<***line-num*> (or **10** if omitted) and increment each line by **<***increment*> (or **10** if omitted).
When symbols are found into the table, the disassembler will use them.
When making a jump (absolute or relative) without any symbols found, the disassembler will build a symbol **"L***nnn***"** where *nnn* is incremented at each new symbol.

Note that the reduced images do not support the **SRCS** command.

3/ The keyboard driver

With **MACBAS2014** a fully enhanced keyboard driver is provided. There are 3 modes with this new driver

3.1/ The normal mode

With this new driver, all keys have auto-repetition, also direct or **SHIFT** or **DEF** key are pressed.

In all modes (**RUN**, **PRO**, **RESERVE**), the following key extension are usable:

DEF OFF	Compute the check sum of **MACBAS2014** and perform a soft RESET if code is corrupted,
DEF CL	Clear the line from the cursor position to the end of line,
DEF MODE	Jumps from **:** to **:**,
DEF (left-arrow)	Go to the beginning of the line,
DEF (right-arrow)	Go to end of the line,
SHIFT INS	Activate or deactivate the auto-insertion mode.

3.2/ The mnemonics shortcut

When in **PRO** or **RESERVE** mode, a mnemonics shortcut keyboard is redefined. To enter easier the main mnemonics, press **DEF+**<*key*> with the layout shown below:

The both layout are provided, depending of the image loaded.

3.2.1/ Layout 1:

```
ldd  ldi  lda  ld   adc  sbc  and  or   xor  bit byte word text  sr
 Q    W    E    R    T    Y    U    I    O    P    7    8    9    /
std  sti  sta  cp   cpi  cpa  jp   jr   djc       if  else endif sl
 A    S    D    F    G    H    J    K    L         4    5    6    *
inc  dec  add call  sbr push  pop  scf  rcf   begin while repeat rr
 Z    X    C    V    B    N    M    (    )         1    2    3    -
               ret                              until  swp defs  rl
              SPACE                               0    .    =    +
```

3.2.2/ Layout 2:

```
byte word text  if else endif begin while repeat until  cp  cpi  cpa  djc
 Q    W    E    R    T    Y    U    I    O    P    7    8    9    /
adc  sbc  add  and  or   xor  bit  inc  dec       std  sti  sta  jr
 A    S    D    F    G    H    J    K    L         4    5    6    *
swp  sr   sl   rr   rl   scf  rcf push  pop      ldd  ldi  lda  jp
 Z    X    C    V    B    N    M    (    )         1    2    3    -
               defs                              ld  ret call  sbr
              SPACE                               0    .    =    +
```

3.3/ The debugger/monitor keyboard

When in **PRO** mode, the following key extension are usable:

DEF (up-arrow)	Read the address in the input buffer (like **AREAD**) and start disassembling at the given address,
DEF (down-arrow)	Start disassembling at the last address, or at the start address of the assembler, ie, <*start-code*>,
(up-arrow)	One instruction, 8 bytes or 16 characters backward,
(down-arrow)	One instruction, 8 bytes or 16 characters forward,
(left-arrow)	One byte backward,
(right-arrow)	One byte forward,
DEF RCL	Show the general registers **PPPP:BBCC DDEE HHLL AA FF**
SHIFT RCL	Show the status registers **PPPP:SSSS aaaaaaaa hvzic** with **aaaaaaaa** the binary representation of the accumulator **A** and **hvzic** the binary representation of the status register **F**, lower case if cleared, upper case if set,
DEF MODE	Switch between instruction and text display,
SHIFT MODE	Switch between instruction and byte display,
SHIFT CA	Deactivate the disassembler.

When in **RUN** mode, after the debugger was activated by **EXEC TRON**, the following key extension are available:

(up-arrow)	Show the current instruction,
(down-arrow)	Execute the current instruction,
DEF (up-arrow)	Shows the stack **SSSS->xx yy ...()**,
DEF (down-arrow)	Execute a **call** as a standalone instruction,
DEF (double-arrow)	Execute up to a **ret** instruction,
DEF (doublea-rrow)	Continue execution,
SHIFT CA	Deactivate the debugger.

4/ Examples

In this chapter, we assume that a **MACBAS2014** image for a **CE-161** has be loaded.

First we will reserve some space from **&2000** to **&2200** for ML code and symbols. In **PRO** mode, just do:

```
NEW &2200
```

Type the following code (note that the indentation is here only to have a pretty view):

```
10 asm &2000,&20FF,&2100,&21FF
20 "STRLOWER"dec l
30 if C
40   begin
41     lda (bc)
42     cpa ASC("A"):jr nc,"NUPPER"
43     cpa ASC("Z")+1:jr c,"NUPPER"
44     or ASC("a")-ASC("A"):sta (bc)
45     "NUPPER"inc bc
46   until djc
50 endif
60 scf: ret
70 end
80 END
```

Return to **RUN** mode and launch **RUN**, and wait for 3 seconds. The code will be assembled.

In **PRO** mode, doing **DEF (down-arrow)** will show:

```
2000="STRLOWER":dec l
```

Use **(down-arrow)** to see the next instruction:

```
2001:jr nc,&2012
```

Now execute it (**A$** is at address **&78C0** and the string has 16 characters).

```
A$="TH!S 1s -MaCBaS-"
EXEC &2000;,&78C0,,16
A$
```

and you will show:

```
th!s 1s -macbas-
```

Now switch to **RUN** mode, and do

```
A$="HeLLo WoRLD!"
```

We will execute it step by step by doing:

```
EXEC TRON &2000;,&78C0,,12
```

You see:

```
2000="STRLOWER":dec l
```

Pressing the **(down-arrow)** key will execute the pointed instruction.

```
2001:jr nc,&2012
2003:lda (bc)
2004:cpa &41
```

Pressing **DEF+RCL** will commute to *REGISTER* display. You see:

```
2004:78C0 78C0 000B 48 13
```

Registers are shown as: *PCPC:BBCC DDEE HHLL AA FF*

Pressing **Shift+RCL** will commute *STATUS REGISTER* display. You see:

```
2004:7BAF 01001000 HvzIC
```

Registers are shown as *PCPC:SPSP aaaaaaaa fffff* where *aaaaaaaa* is the binary representation of the accumulator **A** and *fffff* are the flags (uppercase if set, lowercase if cleared).

Pressing **DEF+RCL** go back to instruction display.

```
2004:cpa &41
2006:jr nc,"NUPPER"
2008:cpa &5B
200A:jr c,"NUPPER"
200C:or &20
200E:sta (bc)
```

Press **DEF+RCL**, you see that **A** is now **68** (i.e **"h"**). Continue the program by pressing **(down-arrow)** until you reach:

```
2012:scf
2013:ret
```

If you execute the **ret**, you get **ERROR AT &2013**. This is because when executing a ML program step-by-step, no return address is pushed on the stack, but the debugger *"knows"* that the **ret** goes nowhere and raises an error.

Now do **A$** and you see:

```
hello world!
```

Also, it is possible to execute step-by-step by displaying the registers instead of the instructions:

```
A$="Z"
EXEC TRON &2000;,&78C0,,1
```

You see:

```
2000="STRLOWER":dec l
```

Do **DEF+RCL** and execute the program by pressing **(down-arrow)** and you will see:

```
2000:78C0 78C0 0001 21 13
2001:78C0 78C0 0000 21 17
2003:78C0 78C0 0000 21 17
2004:78C0 78C0 0000 5A 13
2006:78C0 78C0 0000 5A 13
2008:78C0 78C0 0000 5A 13
200A:78C0 78C0 0000 5A 02
200C:78C0 78C0 0000 5A 02
200E:78C0 78C0 0000 7A 02
200F:78C0 78C0 0000 7A 02
2010:78C1 78C0 0000 7A 02
2012:78C1 78C0 00FF 7A 02
2013:78C1 78C0 00FF 7A 03
ERROR AT &2013
```

Now display **A$** and see:

```
z
```

The registers values may be also fixed by the command **REGS**. Now we use the variable **D$** (address **&78F0**). In **RUN** mode, do

```
Shift+CA
REGS BC=&78F0,HL=6
D$="MACBAS"
EXEC &2000
```

When returning to the BASIC prompt, **D$** will show:

```
macbas
```

If you press the **DEF+RCL**, you will also see:

```
4000:78F6 78C0 00FF 73 05
```

No switch in **PRO** mode and do

```
&78F0 (up-arrow)
```

and you will see:

```
78F0:bit (hl),&61
```

Pressing **Shift+MODE** enters the *BYTE* mode:

```
78F0:6D61636261730000
```

And finally **DEF+MODE** enters the *TEXT* mode:

```
78F0:"macbas~~~~~~~~~~"
```

Press **Shift+CA** to exit from the disassembler

Now, switch to **PRO** and do **NEW** to clear the current program, and call the sourcer **SRCS**:

```
NEW
SRCS &2000,&2013,,&2100
```

And now do **LIST** and use the **(down-arrow)** to show the ML program sourced:

```
LIST
10:"STRLOWER":dec l
20:jr nc,"L002"
30:"L001":lda (bc)
40:cpa &41
     ...etc...
130:ret
```

Note: This source code is provided in the **strupper.bas** file. Use the **strupper.wav** to load it.

Another example to implement a *FOR..STEP..NEXT* loop:

```
10 asm &2020,&20FF
20 lda &21
30 begin
40   cpa &80
50 while <
60   sti (bc)
70   rcf:adc &10
80 repeat
90 scf:ret
100 end
```

This is the equivalent of the BASIC program:

```
10 B$="":FOR I=&21 TO &80 STEP &10:B$=B$+CHR$ (I):NEXT I:END
```

Switch to **RUN** mode, and launch the program by **RUN**. Now execute the LM routine. This program will fill the string pointed by **BC** with **!1AQaq** (The variable **B$** is located at the address **&78D0**).

```
B$=""
EXEC &2020;,&78D0
B$
```

Switch in **PRO** mode, and do:

```
&2020 DEF (up-arrow)
```

You see:

```
2020:lda &21
```

If you enter directly:

```
lda &24
```

You will see now:

```
2020:lda &24
```

Rerun the ML program using **C$** (located at address **&78E0**):

```
C$=""
EXEC &2020;,&78E0
C$
```

and see **$4DTdt**

By the way, you can also do:

```
D$=""
REGS BC=&78F0,A=&26
EXEC &2022
D$
```

and you see **&6FVfv**

As third example, switch to **PRO** mode and do:

```
NEW
Shift+CA
&7900 (up-arrow)
```

You see:

```
7900:sbc c
```

Enter **push HL** and when **7900:push hl** is displayed, press the **(down-arrow)**.
Enter **pop DE** and when **7002:pop de** is displayed, press the **(down-arrow)**.
Enter **ret** to terminate.

Now, switch back to **RUN** mode and do

```
REGS HL=&8899
EXEC TRON&7900
```

You see:

```
7900:push hl
```

Execute the instruction by pressing **(down-arrow)**, and **DEF+(up-arrow)** shows:

```
7BAD->88.99.()
```

This is the stack pointer and the stack values until the **()** which is the *"end-stack marker"*.
So the HL content &8899 is pushed into the stack.
Execute the next instruction by pressing **(down-arrow)**, **DEF+(up-arrow)** and now you see:

```
7BAF->()
```

The stack is empty because **DE** has been *"popped"* from the stack. **DEF+RCL** shows:

7904:*bbcc* **8899 8899** *aa ff*

Finally, just press **Shift+CA** to exit from debugger.

As last example, the source **renum.bas** will replace the old **RENUM** command. This is a simple source code to *RENUMBER* the BASIC lines inside a program. The **GOTO** and **GOSUB** are not renumbered.

Load the **renum.wav** image and change the address on the line **10** according to the image you loaded. Do **RUN** to assemble the program. In this example, we assume that the **MACBAS2014** image is for **CE-161** and that we change the line **10** as follow:

```
10 asm &2000,&20FF,&2100,&21FF
```

To execute it, just do

```
EXEC #&2000
```

Called without any argument, the first line is **10** and the increment is **10**.
Try now to call the renumber routine as follow:

```
EXEC#&2000;1000,2
```

In **PRO** mode, you see:

```
1000 asm...
1002 IF PASS=1LET N=THIS+100:R=N:E=N:S=N
1004 defs "RENUM":sbr &C8,N
     ...
```

Like a symbol **"RENUM"** is defined, you may also call **EXEC #"RENUM";100,1**

```
100 asm &2000,&20FF,&2100,&21FF
110 IF PASS=1LET N=THIS+100:R=N:E=N:S=N
120 defs "RENUM":sbr &C8,N
130 sbr &C6
140 ld h,00:ld l,10:lda l:jr R
150 N=THIS:sbr &C6
160 sbr &DE,E:sbr &D0,&3,E:push hl
170 sbr &C2,ASC",",S
180 sbr &DE,E:sbr &D0,&9,E:pop hl
190 R=THIS:push de:sta e:sbr &CC,&69
200 begin
210 "  "lda (bc):inc a
220 while nz
230 "  "lda h:sti (bc):lda l:sti (bc)
240 "  "ldi (bc):add bc
250 "  "lda e:add hl
260 repeat
270 pop de:sbr &E2
280 E=THIS:sbr &E0:S=THIS:sbr &E4
290 end
```

The code of **RENUM** (**renum.bas**) above is also an example of a source code not using any symbols. Just the BASIC variables are set to remember the jump addresses.

Note that the symbols **" "** in lines **210**, **230**, **240** and **250** are ignored by the assembler because they begin by a *space*. This is just for *"prettying"* the source code.

In line **100**, the check of the **PASS=1** is done to initialize some variables **N R E** and **S** to the value of the current assembler address (**THIS**) added with an offset **100**. The fact to initialize the variables is to avoid a displacement or a value error while assembling. Each time a jump point has to be set, an instruction *var*=**THIS** is written (lines **150** for **N**, **190** for **R**, and **280** for **E** and **S**). The variable is fixed to current assembler address and the value is kept, because the initialization of the line **110** is done only one time, when the assembler pass is **1**. The assembler is a *"two passes"* assembler; this means that the code is *"evaluated"* and **"assembled"** two times, the first with **PASS=1** and the second with **PASS=2**. So at the *"second pass loop"*, the test on the line **110** is false and the **LET** instruction is not executed. Note that **PASS=0** if the assembler is not running, i.e outside a loop **asm** .. **end**.

The two passes have the following interests:

- The first pass (**PASS=1**) evaluates the assembly code, check the syntax, increment the current assembler address at each instruction and declare the symbols. The values affected or read are *"dummy"*. But some syntax or invalid values may be rejected during this pass.
- The second pass (**PASS=2**) assembles the instructions, writes the code into memory, evaluates the symbols and raise error on undefined symbols or out-of-range expressions or symbols values.

When assembling in immediate or command mode in **PRO** mode, the assembler is considered as in pass 2 (**PASS=0**).

5/ Assembler/debugger/sourcer errors

100 The assembler is already running
101 The assembler is not running
102 Code area and symbol tables overlaps
103 No **end** found while assembler is running
104 **end** does not return to **asm**
105 The code differs between pass 1 and 2
106 Code goes outside its space (<*end-code*>)
107 Symbols goes outside the table (<*end-symb*>)
108 Structured macro stack not empty at **end**
109 Structured macro stack empty or full up
110 Incorrect structured macro stack pop
111 Condition **cc** expected
112 Incorrect condition **cc** for **sbr** or structured macros
113 Relative jump greater the 255 bytes
114 8-bits value expected, **0..255**
115 16-bits value expected, **0..65535**
116 Incorrect instruction
117 Incorrect instruction in second page **&FD**
118 **EXEC** can not return register value in **TRON** mode
119 **SRCS** overlaps
120 Unknown symbol
121 Redefined symbol
122 Invalid character in symbol name
123 Symbol value differs between pass 1 and 2

6/ MACBAS2014 BASIC programs and the lhTools

It is possible to write and build BASIC programs containing **MACBAS2014** mnemonics using the **`lhasm`** utility from the **lhTools**. In a same way, when uploading a BASIC program from **MACBAS2014**, it is possible to decode properly the mnemonics with **`lhdump`**.

The keyword file **`macbas2014.kyw`** is provided for a usage with the **lhTools** utilities.

To decode a BASIC binary from the **MACBAS2014**, just specify the keyword file with the option **`-K`** *<keword file>* to **`lhdump`**.
For example, **`lhdump -K macbas2014.kyw strupper.bin`** will show:

```
asm &4000,&40FF,&4100,&41FF
"STRLOWER"dec l
if C
begin
lda (bc)
cpa ASC ("A"):jr nc,"NUPPER"
cpa ASC ("Z")+1:jr c,"NUPPER"
or ASC ("a")-ASC ("A"):sta (bc)
"NUPPER"inc bc
until djc
endif
scf :ret
end
END
```

To build a BASIC program containing **MACBAS2014** mnemonics, just add a

```
.IMPORT: <path to macbas2014>/macbas2014.kyw
< ... BASIC code ... >
```

You can see the examples of **`strupper.bas`** and **`renum.bas`** provided.

Note that the **lhTools** version **`0.7.2`** or higher are mandatory to build **MASBAS2014** from the sources.

7/ Differences between previous and next release

Yes ! 4 releases of **MACBAS** exists. The **MACBAS 1.0** in 1990/1991 and **MACBAS 3.0** in 1999 but never really tested and really not fully mature.

7.1/ MACBAS 1.0 vs MACBAS2014

The register names are **X**,**Y**,**U**,**P**,**S** instead of **BC**,**DE**,**HL**,**PC**,**SP**. So **XH**,**XL**,**YH**,**YL**,**UH**,**UL** stands for **B**,**C**,**D**,**E**,**H**,**L**. Why ? This is way SHARP name the LH5801 registers. But, because MACBAS use z80-like mnemonics, I prefer to use also the z80-like registers.

The debugger is not embedded inside the keyboard driver, but the BASIC command: **DBGS [&mn]** which is less useful and integrated.

GETS is **DEEK** and the others changes are:

#INLINE	is **asm**
#END	is **end**
#DEFINE	is **defs**
#IF	is **if**
#ELSE	is **else**
#ENDIF	is **endif**
#BEGIN	is **begin**
#UNTIL	is **until**
#WHILE	is **while**
#REPEAT	is **repeat**
#BYTE	is **byte**
#WORD	is **word**
#TEXT	is **text**

REGS does not exists.

The assembler works only while running program. It is not possible to assemble in live.

7.2/ MACBAS 3.0 vs MACBAS2014

The register names are the same as in MACBAS 1.0, for the same reason.

RENUM, **PRGM LOCK** and **PRGM UNLOCK** no more exist under **MACBAS2014**.

The keyword **BKPT** is **bkp**. It insert a call to the debugger breakpoint entry into the code. Very useful for debugging.

Some space optimization realized, but not with a real success as I see the number of regressions. Currently work-in-progress, the patch 1 is on tests, but no so stable and mature to be diffused.

7.3/ MACBAS95 vs MACBAS2014

Except for the instructions **RENUM**, **LINK** and **UNLINK** which no more exist in **MACBAS2014**, the sources code are fully compatible. Because **MACBAS2014** search for registers and conditions in both lower-and-uppercase, it is able to assemble directly source code from **MACBAS95**.

Another exception: **MACBAS2014** ignore the symbols of only 1-character or beginning by a space.

7.4/ And why MACBAS2014 ?

Because, it is the more mature and this I prefer: I found better to use single letter for 8-bits register, and 2 letters for 16-bits, like in Z80 naming. Also, I found the debugger embedded inside the keyboard very useful and let switch from ML to BASIC.

8/ License