My Little Compiler & PreCompiler

http://gtello.pagesperso-orange.fr/mlc_e.htm

1° Purpose

1-a With a single TI 99 : MLC

Create fast assembly routines directly from Extended Basic:

- no assembly knowledge required
- no Editor/Assembler cartridge required

Just compose your routines with an easy macro language and send them to the compiler through DATA lines, and that's it!

Hardware required:

- A TI-99 with 32k Ram expansion, **F18A supported**.
- A disk system or equivalent (CF7 for example)

1-b Preparing your work on a PC: PreCompiler

If you have a PC and an easy way to transfer files to the TI 99, or eventually an emulator, then the PreCompiler is for you. It offers a high level language with some specific pseudo instructions that you don't find in MLC.

It translates this readable language into the MLC macro codes. Some features:

- High level language with comments to make your source code clear
- More loop structures than in MLC (FOR/NEXT, DO/WHILE/UNTIL...)
- Integration of the BASIC lines to manage the compilation
- Easy definition of characters as patterns
- An inline assembler to bypass the limits of MLC (no E/A cart required!)
- Support of the enhanced graphic processor F18A

2° How does it work?

2-a With a single TI 99 : MLC

First, you must load the compiler, there are two ways, either the relocatable binary or the FastLoader.

```
CALL INIT::CALL LOAD("DSK1.MLCO")
```

(Only one file required, but loading is very slow!)

Or

```
CALL INIT::CALL LOAD("DSK1.NEWFLO")::CALL LINK("NEWFLO")
```

(Files required: NEWFLO the FastLoader and MLC1/MLC2 the compiler, 12 times faster but uses temporarily a large VDP Ram bloc).

Then you have access to those routines via a CALL LINK statement:

FSIZE

Syntax: **CALL LINK("FSIZE",A,B)**

Returns in A the amount of free bytes in Low Ram expansion and in B the amount of bytes used in High memory (starting at >A000).

In the earlier versions of MLC, you had the choice to compile either in low or high memory. But as MLC was growing, it took nearly all the Low ram space. So now the compilation is directed to High Memory and the instruction HIGHMEM nn (@nn) was removed.

You must be careful because High Memory is used by the TI Extended Basic to store the current program, but this space is used from >FFFF and downwards. *See **$DEL section** for details on Memory Map.*

POP

Syntax: **CALL LINK("POP",A,B)**

Removes from memory the last routine compiled by COMPIL and returns in the variable Var the amount of free bytes in Low Ram expansion and in B the amount of bytes used in High memory (starting at >A000).

CLEAR

Syntax: **CALL LINK("CLEAR",A,B)**

After compilation, removes from memory the four MLC routines FSIZE, POP, CLEAR and COMPIL. Returns in A the free memory available in LowMem (overwriting the compiler) and in B the extra memory available (overwriting some routines seldom used at run time as the debugger). *See **Using free space at run time**.*

COMPIL

Syntax: **CALL LINK("COMPIL",IO(),S$(),C$())**

Compiles every information stored into DATA statements and turn them either into programs, sounds or characters according to the sections found.

Input:

IO(1) is the line number of the first DATA statement.

Output:

IO(1) is the error code or zero if Ok *(see **Error Codes** section)*

IO(2) is the line number of the error (if any)

IO(3) is the position of the error in the line (if any)

S$() will contain the compiled sounds (if any)

C$() wil contain the binary character definitions (if any)

In the DATA section, you must follow some rules:

- Only one string per DATA line
- Each section starts with a single character code, ex: `3000 DATA P`
- Each section ends with a null string, ex `3100 DATA ""`
- The last section starts with a null string.

CODE P : Program section

A program section looks like this:

```
DATA P
DATA PRGNAM                 (max 6 characters for the name)
DATA ...MLC codes...        (max 128 bytes per line)
DATA ...MLC codes...
DATA ""
```

The program name PRGNAM must be from 1 to 6 characters long, it is the one you will use in the CALL LINK("PRGNAM",...) statement to call your assembly routine.

You can have several program sections to create more than one program.

Example:

```
 10 CALL INIT::CALL LOAD("DSK1.MLCO")
 30 IO(1)=500::CALL LINK("COMPIL",IO(),S$(),C$())
100 A=15::B=20
110 CALL LINK("TEST",A,B)
120 PRINT A,B :: END
500 DATA P
510 DATA TEST
520 DATA G1X G2Y P1Y P2X
530 DATA ""
540 DATA ""
```

The program name will be TEST, in 520 I wrote a simple program that:

- takes parameter 1 into variable X (G1X)
- takes parameter 2 into variable Y (G2Y)
- put variable Y into parameter 1 (P1Y)
- put variable X into parameter 2 (P2X)

So, this routine exchanges the values of the two parameters passed to CALL LINK. After the call A=20 and B=15.

CODE S : Sound section

A sound section allows you to create a sound to use it in your program. It looks like this:

```
DATA S
DATA n                  (string index where to store the sound)
DATA ...sound codes...
```

```
DATA ...sound codes...
DATA ""
```

When the sound is compiled, the binary codes are stored into S$(n). You can have as many sound sections as you want, take care to provide enough space in the S$() array.

Then the S$() array can be passed to your assembly routine that will play them using the "&M" function *(see **MUSIC AND SOUND** section for more details).*

CODE C : Character section

The character section allows you to define the patterns that will be used in your program. The section looks like this:

```
DATA C
DATA ...char def...
DATA ...char def...
DATA ""
```

Every char definition must have exactly:

- 16 hexadecimal signs for a single character
- 32 hexadecimal signs for two consecutive characters
- 48 hexadecimal signs for three consecutive characters
- 64 hexadecimal signs for four consecutive characters

Every DATA line is then turned into binary codes and stored into the C$() string array in sequence *(unless you specify a fixed character number).* This means that the first definition will be stored into C$(1), the next into C$(2) and so on.

Once in binary mode, you can pass them as parameters to your compiled program and use those definitions with the `&D` or `&d` instructions *(see **&D** for more details).*

Note: *only one char section is allowed, else each would overwrite the previous definitions starting again at C$(1)!*

A char definition, if used only once, can be sent directly to VDP Ram to define a char at compilation time. So you save memory because no string is stored into C$() and time because the `&D` or `&d` instruction is not used. To do so, you must precede the definition with the char number and a point:

`Snnn` to send it to the standard zone
`Xnnn` to send it to the extended zone *(see **Pattern discussion**)*

Example:

```
DATA C
DATA 00FF00FF00FF00FF          ; sent to C$(1)
DATA S65.FF8181818181FF        ; def char 65 in standard zone
DATA X255.007E7E7E7E7E00       ; def char 255 in extended zone
DATA 0103070F1F3F7FFF          ; sent to C$(2)
DATA ""
```

CODE D : Delete previous DATA

Insert a `DATA D` between two sections to free memory at compilation time. *See* `$DEL` *in the next* ***PreCompiler*** *paragraph for details.*

2-b On a PC: PreCompiler

Just run PreCompiler.exe and you're driven to a menu where you have to:

- Select once the text editor you want to use (this will be saved)
- Select the source file, I recommend using a name with TXT extension.

Then you can edit this source file, compile it and view the compiled source code. This last has the same name but with BAS extension.

Source organization:

You just write BASIC lines as if they were typed on a real TI and you have some directives to include the lines containing:

- The loader and the compilation
- The MLC routines to be compiled
- The sounds
- The char definitions

Basically, a source looks like this:

```
100 CALL CLEAR::DIM S$(..),C$(..)

     ... the MLC directive to include the compiler
$MLC F 110 10 3000

300 ... my basic code...

310 some CALL LINK("MYPROG",params...)

$EQU
     ... optional equate list ...
$$


$MYPROG
     ...here the source code for MLC...
$$


     .. and eventually :
$SND 1
     ... def for sound one...
$$
```

```
$SND n
     ... def for sound n...
$$


     .. and eventually:
$GPU
     ... here the GPU assembly program (before $CHAR!)...
$$
$CHAR
     ... here char definitions
     ... eventually using $PATTERN/$$
$$
$END
```

You can see that $MLC directive includes the compiler management. Every other directive is a bloc $DIRECTIVE ...$$ and that the last $$ must be followed by $END to finalize the source code.

$MLC

Syntax: `$MLC mode start inc data`

Includes the BASIC lines to load MLC and run compilation.

`Mode:` can be

- **N** **for standard slow loading**
  This is just a CALL LOAD("DSK1.NEWMLCO")
- **F or 2** **to use the FastLoader**
  It uses the Fast loader CALL LOAD("DSK1.NEWFLO")
  It reads DSK1.MLC1/2 to recreate NEWMLCO in memory
  Advantage FAST! With mode 2: CALL LINK("CLEAR",A,B) is added.
- **D** **don't load the compiler**, just run compilation assuming MLC's loaded.

If mode is followed by a S (NS, FS, 2S or DS) then it is "silent" mode, you don't get any screen information regarding the loading or compilation. This can be useful if you have something like a title on screen!

`Start` is the start line where to put the loader

`Inc` is the increment between lines

`Data` is the start line of the data section that will be created with your MLC program, sounds and char definitions.

Example: `$MLC F 110 10 3000`

The loader/compiler will use the FastLoader starting at line 110 with increment 10 (110, 120, etc..) and the data section will start at line 3000 using the same increment.

You can have more than one $MLC directive, but then you must provide the same number of blocks ending with $END. The compilation will be done by blocks. This offers the

opportunity to run compiled program before another compilation and to use the POP subprogram to remove useless routines from RAM.

Example:

```
$MLC F 100 10 3000
```
… here the instructions you want…
… maybe CAL LINK to the first block of routines….
…maybe CALL LINK to POP to get back free space…

```
$MLC DS 500 10 4000       ; mode D (don't load) and silent.
```
… here the instructions you want…
… maybe CAL LINK to the second block of routines….
…maybe CALL LINK to CLEAR to get more free space…

```
$PROG1    ; this PROG1 will be compiled in data
…etc      ; starting at line 3000
$$
$END

$PROG2    ; this PROG1 will be compiled in data
…etc      ; starting at line 4000
$$
$END
```

$SND

Syntax:
```
$SND i
... sound definition...
$$
```

This includes a definition for sound i that will be stored into S$(i) string. So you must provide a DIM S$(..) fitting your needs.

Then the S$() array can be passed to your assembly routine that will play them using the "SOUND" function *(see **MUSIC AND SOUND** section for more details).*

$PRGNAM

Syntax:
```
$PRGNAM (0)
     .. program instructions...
$$
```

If followed by 0, then there is no space between instructions in the DATA lines.

Defines a bloc with a program to be compiled with MLC.

PRGNAM is the program name and can't exceed 6 characters. This is the exact name you will use in your CALL LINK("PRGNAM"...) statements!

Of course, PRGNAM can't be MLC, SND, CHAR or END as they are reserved for the other directives.

Several program blocs can be defined.

Example:

```
$PLUS1
     GETPARAM 1 A
     INC A
     PUTPARAM 1 A
$$
```

With `CALL LINK("PLUS1",N)` you'll get in return a value incremented by one.

$CHAR and $PATTERN

Syntax:

```
$CHAR
... char defs that can be definitions with
     $PATTERN
     ...
     $$
$$
```

This section allows you to prepare char definitions for your MLC program.

Each definition is similar to the CALL CHAR hexadecimal standard and can be for one, two, three or four consecutive characters. This is known by the PreCompiler with the length of the string you write:

one char if exactly 16 signs
two chars if exactly 32 signs
three chars if exactly 48 signs
four chars if exactly 64 signs

Example:

```
$CHAR
     FF818181818181FF                    ; defines a square
     00E070381F3870E000180C06FF060C18    ; defines an arrow
$$
```

The first def will be stored into C$(1) and the second into C$(2). You'll be able to use this definitions in you MLC program with the `DEFCHAR` or `XDEFCHAR` instructions *(see **Screen Instructions** for details).*

A char definition, if used only once, can be sent directly to VDP Ram to define a char at compilation time. So you save memory because no string is stored into C$() and time because the `DEFCHAR` or `XDEFCHAR` instruction is not used. To do so, you must precede the definition with the char number and a point:

`Snnn` to send it to the standard zone
`Xnnn` to send it to the extended zone *(see **Pattern discussion**)*

Example:

```
$CHAR
     S65.FF818181818181FF          ; defines a square for standard char 65
     X9.00E070381F3870E000180C06FF060C18
                                   ; defines an arrow for extended chars 9 and 10
$$
```

If you're not familiar with the hexadecimal definitions, you can use $PATTERN to define graphically a character, the previous section could have been written like this:

```
$CHAR
     $PATTERN       ; a square
          ; (or use  $PATTERN S65  to define standard char 65)
          00000000
          0......0
          0......0
          0......0
          0......0
          0......0
          0......0
          0......0
          00000000
     $$

     $PATTERN       ; an arrow
          ; (or use  $PATTERN X9  to define extended chars 9 and 10)
          ................
          000........00...
          .000........00..
          ..000........00.
          ...0000000000000
          ..000........00.
          .000........00..
          000........00...
     $$
$$
```

For a pattern, use a period "." for a white pixel and any other character for a black pixel. I use "0", but a "*" can be ok too.

Note: only one $CHAR bloc is allowed!

$END

Syntax: `$END`

This directive must follow the last $$ of the last bloc to finalize the source code.

$GPU *F18a only*

Syntax:
```
$GPU
... GPU assembly program ...
$$
```

For those owning a F18A graphic processor, you can define an assembly program that will be executed by the GPU (the processor located into the F18A) in parallel with your CPU.

Only assembler can be used in this section!

The F18A extends the VDP Ram with 2 extra KB from >4000 to >47FF, that's where MLC will copy this program. The GPU is a clone of the TMS9900 with those differences:

- It can only access VDP RAM from >0000 to >47FF
- The registers R0-R15 are fixed and not memory mapped
- You can't access the ROM routines from it

With the PreCompiler, this section is turned into a "CHAR" section whose definitions correspond to the binary codes of the program *(it was a trick to reuse some existing routines).* Those defs are sent to the VDP with the Xnnn directive at compilation time in the extended character bloc starting at >1000.

Then, the GPU itself is called to move this program to >4000, the final location.

Note: Knowing this, **the $GPU section must precede every $CHAR section** where the Xnnn directive is used, else characters will be overwritten by the GPU routine.

See *"F18A support"* section for more informations.

$EQU

Syntax:
```
$EQU
     \substitutename          original
     …
$$
```

Example:
```
$EQU
     \counter  N
     \repeat   100
     \adr      &H8300
$$
```

Then in your program

`LET \counter 5`	will be	`LET N 5`
`NDO I \repeat`	will be	`NDO I 100`
`ADD A \adr`	will be	`ADD A &H8300`

The $EQU section must appear before any \substitute name is used !

There are **predefined values to access the PAD ram**. This is the fast ram of 256 bytes attached to the 16bits bus of the processor. It is located at addresses **&H8300** to **&H83FF**.

In the "original" value of an equate, you can write **PADxx** (xx being an hexadecimal value from 00 to FF).

Example:

```
$EQU
     \var1          PAD04
     \var2          PAD0E
$$
```

Will give var1 the value of &H8304 and var2 the value of &H830E.

Typically, if you're using MLC and that XBasic is still running, only PAD00 to PAD17 are available. Further, if you exchange parameters with the BASIC through your CALL LINK statement, only PAD00 to PAD0F are available.

But, if you just use one CALL LINK and never return to the XBASIC, then PAD00 to PAD49 are free.

For word access, be sure to use only even values! PAD00, PAD02, PAD04, etc.

Important note: when using such an equate, whenever an instruction expects a parameter that can be a variable or a numeric value, those PADxx will be seen as constants and not addresses of variables !

Example:

```
ADD \var1 \var2
```

will add the constant &H830E into the memory location &H8304.

Because the first parameter is always a variable, but the second can be either a variable or a constant.

If you want to force a numeric value to be seen as an address, it must be followed by a "^" character:

```
ADD \var1 \var2^
```

will add the content of memory location &H830E to memory location &H8304.

If you are not working with the PreCompiler, but directly with MLC and XBASIC, the same applies :

```
(&H8304 = -31996 and &H830E=-31986)

+-31996.-31986      for  ADD \var1 \var2
+-31996.-31986^     for  ADD \var1 \var2^
```

$DEL

Syntax: `$DEL`

This directive can only appear between two sections (*i.e. after a* `$$`) and allows you to remove from memory all the previous DATA lines that have already been parsed by the compiler to free memory. You must understand how MLC manages the memory:

High Memory Map with MLC

>A000	32	**My Workspace**, Registers R0-R15
>A020	52	**Variables** A-Z
>A054	6	**Pseudo variables** `[, \, ]` for debugging
>A05A	??	Essential routines for MLC
>Axxx		Here starts your own programs

… Free Space …

(>8386)	**last free address in memory**, here the numeric variables
(>8330)	**table for BASIC lines** (each entry: WORD line number, WORD line address) *Lines appear in reverse order (the last line first)*
(>8332)	**last byte of the line table**
	Tokenized BASIC lines *Lines appear in reverse order that they were entered* *(not the number line order!)*
>FFFF	**End of High Memory**

The **Extended Basic** uses the High Memory from the end (>FFFF) and backwards. That's why the **MLC** uses this memory from the start (>A000) and forwards. The two systems can conflict if one overwrites the other.

How to estimate the free space?

The **command SIZE** gives you the amount of free memory for program in Extended Basic, this is the size from `>A000` to the GPL Substack, lets say: *21500 bytes*.

Then **CALL LINK("FSIZE",A,B)** returns in B the amount of memory used by MLC in High Memory, this is the size from `>A000` to the end of your compiled programs, let's say *9200 bytes*.

Then the available memory is: *21500 – 9200 = 12300 bytes*.

Standard use of `$DEL`

- If you think **you are short in memory**, then using `$DEL` can save you! The idea is that:

1) First define the characters, they are sent to VDP Ram (*in the pattern zone or in* `C$()`)
2) Define your sounds, they are sent to VDP Ram (*in* `S$()`)
3) Use `$DEL` to remove DATA lines concerning chars and sounds
4) With the extra space you just gained, your compiled program has more space!

- To be **even more efficient** if you still need space, you can separate your program in two blocs:

1) First bloc, define all your tables and everything that will be done once (for example the color definitions, etc), let's say you name this `$TABLE`
2) Use `$DEL` to remove DATA lines concerning this first part
3) Then define your actual program with more space, let's say you name it `$MYPRG`

   At run time, `CALL LINK("TABLE")::CALL LINK("MYPRG").`

- **Important note:** after compiling remember that your program has changed, DATA lines have been removed! So save it BEFORE running it, and not after…!

- **Not less important note:** `$DEL` can only work if lines were entered in order. Else, memory will be in a total mess. If you work directly on the TI-99 and modify one line, then this one will become the first line in memory. To get back to the normal order, use SAVE AS and MERGE. If you use an emulator, like Classic99, prepare your source in an editor and "PASTE" it.

- **More than important note:** After compilation, if you use `$DEL`, ***all numeric variables are lost!*** Their address has changed. So, you must make the compilation very soon in your program, then fill in the numeric variables. Don't expect to find them equal to zero, you MUST initialize them after compilation.

Using free space at run time

While your program is running, you can use this little assembly routine to get the address and size of the free space in High Memory using the assembly reference `HIGHPT` that is the internal pointer of MLC to its first free address in **High Memory**:

```
$[
      mov @&H8386,r0 ; last free byte for Extended Basic
      mov @highpt,r1 ; first free byte for MLC
      s r1,r0        ; r0 = r0 – r1
      inc r0         ; one byte was missing, free size
      mov r1,@A      ; address in A
      mov r0,@B      ; size in B
$]
```

This returns in A the starting address of the free space and in B the size in bytes you can use without overwriting the Basic space.

Low Memory MAP with MLC and Extended Basic

>2000	**start** of low memory
>2002	**FFADDR** pointer to First free address
>2004	**LFADDR** pointer to Last free address
	…. (here a zone managed by the system) …
>24F4	Start of **MLC run time routines**
	…. (everything required to run, as keyboard, joystick, graphic routines)…
>CLRMORE	start of **non essential routines**
	…. (such as the debugger, seldom used) …
>CLRMLC-2	here a pointer to CLRMORE
>CLRMLC	start of **MLC compiler**
	… (Here the COMPIL, FSIZE, POP and CLEAR subprograms)…
>FFADDR-256	… **labels table** : 256 bytes that can be used by TR0 … *(see TR DIMTABLE)*
>FFADDR	here starts the **standard free space**
	… (so little…) ….
>LFADDR	start of **program name table**
…	
>4000	**end** of low memory

The same for the **Low Expansion ram**. Even if MLC is stored there, there may be some bytes free (not guaranteed, this changes at every version):

```
$[
      mov @LFADDR,r0 ; start of REF/DEF table
      mov @FFADDR,r1 ; end of the compiler zone
      s r1,r0        ; free space between the two
      mov r1,@A      ; address in A
      mov r0,@B      ; size in B
$]

A = FFADDR and B = LFADDR-FFADDR.
```

The Labels Table (with MLC):

A 256 bytes zone is available at run time to the user through two instructions: **TR** for table reservation and **'@** for floating point array. The **&D** and **&d** instructions use the last 33 bytes for string transfer!

So, if you used TRA0 for word array, then A(0) to A(110) are available but A(111) to A(127) are lost.

If you used TRE0 for byte array, then E(0) to E(222) are available but E(223) to E(255) are lost.

If you used '@0 for floating point array, then R2 to R28 are available, but R29 to R33 are lost.

The Labels table (with thePreCompiler):

A 256 bytes zone is available at run time to the user through two instructions: **DIMTABLE** for table reservation and **FARRAY** for floating point array. The **DEFCHAR** and **XDEFCHAR** instructions use the last 33 bytes for string transfer!

So, if you used DIMTABLE A 0 for word array, then A(0) to A(110) are available but A(111) to A(127) are lost.

If you used DIMTABLE E 0 for byte array, then E(0) to E(222) are available but E(223) to E(255) are lost.

If you used FARRAY 0 for floating point array, then R2 to R28 are available, but R29 to R33 are lost.

If this is not enough for your needs : CALL LINK("CLEAR"…) :

When you use **CALL LINK("CLEAR",F,FF),** then you get much **more space** in LOW MEM as some routines, only useful once when compiling, are removed. So, the compiler and FSIZE, POP and CLEAR are deleted !

Note: *the new block includes the **Labels Table**, they are not two separate zones.*

In this case the previous routine returns:

```
Address A = CLRMLC and size B = LFADDR - CLRMLC
(here B is the same value as F returned by CLEAR)
```

But, you can get **more space** if you don't use the debugger. In this case, you can consider that the free space starts at CLRMORE, use this routine instead :

```
$[
     mov @LFADDR,r0 ; start of REF/DEF table pointing to CLRMLC
     mov @FFADDR,r1 ; end of the compiler zone
     dect r1        ; pointer to more space
     mov *r1,r1     ; and get the start address, points to CLRMORE
     s r1,r0        ; free space between the two
     mov r1,@A      ; address in A
     mov r0,@B      ; size in B
$]

Address A = CLRMORE and size B = LFADDR - CLRMORE
(here B is the same value as F+FF returned by CLEAR)
```

3° My Little Language

In the following section, each instruction will be described both for MLC and for the PreCompiler. For example:

G	***GETPARAM***	get a parameter from the Call Link statement	**Z**

Tells you that "G" is the MLC macro-code, that" GETPARAM" is the equivalent in the PreCompiler. Then you have a short description.

If a "**Z**" is present, then, the result of this instruction is compared to zero, so you can use jump instructions according to this.

3-a With a single TI 99 : MLC

This language is a serie of codes (one letter or two) followed by zero to three arguments.

Arguments can be:

- one of the 26 predefined variables named A to Z, they can contain an integer from -32768 to +32767
- a numeric constant from -32768 to 32767. If an instruction has two constant arguments, the "." can be used as a separator if needed.
- a single character that represents its ASCII value, use $ followed by the desired character. For example $3 represents the ASCII value 51. Single characters are treated the same as constants.

Instructions can be separated by one or more spaces (for readability) but can be stuck together (for compacity). But, a code can't be separated from its arguments.

Conventions:

k a numeric constant
v a variable
n number, either variable or constant
c a character

Persistence of a variable:

Calling "COMPIL" erases all variables from A to Z.
Else, if you only call your own routines, their values are kept from one call to another.
All of your routines share the same variable space.

3-b On a PC: PreCompiler

The language is a serie of instructions followed by zero to three arguments, most of them are synonyms of MLC macro codes, but others, referenced as pseudo-instructions, are constructions using several MLC codes.

Only one instruction is allowed per line.

One line can be followed by a comment starting with ";". Any comment is cut at compilation time, they are only here for your convenience.

Arguments can be:

- one of the 26 **predefined variables** named A to Z, they can contain an integer from -32768 to +32767
- a numeric decimal constant from -32768 to 32767.
- a numeric **hexadecimal** constant with **&h prefix** (from &h000 to &hFFF)
- a numeric **binary** constant with **&b prefix** (from &b0 to &b1111111111111111)
- a single **character** with **$ prefix** (ex: $A is equivalent to 65)
- a **speech word** address with **&w prefix** (ex &wHELLO)

Arguments must be separated by either a space, a comma or parenthesis.

Example:

```
PUTCHAR 10 7 65          ; write a "A" at position 10,7
```

For readability you can write:

```
PUTCHAR (10,7),$A
```

To show that 10 and 7 are a set of coordinates. You choose the appearance you prefer!

```
AND E 255        ; remove upper byte
```

For readability on binary operations you can write:

```
AND E &h00FF
```

Variable initialization

=	***LET***	store a value into a variable or memory location	**Z**

Syntax: =nn'
Make n=n'. If n is a constant, then it is considered as a memory address.

Ex: =A-10 A=-10
=DF D=F
=8192A poke word A into 8192
=8192.65535 poke >FFFF into 8192

- **With the PreCompiler**, syntax: ***LET n n'***

Ex:
```
LET A -10
LET D F
LET 8192 A
LET 9182 &hFFFF
```

X	***SWAP***	exchange two variables or memory locations

Syntax: Xnn'
Exchange values of n and n'. If one parameter is a constant, it is considered as a memory address. This is the only way to "peek" values from memory.

Ex: XPS P=S and S=old value of P
XA8192 exchange words in variable A and location 8192.
X8192.8194 exchange two memory words at 8192 and 8194.

- **With the PreCompiler**, syntax: ***SWAP n n'***

Ex:
```
SWAP P S
SWAP A 8192
SWAP 8192 8194
```

G	***GETPARAM***	get a parameter from the Call Link statement	**Z**

Syntax: Gnn'
Get parameter number n and store it into n' variable or memory location.
Note: *if the parameter is an array, you must use TG (see **Arrays** section).*

Ex: assuming CALL LINK("TEST",D,E,F)
G1A A = integer value taken from D in basic (1st parameter)
GZA Take from Z the parameter number (must be 1, 2 or 3 for D, E and F) and according to its value, stores in A the corresponding parameter.
G1.12000 poke integer value of parameter 1 into memory location 12000.

- **With the PreCompiler**, syntax: ***GETPARAM n n'***

Ex:
```
GETPARAM 1 A
GETPARAM Z A
GETPARAM 1 12000
```

P ***PUTPARAM*** put a value into a parameter in the Call Link statement

Syntax: Pnn'

Stores a value into one of the variables in the Call Link statement. If n' is a numeric constant, it is not considered as a memory location but as a value. So you can't send directly a memory word to a Basic parameter. you have to use X (exchange) before.

Note: *if the parameter is an array, you must use TP (see **Arrays** section).*

Ex: assuming CALL LINK("TEST",D,E,F)

P3Z give to F (3rd parameter) the integer value of Z

P1.5 give to E (1st parameter) the value 5.

PZ-8 take from Z the parameter number (must be 1, 2 or 3) and according to its value, stores -8 in the corresponding parameter.

- **With the PreCompiler**, syntax: ***PUTPARAM n n'***

Ex:
```
PUTPARAM 3 Z
PUTPARAM 1 5
PUTPARAM Z -8
```

Arrays

In “My little Compiler”, arrays can be of five different types:

- located un CPU RAM with word elements (-32768 to 32767)
- located in CPU RAM with byte elements (0 to 255)
- located in VDP RAM with word elements
- located in VDP RAM with byte elements
- linked to an array passed as an argument in CALL LINK (words)
- linked to a string (array) passed as an argument in CALL LINK.

No need to declare an array, it is a common variable whose name gives the type and whose value gives the starting address according to the following table:

Possible Name	Type	Value of variable
A, B, C, D	word array in CPU RAM	even address of first element
E, F, G, H	byte array in CPU RAM	address of first element
I, J, K*, L	word array in VDP RAM	address of first element
M, N, O, P	byte array in VDP RAM	address of first element
Q, R, S, T	word array in CALL LINK	parameter number in CALL LINK
U, V, W ,X	string (array) in CALL LINK	parameter number in CALL LINK

(*) *K is used by Joystick and Keyboard calls to return the key code, don’t use it as an array name if your routine reads the keyboard or the jostick status.*

Array (from A to P) elements start at ZERO, so an array with 20 elements contains element 0 to element 19. Arrays linked to BASIC (from Q to X) start at ZERO or ONE according to OPTION BASE.

Note : If you want to perform arithmetic instructions, comparisons or anything else with an array element, you must first load it into a variable (with TG) , and then eventually put it back in the array (with TP).

T prefix for table/array instructions in MLC

This character starts every table/array instruction.

TG ***GETTABLE*** get a value from an array **Z**

Syntax: TGtiv

Make v = t(i) ***except for string (arrays) in CALL LINK(*)***

t must be an array name, v must be a variable and I can be constant or variable.

Ex: TGA2E make E = A(2), a word value in CPU RAM located at value of A plus 4 (two bytes per element). If A = >A000, then E = word from >A004

TGQ5F make F=Q(5), a word value from an array in CALL LINK statement. Assuming CALL LINK(“TEST”,A(),B(),C()) and Q=2, so Q(5) refer to B(5) as B() is the 2nd table in the list.

TGNAB make B=N(A) a byte value in VDP RAM. If A = 7 and N = >800, then B = byte from address >807 in VDP RAM.

- **With the PreCompiler**, syntax: ***GETTABLE t(i) v***

Ex:
```
GETTABLE A(2) E
GETTABLE Q(5) F
GETTABLE N(A) B
```

(*) ***String arrays in CALL LINK:*** the value returned is not the string itself, but the VDP Ram address of the string and Z contains the length of the string.

Comparison to zero is performed on the value of Z.

Note: *if the string is a simple variable, then use i=0.*

Example: CALL LINK("TEST",A$,B$())

=U2	to access second array B$()
TGU3M	makes M=VDP RAM address of B$(3) and Z=len(B$(3))

Using M is interesting as it can be the name of a byte array in VDP RAM, what a string is!

=X1	to access first variable A$
TGX0N	makes N=VDP RAM address of A$ and Z=len(A$).
?=a	and jumps to label a if len is zero.

- **With the PreCompiler**,

Ex:
```
LET U 2
GETTABLE U(3) M

LET X 1
GETTABLE X(0) N
IF=THEN a
```

TP ***PUTTABLE*** put a value into an array element

Syntax: TPtin

Make t(i) = n ***except for string (arrays) in CALL LINK(*)***

t must be an array name, i and n variables or numeric constants.

Ex: TPS3.-1 make S(3) = -1, S refer to an array in the CALL LINK statement. Assuming CALL LINK("TEST",A(),B(),C()) and S=1, then S(3) refer to A(3) as A() is the 1st array in the list.

TPFAB make F(A) = B a byte value in CPU RAM. If F = 8192, A = 100 and B = 214, put byte 214 into CPU location 8192+100 = 8292.

- **With the PreCompiler**, syntax: ***PUTTABLE t(i) n***

Ex:
```
PUTTABLE S(3) -1
PUTTABLE F(A) B
```

(*) ***String (arrays) in CALL LINK:*** the value n must be a variable containing the CPU address of the string *(typically a BYTE table name such as E, F, G or H)* and variable Z must contain the length of the string.

If you use i=0 then the parameter in CALL LINK can be a simple string variable.

Ex: With CALL LINK("TEST",A$(),S$)

TRF50	F a table with 50 bytes maxi
...	
=Z10	
TP1.4F	send a 10 bytes string from table F to A$(4)
TP2.0.F	send a 10 bytes string from table F to S$.

- **With the PreCompiler**

Ex:
```
DIMTABLE F 50
...
LET Z 10
PUTTABLE 1(4) F
PUTTABLE 2(0) F
```

TF ***FILLTABLE*** fills a table with a value

Syntax: TFtnv

t must be a table either in CPU or VDP RAM (from A to P). "n" is the number of elements to fill with value "v".

Ex:	TFA10.5000	10 first words of A are filled with value 5000 in CPU ram.
	TFM12.0	12 first bytes of M are filled with 0 in VDP ram.

- **With the PreCompiler**, syntax: ***FILLTABLE t n v***

Ex:
```
FILLTABLE A 10 5000
FILLTABLE M 12 0
```

TR ***DIMTABLE*** reserve a table in CPU RAM

Syntax: TRts

t must be a table name in CPU RAM (from A to H), at compilation time this instruction reserves s words (name from A to D) or s bytes (name from E to H). At run time it gives to the "t" variable the address of the first element of the array.

"s" (size) must be a constant as it has to be know at compilation time!

Ex:	TRD32	reserve 32 words (64 bytes) and give D the first address.
	TRE32	reserve 32 bytes and give D the first address.

- **With the PreCompiler**, syntax: ***DIMTABLE t s***

Ex:
```
DIMTABLE D 32
DIMTABLE E 32
```

Special value: if you use a size of ZERO, the variable is given the value of a 256 bytes zone used by the compiler for label references (see **Using free space at run time**)

Ex:	TRA0	A points to a space with 128 words.
	TRE0	E points to a space with 256 bytes.

- **With the PreCompiler**

Ex:
```
DIMTABLE A 0
DIMTABLE E 0
```

You must know that if you use again “COMPIL” the values in this array will be lost.

TM ***BMOVE*** move bloc of bytes in CPU RAM

Syntax: TMsnd
Copy from source address s to dest address d a bloc of n bytes in CPU RAM.

Ex: TRE50 reserve 50 bytes and address in E
TRF50 idem for F
TME50F copy table E to table F

- **With the PreCompiler**, syntax: ***BMOVE s n d***

Ex:
```
DIMTABLE E 50
DIMTABLE F 50
BMOVE E 50 F
```

TW ***WMOVE*** move bloc of words in CPU RAM

Syntax: TWsnd
Copy from source address s to dest address d a bloc of n words in CPU RAM.

Ex: TRA20 reserve 20 words and address in A
TRB20 idem for B
TWB20A copy table B to table A

- **With the PreCompiler**, syntax: ***WMOVE s n d***

Ex:
```
DIMTABLE A 20
DIMTABLE B 20
WMOVE B 20 A
```

TC ***BMOVECTOV*** move bloc of bytes from CPU RAM to VDP RAM

Syntax: TCsnd
Copy from source address s in CPU RAM to dest address d in VDP RAM a bloc of n bytes.

Ex: TRE32 reserve 32 bytes and address in E
TWE32.0 copy table E to table VDP address 0, so overwrites first screen line !

- **With the PreCompiler**, syntax: ***BMOVECTOV s n d***

Ex:
```
DIMTABLE E 32
BMOVECTOV E 32 0
```

TV	***BMOVEVTOC***	move bloc of bytes from VDP RAM to CPU RAM

Syntax: TCsnd
Copy from source address s in VDP RAM to dest address d in CPU RAM a bloc of n bytes.

Ex: TRE768 reserve 768 bytes and address in E
TW0.768E copy 768 bytes from VDP address 0 into E, so, save the whole screen into table E.

- **With the PreCompiler**, syntax: ***BMOVEVTOC s n d***

Ex:
```
DIMTABLE E 768
BMOVEVTOC 0 768 E
```

TO	***BYTE***	write One byte at the current compilation address

Syntax: TOn
N must be a constant (0-255 or -128-+127), this is used to initialize tables of bytes. *See below*.

- **With the PreCompiler**, syntax: ***BYTE n***
  If you have more than one byte to initialize, then use ***BYTES n1,n2,...***

TT	***WORD***	write Two bytes at the current compilation address

Syntax: TTn
N must be a constant (0-65536 or -32768-32767), this is used to initialize tables of words. *See below*.

- **With the PreCompiler**, syntax: ***WORD n***
  If you have more than one word to initialize, then use ***WORDS n1,n2,...***

TH	***HEX***	write hexadecimal bytes at the current compilation address

Syntax: THn.hhhh...
n must be a constant (0-255) and following must be a "." and n bytes using each two hexadecimal digits. Ex: `TH6.AA12FF06C31E`

- **With the PreCompiler**, syntax: HEX ***hhhh...***
  Note that you don't need to specify the length with the PreCompiler.

TS	***STRING***	write a string at the current compilation address

Syntax: TSn.XXX...

n must be a constant (0-255) and following must be a "." and n characters. The string can't be cut into two different compilation strings. If it's too long, just use several TS statements.

This is used to initialize a string constant. *See below*.

- **With the PreCompiler**, syntax: ***STRING "XXX..."***

Note that you don't need to specify the string length with the PreCompiler, it does this for you, but you must enclose the string with "..".

TJ	***SKIP***	jump over a block and update the compilation address

Syntax: TJn

The compilation pointer is incremented by n.

- **With the PreCompiler**, syntax: ***SKIP n***

Note: *you may have to use EVEN if n is odd.*

TE	***EVEN***	ensure that the current compilation address is even

Syntax: TE

Turns to even the current pointer of compilation if it was odd after TS or TO statement. This is absolutely necessary because assembly instructions must start on an even address. *See below*.

- **With the PreCompiler**, syntax: ***EVEN***

T<	***STOREPTR***	store the current compilation address

Syntax: T<

Store the current compilation address into a temporary space. This is mostly used before a table or string initialization to get the starting address. *See below*.

- **With the PreCompiler**, syntax: ***STOREPTR***

T>	***RECALLPTR***	recall the current compilation address

Syntax: T>V

Give to variable V the value stored in the temporary space with "T<". So now V points to the address you've saved. *See below*.

- **With the PreCompiler**, syntax: ***RECALLPTR V***

Note on Data initialization:

a- With MLC

Typically, you have to do this:

```
B0          Jump over the table
T<          To save the starting address
```
...here use TO, TT, TH, TJ or TS to put your data...
```
TE          eventually if you used TO, TJ or TS
L0          the label you've jumped to
T>V         get the address back into your variable.
```

Then your variable will point at the starting address of data.

Some example here:

Initialize a table of 5 WORDS (the variable will be A to point to a WORD table):

```
B0 T< TT85 TT62 TT42 TT13 TT14 L0 T>A
```

Then A(0)=85,… and A(4)=14.

Initialise a table of 7 BYTES (the variable will be E to point to a BYTE table) :

```
B0 T< TO0 T01 T04 TO9 TO16 TO25 TO36 TE L0 T>E
```

Then E(0)=0, … E(6)=36 the square table!
Note the use of TE because the number of bytes is odd.

Initialize a string (the variable will be E to point to a BYTE table):

```
B0 T< TS10ABCDEFGHIJ TS10KLMNOPQRST TS6UVWXYZ TE L0 T>E
```

Then E(0)='A',…E(25)='Z'
Note that the whole string can be passed at the time with
`TS26ABCDEFGHIJKLMNOPQRSTUVWXYZ`

That was just to show the string cut..

Note: if you don't use the BRANCH instruction before your table, then the assembler will interpret your data as instruction codes and will try to execute them.
This can lead to weird results.

b- With the PreCompiler

Typically, you have to do this:

```
STARTDATA
```
…here use BYTE(S), WORD(S), HEX, SKIP or STRING…
```
ENDDATA V
```

Then your variable V will point to the start of your table.

Some example here:

Initialize a table of 5 WORDS (the variable will be A to point to a WORD table):

```
STARTDATA
     WORDS 85,62,42,13,14
ENDDATA
```

Then A(0)=85,… and A(4)=14.

Initialise a table of 7 BYTES (the variable will be E to point to a BYTE table) :

```
STARTDATA
     BYTES 1,4,9,16,25,36
ENDDATA E
```

Then E(0)=0, … E(6)=36 the square table!

Note that the number of bytes is odd, ENDDATA manages this internally to keep an even address for the next instructions.

Initialize a string (the variable will be E to point to a BYTE table):

```
STARTDATA
     STRING "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
ENDDATA E
```

Then E(0)='A',…E(25)='Z'

STARTDATA

Syntax: ***STARTDATA***

Marks the beginning of a data block. It is a pseudo-instruction, combination of MLC codes with B (Goto) and T< (StorePtr) and has no direct equivalent in MLC.

ENDDATA

Syntax: ***ENDDATA v***

Marks the end of a data block and v is equal to the starting address of the block.

This is a pseudo-instruction, combination of MLC codes with L (Label), TE (Even) and T> (RecallPtr) and has no direct equivalent in MLC.

How to know the size of a data block?

In MLC, the word just before a table is the address of the end of the table. With a simple operation you can compute the table length, example:

```
STARTDATA
     BYTES 1,2,3,4,5      ; 5 bytes
     STRING "YES"         ; 3 bytes
     WORDS -1,0,1         ; 6 bytes in 3 words
ENDDATA A                 ; total of 14 bytes
GETTABLE A(-1) B          ; take the word before the table B=end address
SUB A B                   ; B=end-start, so B is the length of the table, 14.
```

That was easy because the block was pointed to by A, a word array pointer. What happens if the bloc is pointed to by E, a byte array pointer?

```
STARTDATA
…..
ENDDATA E
LET A E                    ; first, put E into a word array pointer as A
GETTABLE A(-1) B           ; take the word before the table B=end address
SUB A B                    ; B=end-start, so B is the length of the table.
```

Arithmetic Instructions

D ***DEC*** decrement **Z**

Syntax: Dn
Decrement the variable or memory location by one.

Ex: DC C=C-1
D16000 memory location 16000-16001 is decremented by one

- **With the PreCompiler**, syntax: ***DEC n***

Ex:
```
DEC C
DEC 16000
```

I ***INC*** increment **Z**

Syntax: In
Increment the variable or memory location by one.

Ex: IC C=C+1
I16000 memory location 16000-16001 is incremented by one

- **With the PreCompiler**, syntax: ***INC n***

Ex:
```
INC C
INC 16000
```

A ***ABS*** absolute **Z**

Syntax: An
Replaces the argument by its absolute value.

Ex: AF F=abs(F)
A24000 memory location 24000-24001 replaced by its absolute value

- **With the PreCompiler**, syntax: ***ABS n***

Ex:
```
ABS F
ABS 24000
```

N ***NEG*** negate **Z**

Syntax: Nn
Replaces the argument by its opposite value.

Ex: NF F= -F
N24000 memory location 24000-24001 replaced by its opposite value

- **With the PreCompiler**, syntax: ***NEG n***

Ex:
```
NEG F
NEG 24000
```

Z	***CLEAR***	clear a variable

Syntax: Zn
Clear the variable giving it a null value.

Ex: ZA make A = 0
Z8200 make bytes at locations 8200-8201 equal to zero.

- **With the PreCompiler**, syntax: ***CLEAR n***

Ex:
```
CLEAR A
CLEAR 8200
```

+	***ADD***	sum into a variable	**Z**

Syntax: +vn
Sums n to v. If V is a constant, then it's a memory location.

Ex: +AB A=A+B
+A92 A=A+92
+15000C add C into memory location 15000-15001

- **With the PreCompiler**, syntax: ***ADD v n***

Ex:
```
ADD A B
ADD A 92
ADD 15000 C
```

-	***SUB***	substract to a variable	**Z**

Syntax: -vn
Substact n to v. If V is a constant, then it's a memory location.

Ex: -AB A=A-B
-A92 A=A-92
+15000C substact C into memory location 15000-15001

- **With the PreCompiler**, syntax: ***SUB v n***

Ex:
```
SUB A B
SUB A 92
SUB 15000 C
```

/	***DIV***	divide a variable	**Z***

Syntax: /vn

Divide v by n and return the quotient in v and the reminder always in variable Z. Both values must be positive!

Ex: assuming that A=255 and B=7
/A10 divide A by 10, so A=25 and reminder Z=5 (255=25*10+5)
/AB divide A by B, so A=36 and reminder Z=3 (255=36*7+3)

- **With the PreCompiler**, syntax: ***DIV v n***

Ex:
```
DIV A 10
DIV A B
```

Z*: note that the comparison to ZERO is performed on the reminder, not the quotient!

*	***MUL***	multiply a variable	**Z**

Syntax: *vn

Multiply v by n (v=v*n), both values must be positive!

Ex: assuming that C = 10 and T= 7
*CT C = C * T = 70
*TT T = T ² = 49
*C22 C = C * 22 = 220.

- **With the PreCompiler**, syntax: ***MUL v n***

Ex:
```
MUL C T
MUL T T
MUL C 22
```

R	***RND***	random number

Syntax: R

Return a pseudo random number in Z (from 0 to 4095 or >0FFF).

If you want to get n different values, compute the reminder of Z : n (remember that Z contains the remainder of every division!).

Ex: R /Z4 return in Z random values from 0 to 3.
R /Z26 +Z$A return in Z 26 different random values from 65 (code for “A”) to 90 (code for “Z”).

- **With the PreCompiler**, syntax: ***RND***

Ex:
```
RND
DIV Z 4

RND
DIV Z 26
ADD Z $A
```

W	***EXTEND***	extend sign from byte to word	**Z**

Syntax: Wv
Extend the sign of the low byte in v to a full signed word.
Ex: =A127 WA make A=127, no change
=A255 WA make A= -1, because 255 = >FF = byte -1.

This is useful when working with byte arrays if you want to work on signed integers.

- **With the PreCompiler**, syntax: ***EXTEND***

Ex:
```
LET A 127
EXTEND A

LET A 255
EXTEND A
```

Floating Point instructions

A floating point support is provided. I have chosen to work as close as the assembler does for two reasons:

- The compiler will use less memory
- You get the best speed as no "hidden" instruction or data transfer is used.

Two float registers are always available (r0 and r1) and they are used (so modified!) by nearly every floating point instruction.

If you need to save some values, just use '@n instruction to reserve more registers.

'@ ***FARRAY*** reserve float array

Syntax: '@n

Reserves space for n float registers. So, as r0 and r1 are always present, you have now access to r0, r1, r2, ..., r(n+1). Each one uses 8 bytes of memory.

If you compile several programs that need a set of floating point registers, you just have to allocate it once in the first program. Then, they all will share the same space.

Note: if n=0, it means that you want to use the 256 bytes memory space used for the compiler labels (that is 32 registers). So you have access to r0,...,r33.

Ex:
'@10 : reserves 80 bytes for 10 registers r2 to r11
'@0 ; reserves 256 bytes for 16 registers r2 to r33.

- **With the PreCompiler**, syntax: ***FARRAY n***

Ex:
```
FARRAY 10
FARRAY 0
```

'G ***GETFLOAT*** get float from BASIC

Syntax: 'Gnm

Get a value from the CALL LINK list and stores it into r0.
n is the order of the value in the argument list.
m is 0 for a simple value or variable, or the index for an array.

Ex: with CALL LINK("TEST",R,FL(),-9.487)
'G1.0 then r0 = R
'G2.5 then r0 = FL(5)
'G3.0 then r0 = -9.487

- **With the PreCompiler**, syntax: ***GETFLOAT n(m)***

Ex:
```
GETFLOAT 1(0)
GETFLOAT 2(5)
GETFLOAT 3(0)
```

‘P ***PUTFLOAT*** put float into BASIC variable

Syntax: ‘Pnm
Give to a variable in the CALL LINK list the value taken from r0.
n is the order of the variable in the argument list.
m is 0 for a simple variable or the index for an array.

Ex: with CALL LINK(“TEST”,R,FL(),-9.487)
‘P1.0 R = r0
‘P2.5 FL(5) = r0
‘P3.0 error because -9.487 is not a variable!

- **With the PreCompiler**, syntax: ***PUTFLOAT n(m)***

Ex:
```
PUTFLOAT 1(0)
PUTFLOAT 2(5)
PUTFLOAT 3(0)
```

‘M ***FMOVE*** move float registers

Syntax: ‘Msd
Move value of register s to register d.

Ex: ‘M0.3 then r3 = r0
‘M1.X copies r1 into register pointed by X.

- **With the PreCompiler**, syntax: ***FMOVE s d***

Ex:
```
FMOVE 0 3
FMOVE 1 X
```

‘+ ‘- ‘* ‘/ ‘^ arithmetic
FADD FSUB FMUL FDIV FX^Y

Those five instructions work the same. They use r1 and r0 and return the result in r0. Note that the value of r1 is lost too!

‘+ : r0 = r1 + r0 *(r1 is the lower absolute value of the two)*
‘- : r0 = r1 - r0 *(r1 is the lower absolute value of the two)*
‘* : r0 = r1 * r0 *(r1 = abs(r1))*
‘/ : r0 = r1 / r0 *(r1=0)*
‘^ : r0 = r1 ^ r0 *(computed as exp(r0 * log(r1)))*

- **With the PreCompiler**, syntax: ***the name without parameter.***

Ex:
```
FADD
FSUB
FMUL
FDIV
FX^Y
```

‘C	‘S	‘T	‘A	‘Q	‘E	‘L	‘I	‘N	‘B	scientific functions
FCOS	***FSIN***	***FTAN***	***FATN***	***FSQR***	***FEXP***	***FLOG***	***FINT***	***FNEG***	***FABS***	

Those eight instructions work the same. They replace r0 with the result:

‘C	: r0 = cosine(r0)	‘E	: r0 = exp(r0)
‘S	: r0 = sine(r0)	‘L	: r0 = log(r0)
‘T	: r0 = tan(r0)	‘I	: r0 = int(r0)
‘A	: r0 = atan(r0)	‘N	: r0 = - r0
‘Q	: r0 = sqr(r0)	‘B	: r0 = abs(r0)

<u>Note:</u> for trigonometric functions, values are in radians.

- **With the PreCompiler**, syntax: ***the name without parameter.***

Ex:
```
FCOS                    FEXP
FSIN                    FLOG
FTAN                    FINT
FATN                    FNEG
FSQR                    FABS
```

‘?	***FCOMPARE***	float compare

Syntax: ‘?

Compares r1 to r0, set internal flags and then you can use the IF... set of instructions. The registers r0 and r1 are not affected by the comparison.

Ex: with CALL LINK(“xxx”,A,B)
```
‘G1.0 ‘M0.1   ; get A and store it in r1
‘G2.0  ‘?     ; get B in r0 and compares
?> x          ; jump to label x if A>B.
```

- **With the PreCompiler**, syntax: ***FCOMPARE***

Ex:
```
GETFLOAT 1(0)
FMOVE 0 1
GETFLOAT 2(0)
FCOMPARE
IF>THEN x
```

‘F	***FLOAT***	convert integer to floating point number

Syntax: ‘Fn

Convert n to a floating point number and stores it into r0.

Any value that fits in a word (-32768 to 32767) can be easily used in floating point calculations.

‘F1 : r0 = 1
‘F-62 : r0 = -62
‘FX : r0 = value stored in X

- **With the PreCompiler**, syntax: ***FLOAT n***

Ex:
```
FLOAT 1
FLOAT -62
FLOAT X
```

<u>Note:</u> if a constant is to be used many times (for example in a loop), you should store it in a register better than converting it each time from integer to float.

Ex:
```
'F-4        ; r0 = -4
'M0.7       ; r7 = r0, so r7 = -4
```

Then when you want to use -4 just recall the value of r7 in r0 or r1 (depending on the function) with:

'M7.0 or 'M7.1 because the 'M (move) instruction is faster than 'F.

- **With the PreCompiler**

Ex:
```
FLOAT -4
FMOVE 0 7

FMOVE 7 0
FMOVE 7 1
```

'Z ***INTEGER*** convert floating point number to relative integer

---

Syntax: 'Zv

Convert r0 into an integer stored in variable v.

Ex:
```
'F500 'Q        ; r0 = 500, then take square root
'ZS             ; S = 22 (the integer part of the square root)
```

<u>Note:</u> this conversion erases the content of r0! You must save it if the value is to be used again before using 'Z.

- **With the PreCompiler**, syntax: ***INTEGER v***

Ex:
```
FLOAT 500
FSQR
INTEGER S
```

Joystick and Keyboard

K ***KEY*** ***KEYWAIT*** ***KEYNEW*** read keyboard **Z***

Syntax: Kkn
Scan the keyboard and return a key code into the K variable.

k is a constant selecting the way the keyboard must be scanned:
- 0 returns immediately, variable K=key code or -1 if no key pressed
- 1 wait for a keypress and variable K=key code
- 2 wait for a new key press and K=key code

n is a variable or a constant:
- 0 to scan the whole keyboard
- 1 to scan left part
- 2 to scan right part

- **With the PreCompiler**, the instruction has been separated into three different ones:
  KEY n for K0n, returns immediately
  KEYWAIT n for K1n, wait for a key
  KEYNEW n for K2n, wait for a new key

Ex: K0.2 scan right part and returns either key code or -1 in K.
K1D scan according to D variable, wait for a key press and return the key code in variable K
KMD bad argument error because 1st parameter is not a constant
Lw K1.0 CK$Y !=w this routine wait for a Y pressed.

- **With the PreCompiler**

Ex:
```
KEY 2
KEYWAIT D
(KMD can't be translated into PreCompiler!)
LABEL w
     KEY 0
     COMPARE K $Y
IF<>THEN w
```

(*) The comparison to zero (**Z**) tells you upon return if a key was pressed because if not, the returned value is negative, for example:

K0.0 ?<1 jump to label 1 if no key was pressed
K0.0 !<1 jump to label if a key was pressed

- **With the PreCompiler**

Ex:
```
KEY 0
IF<THEN 1

KEY 0
IF>=THEN 1
```

J	scan joystick	**Z***

Syntax: Jkn
Scan the selected joystick and return button in K and move in X,Y variables.

k is a constant selecting the way the joystick must be scanned:

- 0 return immediately and fill K, X and Y, K=-1 if no button pressed
- 1 wait for button/key and fill K, X and Y
- 2 wait for a move and fill K, X and Y, K=-1 if no button pressed
- 3 wait for a move OR a button/key and fill K, X and Y, K=-1 if no button pressed
- 4 special mode: set the scale for X and Y with n value, in this case n must be a power of 2: 1, 2, 4, 8, 16, 32, 64, 128, 256… For example with J4.16 X and Y will be +/-16. Default mode is J4.4 to return +/-4 like the BASIC does.

n is a variable or a constant:

- 1 for first joystick and left part of keyboard
- 2 for second joystick and right part of keyboard
- in the special mode (k=4), n is the scale.

➢ **With the PreCompiler**, the instruction has been separated into five different ones:

JOYST n	for J0n, returns immediately
JOYSTBUT n	for J1n, wait for button
JOYSTMOVE n	for J2n, wait for move
JOYSTBUTMOVE n	for J3n, wait either for button or move
JOYSTSCALE n	for J4n set the scale

Ex: J1.1 wait for a button/key pressed on joystick 1
J4.1 set scale to one, so X,Y will be +/-1

➢ **With the PreCompiler**

Ex:
```
JOYSTBUT 1
JOYSTSCALE 4
```

(*) The comparison to zero (**Z**) tells you upon return if a key was pressed because if not, the returned value is negative, for example:

J3.2 ?<c wait for a move or a key/button on joystick 2 and jump to label c if no key was pressed (so, only a movement!)
J2.1 !<c wait for a movement on joystick 1 and jump to label c if the button or a key was simultaneously pressed.

➢ **With the PreCompiler**

Ex:
```
JOYSTBUTMOVE 2
IF<THEN c

JOYSTMOVE 1
IF>=THEN c
```

Compare and Jump instructions

L	***LABEL***	set a label for common jumps

Syntax: Lc

Make the current address marked as "c", it can be any character, but for readability I recommend using small capitals or digits:

Any jump to "c" will transfert the program to this location.

- **With the PreCompiler**, syntax: ***LABEL c***

Ex: `LABEL c`

Note: *when using the PreCompiler, upper case labels should not be used (from LA to LZ), they are reserved internally for pseudo-instructions.*

_	***KILL***	kill a label to reuse it

Syntax: _c

Kill the current value of label c for a later reuse.

If label c doesn't exist, there is no effect.

If label c exists and has a resolved value, then it is declared "unresolved"

If label c exists but is unresolved (this means that there are jumps to label c but no Lc encountered), this is an error! Compilation stops and "UR" error is generated.

- **With the PreCompiler**, syntax: ***KILL c***

Ex: `KILL c`

Note: *this instruction is mostly used by the PreCompiler to manage its upper case labels for the pseudo-instructions. You should use it with care!*

B	***GOTO***	branch to a label

Syntax: Bc

Transfer the execution to label c.

- **With the PreCompiler**, syntax: ***GOTO c***

S	***GOSUB***	branch to a subroutine

Syntax: Sc

Tranfer execution to label c and will return when reaching ";".

Subroutine must start with :c (and not Lc) and end with ";".

Only one subroutine level is allowed! This means that you can't call a subroutine from inside another subroutine.

- **With the PreCompiler**, syntax: ***GOSUB c***

:	***SLABEL***	set a label for a subroutine

Syntax: :c

Make the current address marked as “c”, it can be any character, but for readability I recommend using small capitals or digits. This becomes the entry point of a subroutine called by Sc instruction.

You must provide as many “:c” as “;” instructions (every subroutine has only one entry point and one exit point), else an error is issued.

- **With the PreCompiler**, syntax: ***SLABEL c***

;	***RETURN***	end of a subroutine

Syntax: ;

Return from a subroutine.

You must provide as many “:c” as “;” instructions (every subroutine has only one entry point and one exit point), else an error is issued.

- **With the PreCompiler**, syntax: ***RETURN***

C	***COMPARE***	compare two values

Syntax: Cvn

Compare v to n and then you can use the jump instructions according to the comparison.

- **With the PreCompiler**, syntax: ***COMPARE v n***

?=	?>	?<	conditional jumps
IF=THEN	***IF>THEN***	***IF<THEN***	

Syntax: ?=c or ?>c or ?<c

Jump to label c if condition satisfied, c being a character defined by the Lc instruction.

Ex: CAB ?=c compare A and B and jump to c if A=B
CA7 ?>c compare A and 7 and jump to c if A>7
CZ-10 ?<c compare Z and -10 and jump to c if Z<-10

- **With the PreCompiler**, syntax: ***IF=THEN c IF>THEN c IF<THEN c***

Ex:
```
COMPARE A B
IF=THEN c

COMPARE A 7
IF>THEN c

COMPARE Z -10
IF<THEN c
```

If no other instruction is inserted, you can use more than one test on the same comparison:

CAB ?=c ?>d compare A and B and jump to c if A=B or to d if A>B

- **With the PreCompiler**

Ex:
```
COMPARE A B
IF=THEN c
IF>THEN d
```

The result of each arithmetic or variable initialization is automatically compared to ZERO, so you can use a test directly *(this is not true with floating point)* :

-A10 ?<c A=A-10 and jump to c if result is negative.
DB ?=c decrement B by one and jump to c if B=0

- **With the PreCompiler**

Ex:
```
SUB A 10
IF<THEN c

DEC B
IF=THEN c
```

Instructions that perform a comparison to ZERO are marked with a "**Z**" on the right side.

!=	!>	!<	reverse conditional jumps
IF<>THEN	***IF<=THEN***	***IF>=THEN***	

Syntax: !=c or !>c or !<c

Jump to label c if condition not satisfied, c being a character defined by the Lc instruction.

Ex: CAB !=c compare A and B and jump to c if A<>B
CA7 !>c compare A and 7 and jump to c if A<=7
CZ-10 !<c compare Z and -10 and jump to c if Z >= -10

- **With the PreCompiler**, syntax: ***IF<>THEN c IF<=THEN c IF>=THEN c***

Ex:
```
COMPARE A B
IF<>THEN c      <> is the same as "not ="

COMPARE A 7
IF<=THEN c      <= is the same as "not >"

COMPARE Z -10
IF>=THEN c      >= is the same as "not <"
```

The result of each arithmetic or variable initialization is automatically compared to ZERO, so you can use a test directly:

-A10 !<c A=A-10 and jump to c if result is not negative (A >= 0)
DB !=c décrement B by one and jump to c if B <> 0

- **With the PreCompiler**

Ex:
```
SUB A 10
IF>=THEN c

DEC B
IF<>THEN c
```

Instructions that perform a comparison to ZERO are marked with a "**Z**" on the right side.

(	***LIMIT***	check for limits	**Z**

Syntax: (lhv
Check if v is in (l ; h) and limit v to (l ; h). v must be a variable but l,h can be either variables or constants and l <= h.

The "**Z**" indicates that a comparison to ZERO is made, but in this way:
if v is in (l ; h), then considered as EQUAL that you can test with ?=, v unchanged
if v<l, then considered as LESS THAN, that you can test with ?<, and v=l
if v>h, then considered as GREATER THAN that you can test with ?>, and v=h

Ex: (10.99A ?=1 E L1 test if A is in (10 ; 99), if so, jumps to Label 1, else return to Extended Basic with E.

G1A (1.16A &SA get 1st param in A, limit A to (1,16) and use A to set screen color.

- **With the PreCompiler**, syntax: ***LIMIT l h v***

Ex:
```
LIMIT 10 99 A
IF=THEN 1
END
LABEL 1

GETPARAM 1 A
LIMIT 1 16 A
SCREEN A
```

E	***END***	end

Syntax: E
Return to Extended Basic.

A final E is not needed as the compiler provides one. But if your program needs another exit point, you can use "E".

- **With the PreCompiler**, syntax: ***END***

[	***SELECT***	select (start a select / case / endcase structure)

Syntax: [var
Select the variable that will be compared with "%".

Ex: see full example at '%'

- **With the PreCompiler**, syntax: ***SELECT v***

%	***CASE***	case

Syntax: %n or %%
Compares the selected variable to n (variable or constant) and executes the following instructions if values are equal. If you use "%%", then it's the default case when no other has been satisfied, it should be put in the last position!

Ex: Imagine we're viewing pages and the user can select "N" for next page, "P" for previous page, the page number being stored in variable P:

```
La K1.0     Label a, wait for a key and store it into K
[K          Select K
%$N  IP     case 'N', increment P and jump automatically after ]
%$P  DP     case 'P', decrement P and jump automatically after ]
%%   Ba     default for other keys, jump to Label a to read the keyboard again
]           end of select structure
…           here treat the new page!
```

- **With the PreCompiler**, syntax: ***CASE n*** ***or*** ***DEFAULT***

Ex:
```
LABEL a
KEYWAIT 0
SELECT K
CASE $N
     INC P
CASE $P
     DEC P
DEFAULT
     GOTO a
ENDSELECT
```

This structure can't be nested, this means that you can't start a structure within another. After each case "%", a jump to "]" is inserted so you don't have to provide one yourself.

Optimization: if you are sure that your variable has a limited number of possible values, turn the last one into "%%' (the default case) because a case takes up 10 bytes, and default case zero bytes!

For example:

```
/A3   [Z      Z = remainder of A/3, so Z=0, 1 or 2 (no other solution!)
%0            …here if zero…
%1            …here if 1…
%%            …here if 2!...
]
```

- **With the PreCompiler**

Ex:
```
DIV A 3
SELECT Z
CASE 0
     ...
CASE 1
     ...
DEFAULT
     ...
ENDSELECT
```

] ***ENDSELECT*** endcase

Syntax:]
Terminates a structure started with [var.

- **With the PreCompiler**, syntax: ***ENDSELECT***

Screen Instructions

&	VDP prefix in MLC

This character is a prefix before the MLC instructions concerning VDP access (such as read/write a character, set colors, music and sound…)

&G	***GETCHAR***	read a character from the sceen

Syntax: &Grcv

Return into variable V the ASCII code of the character found at location (r,c), r being the row (1-24) and c the column (1-32). This is CALL GCHAR(R,C,V).

Ex: &G1.1A return in A the ASCII code at position (1,1) upper left.
&GR32B return in B the rightmost ASCII code on line R.

- **With the PreCompiler**, syntax: ***GETCHAR r c v***

```
Ex: GETCHAR 1 1 A        or   GETCHAR (1,1) A
    GETCHAR R 32 B       or   GETCHAR (R,32) B
```

&P	***PUTCHAR***	write a character on the screen

Syntax: &Prcv

Write the character v at location (r,c), r being the row (1-24) and c the column (1-32). This is CALL HCHAR(R,C,V).

Ex: &P24.32.65 write a “A” (65) in the lower right corner (24,32).
&PRC32 write a space “ “ in location (R,C).
&PRC$* write a “*” in location (R,C)

- **With the PreCompiler**, syntax: ***PUTCHAR r c v***

```
Ex: PUTCHAR 24 32 64     or   PUTCHAR (24,32) $A
    PUTCHAR R C 32       or   PUTCHAR (R,C) 32
    PUTCHAR R C $*       or   PUTCHAR (R,C) $*
```

&C	***COLOR***	set color

Syntax: &Cgcb

Set the color (c) and background (b) for the group g. Colors are from 1 to 16 and groups from 0 to 31, each one includes 8 characters. (Example group 8 contains characters from 64 to 71, and character Z(90) is in group 90/8 = 11).

Ex: &C8.1.16 set group 8 (starting at character 64) to black (1) on white (16).

- **With the PreCompiler**, syntax: ***COLOR g c b***

```
Ex: COLOR 8 1 16
```

&S	***SCREEN***	set screen color

Syntax: &Sn
Set the screen color to n (from 1 to 16), this is CALL SCREEN(n).

Ex: =A9 &SA set screen color to light red (9).

- **With the PreCompiler**, syntax: ***SCREEN n***

Ex:
```
LET A 9
SCREEN A
```

Note: In modes `TEXT40` and `TEXT80`, the screen instruction also sets the foreground color.
You use: `foreground*16 + background.`

For example, to have magenta chars (14) on black (2), the code is 14*16 + 2 = 226
`&S226` or `SCREEN 226` or `SCREEN &hE2`

&Wn	***TEXT32/40/80***	set screen width

See section **F18A Support.** *(some features available without F18A)*

&Wn	***MULTICOLOR***	set multicolour mode

See section **F18A Support.** *(some features available without F18A)*

&D	***DEFCHAR***	set character definition

Syntax: &Dpic
Set for character c the definition found in string index i from the array passed as parameter p in the CALL LINK statement.
Exemple:
```
DATA C
DATA ”00FF00FF00FF00FF”
DATA "0000000000000000FFFFFFFFFFFFFFFF”
DATA 0101010101010101030303030303030307070707070707070F0F0F0F0F0F0F0F
DATA ””                          ; always end with empty string!
```
Strings are prepared for my program during compilation.

Note: when prepared, a string is turned into binary. This means that, for example, A$(1) will contain the 8 following bytes:
0,255,0,255,0,255,0,255
Because in hexadecimal “00” is 0 and “FF” is 255.

```
CALL LINK(“MYPROG”,C,I,A$())
G1C         ; get first parameter in C
G2I         ; get second parameter in I
&D3IC       ; change char face! (3 because A$() is the 3rd parameter.)
```

- **With the PreCompiler**, syntax: ***DEFCHAR p i c***

Ex:
```
GETPARAM 1 C
GETPARAM 2 I
DEFCHAR 3(I) C
```

If C=65 (code for A) and I=1 then this will use A$(1) to redefine “A”
If C=65 and I=2 this will use A$(2) to redefine both “A” and “B” because A$(2) contains definition for two characters!
If C=48 (code for 0) and I=3, this will use A$(3) to redefine “0”, “1”, “2” and “3” because A$(3) contains definition for four characters!

&d ***XDEFCHAR*** set character definition in extended zone

Syntax: &dpic

Works the same as DEFCHAR (&D) but in extended zone, see ***Pattern discussion*** in the next section.

&X ***XPATTERN*** extended pattern definitions

Syntax : &Xn

Moves either the character definitions or the sprites definitions or both to a new zone to have more characters available.
n = 0 : normal mode as in Extended Basic
n = 1 : characters don't move but sprites can have 256 new definitions
n = 2 : sprites don't move but you can have 256 new characters
n = 3 : both move to a common new zone for 256 definitions

Pattern discussion:

For a set of 256 characters, you need a 256 x 8 = 2048 bytes bloc. The Extended Basic (XB) has defined this zone at address zero in VDP Ram. But, to free more memory for BASIC, other zones overwrite part of this bloc (the screen, colors tables, sprite tables...). That's why you can define freely characters from 32 to 143 but you don't have access to the other.
Another problem came: for example, character 32 (the space) is defined from address 256 to 263 (32 x 8 to 32 x 8 + 7). But these addresses are included into the screen memory! So, Texas decided to internally add an offset of 96 to every ASCII character.
Actually, when a space appears on the screen, it is the character 96 + 32 = 128 that is in memory. When character 143 appears, it is actually the character 143 + 96 = 239.
(You don't have to manage this as XB does it for you. The same for MLC with instructions GETCHAR, PUTCHAR, SPATTERN, DEFCHAR: they all add 96 when writing a byte or subtract 96 when reading it.)
What XPATTERN offers is to change the position of the patterns table in memory to give access to more char definitions without overwriting the other important tables:

XPATTERN 1 (&X1)

The characters are still in the standard zone, so 112 of them are redifinable, but for the Sprites, you have now access to a new table of 256 characters.

To define a character for sprite in the new zone, use XDEFCHAR (&d), the char number can be from 0 to 255 and no offset is applied neither by XDEFCHAR (&d) nor by SPATTERN (#A).

XPATTERN 2 (&X2)

The Sprites remain in the standard zone (112 characters) but the characters are moved to a new zone with 256 new patterns available. To define a character in the new zone, use XDEFCHAR (&d), the char number can be from 0 to 255 and no offset is applied neither by GETCHAR (&G) nor by PUTCHAR (&P).

XPATTERN 3 (&X3)

Both sprites and characters are moved to this common zone of 256 new characters. To define a character or a sprite in the new zone, use XDEFCHAR (&d), the char number can be from 0 to 255 and no offset is applied neither by GETCHAR (&G) nor by PUTCHAR (&P) or SPATTERN (#A).

XPATTERN 0 (&X0)

To return to XB, you should reset the standard addresses. This is not safe as the new pattern zone may conflict with XB data. But, if your program don't use too much strings or arrays, this should be okay.

Note: See at *"F18A Support"* for use of XPATTERN with the new graphic modes.

Sprites

There are various differences between the Basic and Assembly codes for sprites. But My Little Compiler does the conversion and you program them like you would do with the Extended Basic, so:

28 available sprites numbered from 1 to 28 (0 to 31 in assembly)
Row position from 1 to 192 (-1 to 190 in assembly)
Column position from 1 to 256 (0 to 255 in assembly)

Sprite prefix

This is the prefix character for all sprite instructions in MLC.

#A ***SPATTERN*** change sprite pattern

Syntax: #Asc
Set character c for sprite s *(see **&X, XPATTERN** for extended characters)*

Ex: #A1$* set character "*" for sprite 1

- **With the PreCompiler**, syntax: ***SPATTERN s c***

Ex:
```
SPATTERN 1 $*
```

#C ***SCOLOR*** change sprite color

Syntax: #Csc
Set color c for sprite s.

Ex: #C3.16 set color 16 (white) for sprite 3.

- **With the PreCompiler**, syntax: ***SCOLOR s c***

Ex:
```
SCOLOR 3 16
```

#L ***SLOCATE*** locate or create a sprite

Syntax: #Lsrc
Put the sprite s at position (r,c). s,r,c can be either constants or variables.

Ex: #L1AB put sprite 1 at location (A,B)
#L3.1.1 put sprite 3 in the upper left corner of the screen.

- **With the PreCompiler**, syntax: ***SLOCATE s r c***

Ex:
```
SLOCATE 1 A B       or      SLOCATE 1 (A,B)
SLOCATE 3 1 1       or      SLOCATE 3 (1,1)
```

If the sprite is created with this function, you must specify before its color and pattern, so a complete creation would be:

#A1$F #C1.16 #L1.100.100 ; create sprite 1, pattern F, white at (100,100)

#M ***SMOTION*** set sprite motion

Syntax: #Msrc
Set the automatic motion of sprite s to (r,c) values are from -128 to +127.

- **With the PreCompiler**, syntax: ***SMOTION s r c***

Ex: `SMOTION s r c` or `SMOTION s (r,c)`

#P ***SPOSITION*** get sprite position

Syntax: #Psrc
Get the position of sprite s and return it into r and c. R and c must be variables and s can be either a variable or a constant.
Ex: #P3AB get position of sprite 3 and put row in A, column in B.

- **With the PreCompiler**, syntax: ***SPOSITION s r c***

Ex: `SPOSITION 3 A B` or `SPOSITION 3 (A,B)`

#& ***SCONVERT*** convert coordinates

Syntax: #&rc
Convert coordinates of a sprite (1-192/1-256) to a character location (1-24/1-32).
Note: *previous values of variables r and c are lost!*

Ex: #P1AB #&AB &GABC put in A,B the position of sprite #1, convert them and then get in C the character under that location.

- **With the PreCompiler**, syntax: ***SCONVERT r c***

Ex:
```
SPOSITION 1 (A,B)
SCONVERT (A,B)
GETCHAR (A,B) C
```

#D ***SDISTANCE*** sprite distance

Syntax: #Dss'v
Compute the distance between sprites s and s' and return it in v. V must be a variable and s/s' can be either variables or constants.

- **With the PreCompiler**, syntax: ***SDISTANCE s s' v***

The distance is not the cartesian one, but an approximation with a faster algorithm *(the difference can be up to 7,7% with a mean value of 4,8%).*

Details: Assuming that Sprite s is at (x1,y1) and s' at (x2,y2)

$$\Delta x = |x1 - x2| \qquad \Delta y = |y1 - y2|$$

$$M = \max(\Delta x, \Delta y) \quad and \quad N = \min(\Delta x, \Delta y)$$

$$Dist = (5M + 2N)/5$$

#I	***INTERRUPT***	enable interrupt for automatic motion

Syntax: #I

Insert assembly instructions that allows VDP interrupt to automatically move the sprites when they have a motion defined. This "#I" must be regularly inserted in your routine else, sprites won't move smoothly.

For example if you have a long "wait" loop, then put #I in it.

To ease your work, My Little Compiler automatically insert #I into every K and J instruction (wait for Key or Joystick) as soon as a #I instruction is encountered. So if your loop includes J or K, just put one #I before the loop and that's it, interrupts will work.

- **With the PreCompiler**, syntax: ***INTERRUPT***

Note: *Even if your sprites are yet in motion from Extended Basic you have to use #I to allow your sprites to move as CALL LINK disables the interrupts.*

#K	***SDELETE***	delete a sprite

Syntax: #Ks

Delete sprite s in the same manner as the Extended Basic would do (for compatibility). This means that its motion is set to (0,0) and its position to [194,1] *(this is [192,0] in VDP RAM.).* Therefore, motion and position are lost, but pattern and color preserved if you have to create it again.

- **With the PreCompiler**, syntax: ***SDELETE s***

#<	***SMAX***	set the maximum number of sprites in motion

Syntax: #<n

Tells the system that sprites 1 to n can automatically move. If n=0 then they all are stopped.

If n is greater than previous value, then new moving sprites have their motion set to zero, so #M must be used after #<.

If n is lower than previous value, then excluded sprites are stopped and their motion lost.

Ex:	#<5	tells the system that sprites 1 to 5 can move
	#<0	stop every sprite
	#<A	takes value from A to set the maximum sprite # to be in motion.

- **With the PreCompiler**, syntax: ***SMAX n***

Ex:
```
SMAX 5
SMAX 0
SMAX A
```

Note: *If your sprites are yet in motion when calling CALL LINK, you don't have to use this instruction as the Extended Basic has correctly set up things. Use it to modify the state! (Add new sprites in motion, or remove moving sprites).*

Logical instructions

H Logical prefix

This is the prefix character for all logical instructions in MLC *(Why H? No other interesting letter was free...)*

HA ***AND*** logical AND

Syntax: HAVn

Performs a logical AND between variable V and value n (constant or variable).
Bits are set to "1" if they are both to "1" in V and in n.

- **With the PreCompiler**, syntax: ***AND v n***

HO ***OR*** logical OR

Syntax: HOVn

Performs a logical OR between variable V and value n (constant or variable).
Bits are set to "1" if they are to "1" in V or/and in n.

- **With the PreCompiler**, syntax: ***OR v n***

HX ***XOR*** logical XOR

Syntax: HXVn

Performs a logical XOR between variable V and value n (constant or variable).
Bits are set to "1" if they are different in V and in n.

- **With the PreCompiler**, syntax: ***XOR v n***

HN ***NOT*** logical NOT

Syntax: HNV

Performs a logical NOT into variable V.
Bits are changed from "1" to "0" or from "0" to "1".

- **With the PreCompiler**, syntax: ***NOT v***

HW	***SBYTES***	swap bytes

Syntax: HWV

Swap bytes into variable V. The most significant byte and the less significant byte are exchanged.

- **With the PreCompiler**, syntax: ***SBYTES v***

H<	***SHIFTLA***	logical SHIFT LEFT ARITHMETIC

Syntax: H<Vn

Shift bits in variable V n positions to the left. Empty bits are “0”

- **With the PreCompiler**, syntax: ***SHIFTLA v n***

H>	***SHIFTRA***	logical SHIFT RIGHT ARITHMETIC

Syntax: H>Vn

Shift bits in variable V n positions to the right. Empty bits are “0” or “1” according to the sign of V before shifting.

- **With the PreCompiler**, syntax: ***SHIFTRA v n***

HL	***SHIFTRL***	logical SHIFT RIGHT LOGICAL

Syntax: HLVn

Shift bits in variable V n positions to the right. Empty bits are “0”.

- **With the PreCompiler**, syntax: ***SHIFTRL v n***

HR	***ROTATER***	logical ROTATE RIGHT

Syntax: HRVn

Rotate bits in variable V n positions to the right.

- **With the PreCompiler**, syntax: ***ROTATER v n***

HC	***CARRY***	get last “CARRY” bit

Syntax: HC

Store in Z the last CARRY bit from a Shift/Rotate instruction.
For SHIFT instructions, this is the last lost bit.
For ROTATE, this is the last bit to change from right to left.
So, Z=0 or Z=1.

- **With the PreCompiler**, syntax: ***CARRY***

Note: *this instruction must be as close as possible to the last shift/rotate instruction because the carry bit is stored into a temporary register that can be overwritten!*

Set instructions

F set prefix

This is the prefix character for all set instructions in MLC.

FD ***DIMSET*** dimension of a set

Syntax: FDnmV

Create an empty set that can contain values from n to m (both must be constants) and returns the address of the set in variable V.

In the set, every element is a single bit (0 if absent, 1 if present in the set).

- **With the PreCompiler**, syntax: ***DIMSET (n,m) v***

Memory usage: a set requires 3 words plus one bit per element.

For example, with FD5.80A there are 76 elements, so 76 bits required, but the memory is word aligned so MLC will use 80 bits (5*16).

Finally: 3 * 2 (because words) + 80 / 8 = 16 bytes used.

F+ ***ELEMENT+*** add an element in the set

Syntax: F+Vn

Add element n into set pointed by V.

If the element is already in V, then nothing happens.

If n is outside the range specified when creating the set, this can lead to an unpredictable behaviour.

- **With the PreCompiler**, syntax: ***ELEMENT+ v n***

F- ***ELEMENT-*** remove an element from the set

Syntax: F-Vn

Remove element n from set pointed by V.

If the element is not in V, then nothing happens.

If n is outside the range specified when creating the set, this can lead to an unpredictable behaviour.

- **With the PreCompiler**, syntax: ***ELEMENT- v n***

F?	***ELEMENT?***	test the presence an element in the set	**Z**

Syntax: F?Vn
Test if n is into set pointed by V.
If the element is in, then “equality” (you can test it with ?=...)
If the element is not in the set, then “no equality" (you can test this with !=...).
If n is outside the range specified when creating the set, this can lead to an unpredictable behaviour.

- **With the PreCompiler**, syntax: ***ELEMENT? v n***

FO	***SETFILL***	fill the set

Syntax: FOV
Fill set pointed by variable V.
All elements are added, the set is filled with ones.

- **With the PreCompiler**, syntax: ***SETFILL v***

FZ	***SETCLEAR***	clear the set

Syntax: FZV
Clear set pointed by variable V.
All elements are removed, the set is filled with zeros.

- **With the PreCompiler**, syntax: ***SETCLEAR v***

FN	***SETCOMP***	complement of the set

Syntax: FNV
Change the set pointed by variable V to its complement.
All elements that were not included are now included, all elements that were included are now removed. Zeros and ones are exchanged.

- **With the PreCompiler**, syntax: ***SETCOMP v***

FC	***SETCARD***	cardinal of the set	**Z**

Syntax: FCVV’
Count the number of elements in the set pointed by variable V and returns the value into variable V’.

- **With the PreCompiler**, syntax: ***SETCARD v v'***

FF	***SETFINDNEXT***	find next element in the set	**Z**

Syntax: FFVV'
Find the next element in the set pointed by variable V starting at element V'.
If found: return "equality" and value in V' (so you can use ?=...)
If not found: return "no equality" and V' remains unchanged (so you can use !=...)

Example: FD3.10A so A is a set of [3,10].
F+A4 F+A8 add elements 4, 8

=B3
FFAB find next element starting at 3, so B=4 and "equality"
?=c so it will jump to label c

=B5
FFAB find element starting at 5, so B=8 and "equality"

=B9
FFAB nothing to find, so returns "no equality"
!=c so it will jump to label c

- **With the PreCompiler**, syntax: ***SETFINDNEXT v v'***

Ex:
```
DIMSET (3,10) A
ELEMENT+ A 4
ELEMENT+ A 8

LET B 3
SETFINDNEXT A B
IF=THEN c

LET B 5
SETFINDNEXT A B

LET B 9
SETFINDNEXT A B
IF<>THEN c
```

FU	***SET+***	union of two sets

Syntax: FUVV'
Computes the union of the two sets, V=V union V'
So V contains the elements that were in V or in V'.

- **With the PreCompiler**, syntax: ***SET+ v v'***

Note: V and V' must have the same definition.

FI	***SET****	intersection of two sets

Syntax: FIVV'
Computes the intersection of the two sets, V=V inter V'
So V contains the elements that were both in V and in V'.

- **With the PreCompiler**, syntax: ***SET* v v'***

<u>Note:</u> V and V' must have the same definition.

FX	***SET-***	difference of two sets

Syntax: FXVV'
Computes the difference of the two sets, V=V - V'
So V contains the elements that were in V but not in V' (in other words, this removes from V the elements that were in V').

- **With the PreCompiler**, syntax: ***SET- v v'***

<u>Note:</u> V and V' must have the same definition.

Music, Sound, Speech

Sounds must be prepared through the SOUND section before you run your own assembly routine that will use them. It's a kind of "compilation" because you will write strings describing your sound that "My Little Compiler" will turn into binary codes that the sound processor can understand. The Extended Basic does the same from a CALL SOUND statement to binary codes.

Those codes must be located into VDP RAM, so, the easiest solution I found is to store them into a string array, each string storing a sound, so the sound number is limited only by the amount of free memory for your strings. The sound section is compiled with the other sections using this call:

Syntax: **`CALL LINK("COMPIL",IO(),S$(),C$())`**

Then the S$() array can be passed to your assembly routine that will play them using the "&M" function, the **S$()** array **must be the first parameter** in the CALL LINK statement.

An example:

```
DATA S                  ; start a sound section
DATA "2"                ; store binary codes in string S$(2) (*)
DATA "FA261VA2D30"      ; On voice A play a C (261 Hz) with volume 2 and 30/60 sec
DATA "FA392D15"         ; On voice A play a G (392 Hz) same volume and 15/60 sec
DATA "VA15D0"           ; shut down sound (volume 15 = no sound)
DATA ""                 ; end
```

(*) Note that with :

```
DATA "2T"               the sound is tested so you can hear it during this compilation.
```

Then I can use the &M function into my assembly routine to play this sound #2. If so, in the parameter list, the S$() array must be the first parameter.

```
CALL LINK("MYASM",S$(),…)
```

&M	***SOUND***	***SOUNDQUEUE***	***SOUNDWAIT***	play a music or a sound

Syntax: &Mkn
Play sound n with mode k. Both can be either constant or variables.

k determines how the sound is played:

- k=0 don't wait for previous sound to stop, play this one and execute next instruction immediately.
- k=1 wait for previous sound to stop, then play this one and execute next instruction immediately.
- k=2 wait for previous sound to stop, play this one and wait until it is finished before executing next instruction.

n represents the index of the string containing the sound in the array passed as **the first argument** in the CALL LINK statement.

Example: CALL LINK("TEST",S$(),…)

&M0.3 play sound stored into S$(3) and don't wait
=V10 &M2V play sound stored into S$(10) and wait before going on.

- **With the PreCompiler**, the instruction has been separated in three different ones:

SOUND n	for	&M0n, play immediately
SOUNDQUEUE n	for	&M1n, wait and play
SOUNDWAIT n	for	&M2n, wait, play and wait

Ex:
```
SOUND 3
LET V 10
SOUNDWAIT 2 V
```

Interrupts must be enabled in order to play sound correctly, the same problem as with the sprite automatic motion. So you must provide a **#I** (or ***INTERRUPT*** with the PreCompiler) instruction regularly knowing that:

- wait on keyboard and joystick contain their own interruption instructions if one #I has been met before
- wait on sound end (&M1n and &M2n) does the same.

So regular #I are to be used if no joystick, nor keyboard, neither sound waits are performed.

Let's return to the sound compilation…

Sound description:

Your TI-99 is able to produce three tones (voices A,B and C) and one noise (voice N) simultaneously. Each voice can have its own frequency (from 110 Hz to 55938 Hz for tones and 1 to 8 for noises). Each voice can have its own volume (0 high, 14 low, 15 OFF).

When sounds are simultaneous, only one duration must be provided, it is in 60^{th} of seconds (from 0 to 255). For example D=30 is half a second, and max value is D=255 for 4,25 seconds.

So the sound description is a list of groups each one with frequencies, volumes and one duration.

The general form of a group:

"frequency informations volume informations duration"

Frequency informations start with a **"F"** and then you specify each voice **(A,B,C or N)** followed by its frequency (from 110 to 55938):

Ex: `FA220C440`

Volume informations start with a **"V"** and then you specify each voice **(A,B,C or N)** followed by its volume (from 0 to 15):

Ex: `VA1C5`

Duration information start with a **"D"** followed by the duration in 60^{th} of second (from 0 to 255):

Ex: `D10`

A complete group can be:

```
FA220C440VA1C5D10
```

If, from one group to the next, the frequency doesn't change, you just have to provide the volumes informations, example:

```
FA440VA0D2      * a A with high volume
VA2D3           * decrease volume
VA4D4           * but increase duration
VA6D5           * this gives a note slowly decreasing
VA8D6
VA12D7
VA14D8
VA15D0          * stop voice A with both volume = 15 and duration = 0
```

In the same way, if volume doesn't change, you just have to provide frequencies from one group to another:

```
FA440VA0D10     * a A with high volume
FA554D10        * a C# with same volume
FA659D20        * a E with same volume
VA15D0          * stop sound
```

Important: ***what you can't omit is the duration that validates a group.***

Not less important: don't forget to **stop the sound** with **volumes at 15** for each voice used and **duration at zero**.

- **With the PreCompiler**, the previous sounds could be defined as:

Ex:

```
$SND 1     ;  this will be sound 1 in S$(1), use with  SOUND 1
     FA440VA0D2 VA2D3 VA4D4 VA6D5
     VA8D6 VA12D7 VA14D8 VA15D0
$$
$SND 2     ;  this will be sound 2 in S$(2), use with  SOUND 2
     FA440VA0D10 FA554D10
     FA659D20 VA15D0
$$
```

&T ***SAY*** say rom word

Syntax: &Tn
n is the ROM address of a word into the Speech Synthesizer memory.

Example: &T13594 will say HELLO (13594 = >351A address for hello)

- **With the PreCompiler**, syntax ***SAY n***

Ex:

```
SAY 13594
SAY &h351A      ; same address but in hexadecimal
SAY &wHELLO     ; same address but by word directly !
```

Note: The PreCompiler knows all the standard vocabulary of the Speech unit. Some contain several words, like TEXAS INSTRUMENTS, you can't use the space, you must replace it with an underscore:

```
SAY &wTEXAS_INSTRUMENTS
```

&?	***TALKING?***	is talking?

Syntax: &?
Returns "=" if the unit is talking (so you have to wait before saying another word)
Returns "<>" if the unit is ready for another word.

Example:

```
&T13594          ; say HELLO
Lx &? ?=x        ; wait loop until <>
&T29118          ; say YOU
```

- **With the PreCompiler**, syntax ***TALKING?***

Ex:
```
SAY &wHELLO          or        SAY &wHELLO
LABEL x                        REPEAT
     TALKING?                       TALKING?
IF=THEN x                      UNTIL<>
SAY &wYOU                      SAY &wYOU
```

&U	***SAYUSER***	say user word

Syntax: &Unc
If c=1 then n is the even CPU address of the data starting with a word containing the number of bytes to send to the Speech unit.
If c=-1 then n is the VDP address of the data and Z must contain the number of bytes *(this organization to be coherent with the TG instruction when receiving a string address).*

Example: with CALL SPGET("TWENTY",T$)
CALL LINK("TEST",T$)

```
=U1              ; to get the first parameter
TGU0A            ; A=vdp string address and Z=string length
&UA-1            ; say the word "TWENTY" from vdp ram.
```

- **With the PreCompiler**, syntax ***SAYUSER n c***

Ex:
```
LET U 1
TABLEGET U(0) A
SAYUSER A -1
```

File access

The TI-99/4A always uses the VDP ram for file transfers. Two zones are necessary:

- The data bloc where bytes read from or written to the file are stored
- The PAB (peripheral access bloc) that describes the file and the operations to perform

PAB discussion

This is a bloc with 10 bytes plus the file name you have to create in VDP Ram. Some operations don't use every entry, you have to fill every useful entry yourself except the OpCode that is sent with the instruction:

BYTE	**OpCode**	*File operation (0 to 9), see below*
BYTE	**flags**	*See below*
WORD	**Data Address**	*Address of data bloc in VDP Ram*
BYTE	**Record length**	*For fixed files or max length for variable files*
BYTE	**Count**	*In input for FWRITE/FREAD and in output for FREAD*
WORD	**Record number**	*For relative files or size for FSAVE/FLOAD*
BYTE	**Screen Offset**	*Used for cassette operations (>60 for BASIC, >00 for E/A)*
BYTE	**Name length**	
STRING	**Name**	*Ex: DSK1.MYFILE or CS1*

Flags:

This is the sum of different values to describe the file:

SEQUENTIAL	0	*Sequential access to the file*
RELATIVE	1	*Access with Record Number*
UPDATE	0	*Read and write allowed*
OUTPUT	2	*Only write allowed*
INPUT	4	*Only read allowed*
APPEND	6	*Add data to the end of the file but you can't read*
DISPLAY	0	*Data are displayable chars, a text file for example*
INTERNAL	8	*Data are binary bytes, a character definition for example*
FIXED	0	*Fixed record length*
VARIABLE	16	*Variable record length*

For example, to write to a text file, I select:
SEQUENTIAL + OUTPUT + DISPLAY + VARIABLE = 0 + 2 + 0 + 16 = 18.

If I want to read sequentially binary data in blocs of 32 bytes:
SEQUENTIAL + INPUT + INTERNAL + FIXED = 0 + 4 + 8 + 0 = 12.

- See ***FLAGS*** in the **Pseudo-Instruction** chapter to ease the operation

Error codes:

After a file operation, the Flags byte also contains an error code or zero, you don't have to manage this as MLC returns automatically the error code into Z.

0	*No error*
1	*Device is write protected*
2	*Open error: flags don't match the file*
3	*Illegal operation (not supported by device or don't match the Open flags)*
4	*Out of table or buffer space on the device*
5	*Read after file end, this closes the file.*
6	*Hardware device error.*
7	*File error (prog/data mismatch, read from a non existing file...)*
8	*Bad device name*

Where to locate the PAB and Data bloc?

Extended Basic uses part of VDP Ram to store the variables names and variables values (strings, numbers, arrays...). There is a pointer at address >831A containing the VDP address of the string/variables blocs. Under this address, the Ram should be free, but take care not to overwrite the char definitions, the sprites and screen.

1) With the PreCompiler

Example, you want to access to a text file with a maximum length of 80 chars:

```
STARTDATA
     BYTE 0          ; no OpCode for now
     FLAGS SEQUENTIAL,DISPLAY,VARIABLE,INPUT
     WORD 0          ; address will be computed later
     BYTE 80         ; max length
     BYTE 80         ; length
     WORD 0          ; no rec number for sequential
     BYTE 0          ; offset unused
     BYTE 11         ; name is 11 bytes long
     STRING "DSK1.MYTEXT"
ENDDATA A

LET B &h831A
GETTABLE B(0) B      ; peek(831A) = VDP address

SUB B 22             ; place for PAB so B = PAB ADDRESS
LET C B
SUB C 80             ; place for one string, C = DATA ADDRESS
PUTTABLE A(1) C      ; data address stored into PAB (i.e word A(1))
BMOVECTOV A 22 B     ; copy PAB to VDP Ram
```

That's all for initialization, then you can use the file instructions with B as PAB address such as:

```
FOPEN B     to open the file
FREAD B     to read from the file
Etc...
```

2) <u>With MLC</u>

```
BA T<                       ; jump over table and keep bloc address
     TO0                    ; no Opcode for now
     TO20                   ; sequential, display, variable, input
     TT0                    ; address will be computed later
     TO80                   ; max length
     TO80                   ; length
     TT0                    ; no rec number for sequential
     TO0                    ; offset unused
     TO11                   ; name is 11 bytes long
     TS11DSK1.MYTEXT
     TE                     ; ensure even address
LA T>A                      ; get bloc address in A
=B-31974                    ; B=>831A
TGB0B                       ; peek B into B
-B22                        ; B=B-22 to get place for PAB
=CB
-C80                        ; C=PAB-80 to get place for string
TPA1C                       ; store C into data address of PAB (i.e word A(1))
TCA22B                      ; copy PAB to VDP Ram
```

That's all for initialization, then you can use the file instructions with B as PAB address such as:

```
Y0B   to open the file
Y2B   to read from the file.
Etc...
```

Ram conflicts...

When you get back to Extended Basic, the interpreter may use the RAM where you put your PAB and Data Bloc. So, don't expect to find it again when back in MLC. The safer is to create a PAB in CPU Ram and to copy it to VDP Ram each time you get into your assembly program using the >831A pointer.

Y0	***FOPEN***	open a file	**Z**

Syntax: Y0n or ***FOPEN n***

Open the file whose PAB is at address n (constant or variable). Return the error code in Z.

Input: Flag, Record length, Name
Output: If Record Length=0 in input, returns the Record length of the file.

Y1	***FCLOSE***	close a file	**Z**

Syntax: Y1n or ***FCLOSE n***

Close the file whose PAB is at address n (constant or variable). Return the error code in Z.

Input: Flag, Name
Output: In OUTPUT or APPEND mode, an EOF record is written.

Y2	***FREAD***	read from a file	**Z**

Syntax: Y2n or ***FREAD n***

Read a bloc of bytes from the file whose PAB is at address n (constant or variable). Return the error code in Z.

Input: Flag, Data Address, Record length, Name, Record Number (if relative)
Output: Count is filled with the actual bytes read.

Y3	***FWRITE***	write to a file	**Z**

Syntax: Y3n or ***FWRITE n***

Write a bloc of bytes to the file whose PAB is at address n (constant or variable). Return the error code in Z.

Input: Flag, Data Address, Record length, Name, Record Number (if relative)
Count (is the number of bytes to write)
Output: None.

Y4	***FRESTORE***	set file position	**Z**

Syntax: Y4n or ***FRESTORE n***

Set position into the file whose PAB is at address n (constant or variable). Return the error code in Z.

If sequential, the pointer is restored at the beginning of the file.

If relative, the pointer is restored at the Record Number specified.

Input: Flag, Record length, Name, Record Number (if relative)

Output: The pointer is positioned as indicated.

Note: *FRESTORE works only for INPUT or UPDATE modes. With a relative file, you can simulate a FRESTORE by filling Record Number for any other operation even with OUTPUT and APPEND modes.*

Y5	***FLOAD***	load a file	**Z**

Syntax: Y5n or ***FLOAD n***

Read the whole file whose PAB is at address n (constant or variable). Return the error code in Z.

Input: Data Address, Name, Record Number (is the number of bytes to read)

Output: The Data Bloc is filled with the whole file (limited by Rec Num)

Note: *no previous FOPEN operation is needed.*

Y6	***FSAVE***	save to a file	**Z**

Syntax: Y6n or ***FSAVE n***

Write a bloc to the file whose PAB is at address n (constant or variable). Return the error code in Z.

Input: Data Address, Name, Record Number (is the number of bytes to save)

Output: The Data Bloc is copied to the file.

Note: *no previous FOPEN operation is needed.*

Y7	***FDELETE***	delete a file	**Z**

Syntax: Y7n or ***FDELETE n***

Read the whole file whose PAB is at address n (constant or variable). Return the error code in Z.

Input: Name

Output: The file is closed and deleted.

Y8	***FSCRATCH***	delete a record	**Z**

Syntax: Y8n or ***FWRITE n***

Delete a record from the file whose PAB is at address n (constant or variable). Return the error code in Z.

Input: Flag, Record length, Name, Record Number
Output: The record pointed by Record Number is deleted from the file.

Y9	***FSTATUS***	returns informations about a file	**Z**

Syntax: Y9n or ***FSTATUS n***

Returns the status byte information.

Input: Name
Output: Screen Offset contains a bit field as described below:
STATUS BYTE `a b c d e f g h`
`Bit a = 1` if file does not exist
`Bit b = 1` if file is protected
`Bit c = 0` unused
`Bits de = 10` data internal, `00` data display, `01` program
`Bit f = 1` variable length, else fixed length
`Bit g = 1` physical device end reached
`Bit h = 1` EOF

F18A support

The TMS9918 of the TI-99/4A can be replaced with a new F18A with extra features. For example:

- using a **4 color BML (BitMap layer)**
- using the **second processor**, called the GPU, that can work directly on the VDP RAM with high performances (See ***$GPU section*** for more informations.).
- using the **timer** (a clock in the TI!)
- extended mode with **40/80 columns**.

	XREGENABLE	enable use of extra F18A registers

Syntax: ***XREGENABLE***

This is a pseudo code equivalent to VDPREG 57 28 twice (or QV57.28). This is the sequence that tells the F18A to give access to the extended registers.

Note**: if you use a $GPU section**, this instruction is useless because the compiler enables the extended registers to install the GPU program.

If not, **you MUST use XREGENABLE** else, none of the following instructions will work.

QV	***VDPREG***	write a value in a VDP register

Syntax: QVnv or ***VDPREG n v***

GPU : programming the Co-Processor

QR	***GPURUN***	run a program in VDP RAM

Syntax: QRn or ***GPURUN n***

Run a program on the GPU processor located at VDP address n, typically >4000 if you use the standard settings of the PreCompiler.

QW	***GPUWAKE***	re runs a previous stopped program in VDP RAM

Syntax: QW or ***GPUWAKE***

Start again a program where it stopped by itself or with GPUSLEEP.

QS	***GPUSLEEP***	stops a running program in VDP RAM

Syntax: QS or ***GPUSLEEP***

Stops the running program in VDP Ram.

A GPU program can return to sleep mode by itself with the IDLE assembly instruction.

Q?	***GPUSTATE***	returns GPU processor state	**Z**

Syntax: Q? or ***GPUSTATE***

Returns "EQUAL" if the GPU is stopped (Idle mode) or "Not EQUAL" if the GPU is running.

Example:

```
GPURUN &h4000        ; run program at >4000
REPEAT
     GPUSTATE        ; get state
UNTIL=               ; wait until finished (EQUAL)
```

BML : enhanced graphics with the BitMap Layer

The **BML (BitMap layer)** is a kind of super sprite with definable position and size. It has four colors, so each byte contains 4 pixels of 2 bits each.

To define a BML, you must create a byte table with 6 bytes as this:

BYTE 0 : flags 8 bits as this `eptxcccc`
- Bit e : enable BML if 1
- Bit p : priority over tiles if 1
- Bit t : transparency (color 0 transparent) if 1
- Bit x : unused
- Bits cccc : palette number from 0 to 15.

BYTE 1 : BML address in VDP RAM, this byte times 64.
BYTE 2 : X position of the BML from 0 to 255
BYTE 3 : Y position from 0 to 191
BYTE 4 : W width up to 256
BYTE 5 : H height up to 192

When transparency is set, color 0 is transparent and only colors 1 to 3 are seen.

Example:

```
STARTDATA
     Byte &b11100011      ; enable, prior, trans + palette #3
     Byte &h40            ; VDP address &h40 times 64 = &h1000 = 4096.
     Bytes 10,10,32,32    ; position 10,10 and size 32,32
ENDDATA E
```

So you can modify individual bytes using `GETTABLE` or `PUTTABLE` with `E(i)`.

QB ***BMLSET*** set bitmap layer information

Syntax: QBma or ***BMLSET m a***

Set the current BML definition block (for use with PLOT, DRAWTO and FILLRECT) and if m<>0 then send the inner information to the F18A hardware.

Example: (with previous definitions)

```
BMLSET 1 E           ; define my BML and make it appear (m=1)
PUTTABLE E(2) 50     ; change X to 50
BMLSET 1 E           ; apply new changes in the hardware.
```

QP ***BMLPLOT*** plot a pixel in the BML

Syntax: QPxyc or ***BMLPLOT x y c***

Plots pixel (x,y) in color "c".
The x position runs from 0 to W-1, y from 0 to H-1, upper left corner is (0,0).
The color c goes from 0 to 3.

Note: **no check for limits is performed**, so take care! You could delete data in VDP RAM outside your BML zone.

QD ***BMLDRAWTO*** draw a line in the BML

Syntax: QDxyc or ***BMLDRAWTO x y c***

Draws a line from the last plotted pixel to (x,y) in color "c". Several DRAWTO can be chained.

Example:

```
BMLPLOT 0 0 2          ; first pixel
BMLDRAWTO 30 30 2      ; draw from (0,0) to (30,30) in color 2
BMLDRAWTO 0 30 1       ; draw from (30,30) to (0,30) in color 1.
```

Note: **no check for limits is performed**, so take care! You could delete data in VDP RAM outside your BML zone.

QF ***BMLFILLRECT*** fill rectangle in the BML

Stntax: QFwh or ***BMLFILLRECT w h***

Fills a rectangle with dimensions (w,h) taking the last plotted pixel as the upper left corner. Limitations are these:

- The color is the same as the last plotted pixel
- The width must be multiple of 4
- The X position (of upper left corner) must be a multiple of 4.

Example: (with previous definitions), this clears the BML at start

```
BMLSET 0 E             ; just define my BML
BMLPLOT 0 0 0          ; upper left pixel in color 0
BMLFILLRECT 32 32      ; fills the whole BML with 0, so clear it!
BMLSET 1 E             ; now sends info to the F18A, the BML appears
```

Note: **no check for limits is performed**, so take care! You could delete data in VDP RAM outside your BML zone.

TIMER : programming the 32bits counter

The **F18A** has a **internal 32bits counter** that can be used as a clock (with a precision of 10ns) or as an event counter. Four bits are used to control it in the F18A through the extended register 37.

With 32bits and 10ns the limit is: $(2^{32}-1)*10^{-8}$ = 42,95 seconds.

MLC brings a second **6 bytes memory space** where you can load the counter value or cumulate it to bypass the 32bits limit. The internal counter and the MLC zone are independent, so you can clear one without affecting the other.

With 47bits and 10ns, the limit is: $(2^{47}-1)*10^{-8}$ = 390 h 56' 14".

QT ***TIMER*** manages the 32bits counter

Syntax: QTm or ***TIMER ..options...***

"m" is a bit mask equal to the sum of each option's value:

PreCompiler	**MLC**	**In R37**	**Action taken**
CLEAR	32768		Clear MLC counter
READ	16384		Read F18A counter into MLC counter
SUM	8192		Read F18A and sum into MLC counter
TOFLOAT	4096		Convert MLC counter to float in r0
(*)	2048		If set, does not modify R37
RESET	8	Yes	Clear F18A counter
LOAD	4	Yes	Load F18A counter with values from xreg 38 to 41
RUN	2	Yes	Run F18A counter as a 10ns clock
INC	1	Yes	Increment by one F18A counter
STOP	0	Yes	If it was running, then stop F18A clock

(*) on the PreCompiler, if none of the R37 options is specified, then this bit is set automatically, so you don't have to care about it.

Example: you just want to know the duration of a portion of code that should last less than 43 seconds.

- **With the PreCompiler**

```
TIMER RESET RUN            ; timer set to zero and run
...your code here...
TIMER READ STOP TOFLOAT    ; get timer value and turns it into a float
.. and for example..
PUTFLOAT n i               ; to send back the float value to BASIC
```

- **The same with MLC**

```
QT10          ; 10 = 8 + 2 (RESET + RUN)
... your code here ...
QT20480       ; 20480 = 16384 + 4096 + 0 (READ + TOFLOAT + STOP)
'ZA
or
'Pni
```

Example: you want to know the duration of a portion of code that can be repeated and that can last more than 43 seconds.

- **With the PreCompiler**

```
TIMER CLEAR           ; init: clear MLC zone for sum
…
TIMER RESET RUN       ; start timer
...your code here...
TIMER SUM             ; sum into MLC zone
…
TIMER TOFLOAT         ; total into float register r0
```

- **The same with MLC**

```
QT32768               ; 32768 = CLEAR
…
QT10                  ; 10 = 8 + 2 = RESET + RUN
… your code here …
QT8192                ; 8192 = SUM
…
QT6144                ; 6144 = 4096 + 2048 = TOFLOAT + nothing in R37
```

Priority: the different actions are performed in the order of the table, so CLEAR is the first and STOP the last.

For example, if you use TIMER CLEAR TOFLOAT you will always get zero as CLEAR is performed before.

If you want to get the current value and then clear, you must provide two instructions:
TIMER TOFLOAT
TIMER CLEAR.

TEXT : 40 or 80 columns, more lines and colors!

The TI-99 has the ability to display 40 columns, but, as the **F18A** brings new features (80 columns, 30 lines, color in TEXT modes), this is discussed here.

&W ***TEXT32 TEXT40 TEXT80*** set text mode

Syntax: &Wm or ***TEXTnn [options L30 XC]***

Set the number of text rows and columns according to m:

TEXT mode	F18A	MLC	PreCompiler	Screen adr/size	Color table adr/size
30x24		&W0	TEXT32	>00, 768 bytes	>800 , 32 bytes
40x24		&W1	TEXT40	>1000, 960 bytes	>800 , unused (32B)
80x24	*	&W2	TEXT80	>1000, 1920 bytes	>800 , unused (32B)
32x30	*	&W7	TEXT32 L30	>1000, 960 bytes	>800 , 32 bytes
40x30	*	&W5	TEXT40 L30	>1000, 1200 bytes	>800 , unused (32B)
80x30	*	&W6	TEXT80 L30	>1000, 2400 bytes	>800 , unused (32B)
30x24 ext colors	*	&W8	TEXT32 XC	>00, 768 bytes	>1000, 256 bytes
40x24 ext colors	*	&W9	TEXT40 XC	>1000, 960 bytes	>00, 256 bytes
80x24 ext colors	*	&W10	TEXT80 XC	>1000, 1920 bytes	>00, 256 bytes
32x30 ext colors	*	&W15	TEXT32 L30 XC	>1000, 960 bytes	>00, 256 bytes
40x30 ext colors	*	&W13	TEXT40 L30 XC	>1000, 1200 bytes	>00, 256 bytes
80x30 ext colors	*	&W14	TEXT80 L30 XC	>1000, 2400 bytes	>00, 256 bytes

Unused color table: in those modes, only two colors are available: foreground and background. They are set with SCREEN (&S).

Extended colors: in those modes each character has its own attribute that you can set with ATTRIB (&A).

Note: in all modes, except &W0, you can't use the BASIC instructions PRINT, DISPLAY, CLEAR etc as they expect 32 columns, 24 lines and a screen image table at >0000.

Note: in all modes, except &W0, use of the **extended character zone** is limited as the address >1000 is common for characters and either the color table or the screen image. But, as the screen image table is moved, more characters in the standard zone are now available!

See ***XPATTERN*** (&X) for more information on this zone.

TEXT mode	Free chars in standard zone	Free chars in extended zone
30x24	112: 32 to 143	256: 0 to 255
40x24	208: 32 to 143 & 160 to 255	132: 120 to 255
80x24	208: 32 to 143 & 160 to 255	16: 240 to 255
32x30	208: 32 to 143 & 160 to 255	132: 120 to 255
40x30	208: 32 to 143 & 160 to 255	106: 150 to 255
80x30	208: 32 to 143 & 160 to 255	none
30x24 ext colors	112: 32 to 143	224: 32 to 255
40x24 ext colors	176: 32 to 143 & 192 to 255	132: 120 to 255
80x24 ext colors	176: 32 to 143 & 192 to 255	16: 240 to 255
32x30 ext colors	176: 32 to 143 & 192 to 255	132: 120 to 255
40x30 ext colors	176: 32 to 143 & 192 to 255	106: 150 to 255
80x30 ext colors	176: 32 to 143 & 192 to 255	none

&A ***ATTRIB*** set character attribute

Syntax: &Aca or ***ATTRIB c a***

In extended color mode, set the individual attribute for a character.
"c" is the character (0-255) and "a" the attribute, 8 bits as this: PXYTCCCC

`bit P:` if 1, this character has priority over sprites.
`bit X:` flip horizontally the pattern
`bit Y:` flip vertically the pattern.
`bit T:` transparency instead of foreground color
`CCCC :` color selection.

For each pixel of the pattern, its bit is added to the right of CCCC to determine the actual color used:

Example, if CCCC = 1110

- a pixel ON will have color 11101 = 29
- a pixel OFF will have color 11100 = 28.

Note on L30 flag (Line 30) and Sprites list.

Extended Basic uses a marker for the end of sprite list (in dummy sprite #29) with a row value of 210 outside the display. With L30 flag ON, this value is now 250 as there are more rows of pixels!

If an unexpected sprite appears with L30, just add: ***SLOCATE 29 250 1*** (#L29.250.1)
When back to 24 lines, use: ***SLOCATE 29 210 1*** (#L29.210.1).

Multicolor mode: 64x48 or 64x60 pixels in 16 colors

The TI99 has the ability to display 64x48 blocs with 16 independent colors. With the F18A installed, you can increase this up to 64x60.

&W	***MULTICOLOR***	set multicolour mode

Syntax: &Wm or ***MULTICOLOR [option L30]***

Set the display in multicolor mode according to m:

Display	**F18A**	**MLC**	**PreCompiler**	**Screen adr/size**	**Pattern adr/size (*)**
64 x 48		`&W16388`	`MULTICOLOR`	>0000, 768	>1000, 1536
64 x 60	*	`&W24576`	`MULTICOLOR L30`	>1800, 1024	>1000, 1920

Important note: the pattern table must be moved to >1000, this is done with XPATTERN 2 instruction (&X2).

- So the complete sequence for Multicolor is:

With MLC :	*With the PreCompiler :*
&X2	XPATTERN 2
&W16388	MULTICOLOR

- And to go back to standard display, you must use:

With MLC :	*With the PreCompiler :*
&X0	XPATTERN 0
&W0	TEXT32

If **you use the L30** option, **XREGENABLE** must be provided first and read the previous note about the Sprite list termination.

In these modes, each pixel can be access with x coordinate from 0 (left) to 63 (right) and y coordinate from 0 (up) to 47 or 59 (down).
The colors are from 0 to 15 (not 1-16 like in the Extended Basic).

&p	***MCPLOT***	plot a pixel in multicolor mode

Syntax: &pxyc or ***MCPLOT x y c***

Plot one pixel at position (x,y) with color c.

&l	***MCDRAWTO***	draw a line in multicolor mode

Syntax: &lxyc or ***MCDRAWTO x y c***

Draw a line from last plotted pixel to (x,y) in color c. Last pixel is the one from the last MCPLOT or MCDRAWTO instruction. So lines can be chained easily for a curve for example, or a box.

Debugging

One main routine can help you debugging your code. It allows you to display the content of one to 6 variables or array elements. The values are given in hexadecimal.

1 With MLC

The value displayed is always the content of the pseudo variable "[". So you have to fill it before either with:

=[v	for the content of variable v
TGti[	for the content of the array element t(i).

Another pseudo variable,], lets you specify which screen line will be used by the debugger giving the VDP starting address of the line:

C]	]=0 for line 1
=]32	]=32 for line 2
=]64	]=64 for line 3

And so on, the address is 32 x (Line -1).
By default, the address is zero.

Then using the code M, this runs the debugger routine.
The debugger routine requires the use of a 164 bytes buffer to save the content of the screen and of the characters it will redefine. So your program is preserved and will continue without any modification.
To specify the buffer address, you must use the third pseudo variable "\" in a TR statement:

TR\164	reserve 164 bytes for the debugging session.

If you do not provide this buffer, the "M" routine will remain without effect.

M	display content of the pseudo variable

Syntax: Mn

What the routine can do:

- Save the STATUS word to keep the conditional flags for jumps
- Save the debugger screen line
- Save the current definition of characters 96 to 111 **(*)**
- Display the content of the pseudo variable "[" in hexadecimal in position 0 to 5.
- Output a tone (as Input does)
- Wait for a key
- Restore the debugger line
- Restore the char definitions
- Restore the STATUS word

So this instruction shouldn't interfere with the current program.

(*) Note: *the characters are mapped to 192 – 207 by the Extended Basic. If you used* `&X2` *or* `&X3` *to move the character table, then they are characters 192 to 207 equally. Those characters are temporarily redefined to display the 16 hexadecimal digits from* `0` *to* `F`.

The "n" parameter allows you to select precisely the behaviour of the function. It is the sum of a series of flags that enable one operation each:

PreCompiler	MLC	Action taken
#SL	256	*Save the debugger line under hex values*
#SD	512	*Save char 192-207 definitions*
#RC	1024	*Redefine chars 192-207 as the 0-F hexadecimal digits*
#OB	2048	*Output a tone*
#WK	4096	*Wait for a key stroke and let your read values*
#RD	8192	*Restore char 192-207 definitions*
#RL	16384	*Restore saved line*
#ALL	35512	*All actions performed*
Position	0 to 5	*You can debug from 1 to 6 values on the same line*

Example of a debugging session:

```
TR\164          ; reserve zone

=[D             ; take value of variable D
&M35512         ; debug D with default values in position 0.
```

Example of multi-debugging session:

```
TR\164          ; reserve zone

=[A
&M1792          ; debug A in position 0, save line and defs, define chars
                ;  (position 0 +256 + 512 + 1024 = 1792 )
=[B
&M1             ; debug B in position 1, every other operation disabled
                ; ( position 1 + 0 = 1)
=[C
&M30722         ; debug C in position 2, beep and wait key, then restore line and
                ; previous char defs.
                ; ( position 2 + 2048 + 4096 + 8192 + 16384 = 30722 )
```

In this session, you'll see variable A, B and C displayed in hexadecimal and then the computer waits for a key. For the first variable, the definitions of characters and the line are saved. For the last one, the definitions and the line are restored.

2 With the PreCompiler

To provide a buffer for the debugging routine, insert this pseudo-instruction:

```
DEBUG_BUFFER
```

If you want to use another line than de default first screen line, use:

```
DEBUG_LINE n    ; n from 1 to 24
```

To display the content of a simple variable use:

```
DEBUG pos #ALL v     ; pos from 0 to 5
```

To display the content of an array element, use:

```
DEBUG pos #ALL t(i)
```

Read carefully the section "**1 With MLC**" for details on the debugging routine. Only one `DEBUG_BUFFER` is necessary in the program, but it must precede the `DEBUG` call.

Note: *Internally, the PreCompiler manages the pseudo variables `[, ]` and `\`, so you don't have to deal with them.*

If you want to select precisely the actions performed, you can use the options in the table on the previous page. If you turn an option `#XX` to `#-XX`, then you will disable it instead of enabling it (Example: `#ALL #-OB` means every action except output beep).

Example of a multi debugging session:

```
DEBUG_BUFFER                    ; reserve buffer
DEBUG 0 #SL #SC #RC A           ; save defs, line, redefine chars and
                                ; display A in position 0
DEBUG 1 B                       ; display B in position 1
DEBUG 2 #RL #RC #OB #WK C       ; display C in position 2, output beep,
                                ; wait key
                                ; then restore line and char defs.
```

So you can watch variables continuously (disabling key stroke) or stop when you want to. Is speed is important, then selecting exactly the actions you need allows you to let your program behave as normal.

Note: When MLC is loaded, the three pseudo variables contain **the date of the current MLC version**:

[is the day (0-31)
\ is the month (0-12)
] is the year (2019…)

4° The Precompiler unique features

Using the Precompiler gives you more flexibility and more power than the MLC alone. Pseudo-instructions have been added. They are not a synonym of a single MLC code, but they use a construction of several codes.

This simplifies your work and makes the source code clearer.

Another unique feature is the Inline Assembler to bypass the limitation of MLC.

Let's start with...

The Pseudo-Instructions

Unlike the SELECT/CASE structure integrated to MLC, the following can be nested. Internally, those constructions use labels. So, you MUST leave the upper case labels free (from Label A to Label Z) and you can use, for example, lower case or numeric labels.

NDO ... NLOOP

Syntax:

NDO v n

..instructions..

NLOOP

Initializes variable V with value n and execute the block of instructions for V equals n to 1. A total of n iterations.

Ex:

```
NDO J 7
     PUTCHAR (1,J) $*
NLOOP
```

This loop writes on the screen seven "*" on the first line.

N0DO ... N0LOOP

Syntax:

N0DO v n

..instructions..

N0LOOP

Initializes variable V with value n and execute the block of instructions for V equals n to zero. A total of n+1 iterations.

Ex:

```
DIMTABLE A 10       ; reserve 10 words for array A
CLEAR B             ; B = 0
N0DO J 9
     PUTTABLE A(J) B
N0LOOP
```

This loop fills all elements A(9) to A(0) with the value of B, i.e. zero.

N-1DO ... N-1LOOP

Syntax:
N-1DO v n
..instructions..
N-1LOOP

Initializes variable V with value n-1 and execute the block of instructions for V equals n-1 to zero. A total of n iterations.

Ex:

```
DIMTABLE A 10        ; reserve 10 words for array A
CLEAR B              ; B = 0
N-1DO J 10           ; loop from 9 to 0
     PUTTABLE A(J) B     ; clear table
N-1LOOP
```

This loop fills all elements A(9) to A(0) with the value of B, i.e. zero.

FOR ... NEXT

Syntax:
FOR v n m
..instructions..
NEXT

Initializes variable V with value n and execute the block of instructions for V equals n to m. A total of (m-n+1) iterations.
Note: you must have m >= n.

Ex:

```
FOR R 1 24                ; row from 1 to 24
     FOR C 1 32           ; column from 1 to 32
          PUTCHAR (R,C) $Z    ; write "Z"
     NEXT
NEXT
```

This loop fills the whole screen with character "Z".

REPEAT ... UNTILtest

Syntax:
REPEAT
..instructions..
UNTILtest

The bloc of instructions is executed until the test is positive. The six tests are allowed:
UNTIL=, UNTIL<>, UNTIL>, UNTIL<, UNTIL>=, UNTIL<=.

Ex:

```
SAY &wHELLO          ; say one word
REPEAT
     TALKING?        ; talking is finished?
UNTIL<>              ; if =, no, so wait
SAY &wYOU            ; if <>, say next word
```

The word HELLO is said, then the loop waits until the computer is not talking anymore, then it says YOU.

DO ... WHILEtest ... LOOP

Syntax:

DO

..instructions 1..

WHILEtest

..instructions 2 ..

LOOP

Will execute the two blocs of instructions between DO and LOOP while the test is positive. Note that "instructions 1" is always executed at least once.

The six tests are allowed:

WHILE=, WHILE <>, WHILE >, WHILE <, WHILE >=, WHILE <=.

Ex:

```
LET A &hA000         ; start of high memory
LET J 0              ; index in A
DO
     GETTABLE A(J) B     ; read one word from memory
WHILE=               ; if = zero, go on searching
     INC J           ; next index
LOOP                 ; exit loop if word <>0 found
```

IFtest ... ENDIF

Syntax:

IFtest

..instructions ..

ENDIF

Execute the bloc of instructions only if the test is positive.

The six tests are allowed:

IF=, IF <>, IF >, IF <, IF >=, IF <=.

Ex:

```
LABEL x
     SLOCATE 1 (R,C)      ; locate sprite
     INC C                ; next column
     COMPARE C 256        ; limit is 256
     IF>
          LET C 1         ; if limit reached, return to 1
     ENDIF
GOTO x                    ; again
```

IFtest ... ELSE ... ENDIF

Syntax:

IFtest

.. instructions 1 ..

ELSE

.. instructions 2 ..

ENDIF

Execute the bloc of "instructions 1" if the test is positive and "instructions 2" if the test is negative.

The six tests are allowed:

IF=, IF <>, IF >, IF <, IF >=, IF <=.

Ex:

```
KEYWAIT 0                 ; suppose we wait for ENTER
COMPARE K 13              ; is it ENTER (code 13)
IF=
     SAY &wTHAT_IS_RIGHT ; if so, congratulations
ELSE
     SAY &wTHAT_IS_INCORRECT  ; else, too bad.
ENDIF
```

BYTES WORDS
STARTDATA ... ENDDATA v

Pseudo-instructions explained in section "**Arrays**" and particularly in :
"**Note on data initialization**" / "**b- With the PreCompiler**".

Note: BYTES and WORDS use internally the ***HEX*** `(TH)` code

FLAGS

Syntax: ***FLAGS*** ...key words...

The keywords are: ***INTERNAL, DISPLAY, SEQUENTIAL, RELATIVE, INPUT, OUTPUT, APPEND, UPDATE, FIXED, VARIABLE.***

The instruction is equivalent to ***BYTE*** and builds the value according to the key words. Use it into the PAB structure. See **File Access**.

The Inline Assembler

Let's enter a deeper world: the assembler! If you work with MLC and the PreCompiler, you'll surely find some limitations in what can be done. Sometimes in terms of speed or sometimes in terms of ability.

Why those limits? The main answer is that MLC uses the low ram expansion that is limited to 8KB and I couldn't integrate everything in it. The other answer is that I can't predict everything that programmers can imagine!

MLC and Assembler dialog

The Assembler and MLC can exchange data through the 26 variables from A to Z. They are predefined in the assembler, so you can write:

MOV @A,R9

This copies the value of variable A into register R9.

```
LI R0,K    ; r0 points to the K variable
CLR *R0+   ; clear K
CLR *R0+   ; clear L
CLR *R0    ; clear M
```

You can access the two standard Floating Point registers that I named r0 and r1 in MLC. They correspond to FAC and ARG in Assembler.

```
LI R0,FAC        ; points to r0=FAC
LI R1,ARG        ; points to r1=ARG
LI R2,4          ; four words to treat
MOV *R0,R3       ; exchange FAC and ARG
MOV *R1,*R0+
MOV R3,*R1+
DEC R2           ; decrement counter
JNE -5           ; if not ended, jump back 5 words (jne included)
```

$[... $]	enter and exit assembler

Those instructions must appear on a single line, as every PreCompiler instruction, so you can't write :

```
$[  INCT @D  $]
```

But instead, you must use this:

```
$[
     INCT @D
$]
```

DATA BYTE EVEN

DATA is the assembler equivalent for ***WORDS*** in the PreCompiler:

Syntax: `DATA w1, w2, w3, ....`

BYTE is the assembler equivalent for ***BYTES*** in the PreCompiler:

Syntax: `BYTE b1, b2, b3, ...`

EVEN is equivalent to EVEN in the PreCompiler and is used to ensure that the compilation pointer is even.

Syntax: `EVEN`

LABEL GOTO

They are equivalent to LABEL and GOTO in the PreCompiler. They share the same label space, so take care not to duplicate a label!

You can GOTO from the Assembler to a label outside.

You can GOTO from outside to a label in the Assembler bloc.

Note: *you can't use a Label with the B, BL, or BLWP instructions, GOTO is the only way (and is equivalent to B).*

Note: *you can't use a standard label with the JMP and conditional jumps instructions. For those, see "**The JMP problem**".*

Note on the addressing modes

Six addressing modes are available:

`Rn`	register	refer to a workspace register 0 <= n <= 15
`*Rn`	indirect	refer to the value pointed by a register
`@address`	address	refer to the value pointed by the address
`@adr(Rn)`	adr indexed	refer to the value pointed by address + Rn
`*Rn+`	ind auto inc	same as *Rn but then Rn incremented
`constant`	immediate	the value itself

the addresses or constants can be:

- A decimal value from -32768 to 32767, or 0 to 65535 for words
- A decimal value from -128 to 127 or 0 to 255 for bytes
- An hexadecimal value with ">" prefix such as >8300
- A variable from A to Z
- One of the standard references of the Assembler listed here:

GPLLNK, FAC, ARG, VDPWA, VDPWD, VDPRD, VDPSTA, GPLWS, PAD
SOUND, SPCHRD, SPCHWT, GRMRD, GRMRD, GRMWD, GRMWA

SCAN, NUMASG, NUMREF, STRASG, STRREF, XMLLNK, KSCAN
VSBW, VMBW, VSBR, VMBR, VWTR, ERR, FADD, FSUB
FMUL, FDIV, SADD, SSUB, SMUL, SDIV, CSN, CFI
FCOMP, NEXT, COMPCT, GETSTR, MEMCHK, CNS, VPUSH
VPOP, ASSGNV, CIF, SCROLL, VGWITE, GVWITE, DSRLNK

Addressing mode usage

The arguments passed to an assembly instruction can be of three types:

I. `REG`: A register with the only addressing mode `Rn`
II. `CST`: A constant
III. `GEN`: A general argument that can be `Rn, *Rn, @address, @adr(Rn) or *Rn+`

Instructions OPCODE Gen,Gen

Those instructions require two `GEN`eral arguments. The first is the source and the second the destination where the result will be stored.

`A`	*Add*	`AB`	*Add bytes*	`S`	*Sub*
`SB`	*Sub bytes*	`C`	*Compare*	`CB`	*Compare bytes*
`MOV`	*Move*	`MOVB`	*Move byte*	`SOC`	*Set one corresponding*
`SOCB`	*Soc byte*	`SZC`	*Set zero corresponding*	`SZCB`	*Szc byte*

Ex: `MOV @>2004,R2` move in R2 the content of word at address >2004.

Instructions OPCODE Gen,Reg

Those instructions take the source in a `GEN`eral argument and the destination is a `REG`ister.

`COC`	*Compare ones corresponding*	`CZC`	*Compare zeros corresponding*	`XOR`	*Exclusive or*
`MPY`	*Multiply*	`DIV`	*Divide*		

Ex: `XOR *R4+,R5` take word pointed to by R4, performs an exclusive OR in R5 and then increment R4 by two (words are concerned).

Instructions OPCODE Reg,Cst or OPCODE Reg,R0

Those instructions move the bits of a word located in a `REG`ister. The shift count is either a non zero `CONST`ant or the content of `REG`ister 0.

`SLA`	*Shift left arithmetic*	`SRA`	*Shift right arithmetic*
`SRL`	*Shift right logical*	`SRC`	*Shift right circular*

Ex: `SRA R5,7` shift R5 seven positions to the right keeping its sign.

Instructions OPCODE Reg,Cst

Those instruction use a ConSTant as first argument to perform an operation into a REGister.

AI	*Add immediate*	ANDI	*And immediate*	LI	*Load immediate*
CI	*Compare immediate*	ORI	*Or immediate*		

Ex: `LI R1,FAC` load into R1 the address of FAC
`LI R2,D` load into R2 the address of variable D

Instructions OPCODE Reg

Those instructions store a value into a REGister.

STST	*Store status*	STWP	*Store workspace pointer*

Ex: `STST R6` store status register in R6

Instructions OPCODE Gen

Those instructions operate on one GENeral argument to get the value or/and store the result.

ABS	*Absolute*	CLR	*Clear*	DEC	*Decrement*
DECT	*Decrement by 2*	INC	*Increment*	INCT	*Increment by 2*
INV	*Inverse (not)*	NEG	*Negate*	SETO	*Set to one*
B	*Branch*	BL	*Branch and link*	BLWP	*Branch and link workspace pt*
SWPB	*Swap bytes*	X	*execute*	CALL	*Call subroutine F18A*
PUSH	*Push F18A*	POP	*Pop F18A*	SLC	*Shift left circular F18A*

Ex: `SETO @>8300` fills word at address >8300 with >FFFF
`BLWP @VMBR` jump into a system routine for VDP multi byte read

Instructions OPCODE Cst

Those instructions use a ConSTant to modify a CPU register.

LIMI	*Load interrupt mask immediate*	LWPI	*Load workspace pointer immediate*

Ex: `LWPI >5000` will use 32 bytes starting at >5000 for R0 to R15.

Instructions OPCODE

No argument is required.

NOP	*No-operation*	RT	*Return (=B *R11)*	RTWP	*Return workspace pointer*
IDLE	*With F18A*	RET	*Return sub F18A*		

Instructions OPCODE Gen,Cst

Those instructions use a GENeral argument as source and a constant for destination.

LDCR	*Load CRU bit*	STCR	*Store CRU bit*	XOP	*Extended operation*

Ex: LDCR *R1,7 send seven bits from address pointed by R1 to the CRU, the CRU address is in R12.

Instructions OPCODE Cte

For the jump instructions the ConSTant is an offset in words, for the CRU instructions, the ConSTant is an offset in CRU memory added to the content of R12.

JEQ	*Jump if equal*	JGT	*Jump if greater than*	JH	*Jump if high*
JHE	*Jump if high or equal*	JL	*Jump if low*	JLE	*Jump if low or equal*
JLT	*Jump if less than*	JMP	*Jump*	JNC	*Jump if no carry*
JNE	*Jump if not equal*	JNO	*Jump if no overflow*	JOC	*Jump on carry*
JOP	*Jump if odd parity*				
SBO	*Set cru bit to one*	SBZ	*Set cru bit to zero*	TB	*Test bit*

The JMP problem

The jump instruction includes a relative offset in words where to jump, this offset is integrated into the OPCODE word in the less significant byte.
Two ways to do it.

Either you calculate the offset by yourself:

Ex:
```
JMP +0      jump to the next instruction
JMP -1          jump to the jump itself, infinite loop.
```

Here…
```
MOV *R1+,*R2+
DEC R4
JNE -3      to jump back "here", 3 words for 3 instructions including JNE
```

Note: *to jump backward, the jump instruction must be included in the word count, to jump forward, you don't have to include it.*

Note: *using @address or @addr(Rn) addressing modes adds one word in the instruction. If the two operands use this mode, then it adds two words to the instruction!*

Here…
```
MOV @FAC,*R2+
DEC R4
JNE -4      to jump back "here", 4 words as the MOV + FAC use 2 words
```

```
      JMP +2      to jump "there", two simple instructions
      NEG R3
      MOV R3,R5
There…

      Jmp +3          to jump there
      Mov @A,@B       3 words in a single instruction!
There…
```

Note: *using instructions ending with "I" such as ANDI, LIMI require an extra word to store the immediate operand.*

Either you use assembly labels:

They must end with a ":", look at this example:

```
My_Label0:
      MOV @FAC,*R2+
      DEC R4
      JNE My_Label0: ; jump back four words

      JMP My_Label1:    ; jump two words forward, two simple instructions
      NEG R3
      MOV R3,R5
My_Label1:

      Jmp My_Label2:    ; jump 3 words forward
      Mov @A,@B         ; 3 words in a single instruction!
My_Label2:
```

Note: *the ":" must be present both in the location of the label and in the call.*

5° Error codes:

0 No error

10 ":" expected because there are more ";" (return) than entry points.
There are more RETURN than SLABEL.

11 return expected, there are more ":" entry points than returns
There are more SLABEL than RETURN.

12 bad argument: a constant expected but variable found or vice-versa.

13 bad name, the name routine is not correct (null or more than 6 chars)

14 bad parameter: a variable or numeric value is expected

15 bad table name (must be from A to X in general and from A to H for TR reservation)

16 double definition, you try to define the same label twice

17 duration expected to end a block in SND section

18 F or V requiered before voice frequency or volume in SND section

19 missing staring "[" when you use "%" or "]".
CASE or ENDCASE without SELECT.

20 nested structures, you start a second "[" without closing the first one with "]"
A new SELECT found before you closed the previous with ENDSELECT.

21 Out of Range, constant mode in "K", "J" or Sound exceeds the known values
This can't appear with the PreCompiler.

22 unknown instruction code

23 unexpected end of line: a macro code must fit into one single string and not be cut in two parts
This shouldn't appear with the PreCompiler.

24 unexpected end of DATA lines (missing DATA "" ?)
You forgot the end of section $$ or the general end $END marker.

25 undefined label(s), you have jumped to a label that doesn't exist, or the "]" is missing to resolve every jump at the end of a "[" structure.
You jump to a label that doesn't exist or a structure is not closed (missing ENDSELECT, ENDIF, LOOP, etc.)

26 the line number in IO(1) passed to "COMPIL" doesn't exist.

27 unknown identifier: a section must start with P, S or C. Something else found.

28 sign expected after ? or ! in conditional jump (?=, ?>, ?<, !=, !<, !>)
This can't appear with the Precompiler.

29 trying to kill an unresolved label

6° Examples

PI Calculation, Floating point example

The program calculates PI with the sum $1/n^2$ equals to $PI^2/6$ with n infinite. The sum is computed both in MLC and in XB to show the difference.

Results: n=3000, MLC 28 seconds, XB 69 seconds.

```
100 CALL CLEAR
$MLC F 110 10 3000
1000 INPUT "N=":N
1005 INPUT "READY FOR MLC...":R$
1010 CALL LINK("PI",N,P)
1020 PRINT "PI=";P::PRINT
1030 INPUT "READY FOR XB...":R$
1040 S=1:: FOR I=N TO 2 STEP -1
1050 S=S+1/(I*I)::NEXT I
1060 P=SQR(6*S)::PRINT "PI=";P::PRINT
1070 GOTO 1000
$PI
     FARRAY 0
     FLOAT 1        ; send float 1 in r0
     FMOVE 0 3 ; save "1"
     FMOVE 0 2 ; sum initialized with "1"
     GETPARAM 1 N
     FLOAT N        ; r0=N
     DEC N
     NDO N N
          FMOVE 0 1 ; dup
          FMUL      ; N^2
          FMOVE 3 1 ; '1'
          FDIV      ; 1/N^2
          FMOVE 2 1 ; sum
          FADD      ; sum+1/N^2
          FMOVE 0 2 ; sum saved
          FLOAT N        ; new N
     NLOOP
     FLOAT 6        ; send float 6 in r0
     FMOVE 2 1 ; sum in r1
     FMUL
     FSQR      ; sqr(6*sum)=PI !!
     PUTFLOAT 2(0)  ; returns PI
$$
$END
```

Prime numbers, set management example

This programs looks for all the prime numbers from 2 to 32767 with the Eratosthenes' method.

Only 22 seconds are necessary to answer that there are 3512 prime numbers.

```
100 CALL CLEAR
;
; compiler inserted
;
$MLC F 110 10 3000
;
; call to my routine
;
310 CALL LINK("ERATHO",C)
;
; upon return, C=number of primes
;
320 PRINT C;" prime numbers"::PRINT "from 2 to 32767."
330 END
;
; the program
;
$ERATHO
    DIMSET 2 32767 A        ; A is a set with elements from 2 to 32767
    SETFILL A               ; all elements in A
    LET B 2                 ; first element to find
    DO
        SETFINDNEXT A B     ; look for an element in A starting from B
    WHILE=                  ; if found, then B=value, and B is PRIME
        LET C B             ; take the prime number
        DO
            ADD C B         ; computes 2*B, 3*B, 4*B, etc...
        WHILE>
            ELEMENT- A C    ; and remove this list from A
        LOOP
        INC B               ; see if there is another prime number after B
    LOOP
    SETCARD A C             ; computes cardinal of set A
    PUTPARAM 1 C            ; and return the cardinal
$$
$END
```

Roman numbers, string and table management.

This program returns the roman number form of an integer. It can return two different forms: the classic using standard rules and the compact one where you can save some signs (example VM = 1000 – 5 = 995 instead of CMXCV).

```
100 CALL CLEAR
$MLC F 110 10 3000
300 INPUT "A=":A
;
; last parameter 0 for COMPACT form
; or 1 for CLASSIC form with restrictive rules for subtraction
;
310 CALL LINK("ROMAN",A,COM$,0)::CALL LINK("ROMAN",A,CLA$,1)
;
; if compact and classic are equal, write one answer
;
320 IF COM$=CLA$ THEN PRINT "  = ";COM$::GOTO 300
;
; else write the two forms
;
330 PRINT "  COMPACT = ";COM$::PRINT "  CLASSIC = ";CLA$::GOTO 300
$ROMAN
    STARTDATA
        WORDS 1000,-999,-995,-990,-950,900  ; create a word table with values of
        WORDS 500,-499,-495,-490,-450,400   ; usable symbols or combined symbols (by pair)
        WORDS 100,-99,-95,90                ; negative values for compact form only
        WORDS 50,-49,-45,40
        WORDS 10,9,5,4,1,0
    ENDDATA A           ; this table pointed by A (cause A, B, C, D are words table pointers)
    STARTDATA
        STRING "M IMVMXMLMCM"           ; create a long string with symbols alone or by pair
        STRING "D IDVDXDLDCD"
        STRING "C ICVCXC"
        STRING "L ILVLXL"
        STRING "X IXV IVI "
    ENDDATA E                       ; pointed by E (cause E, F, G, H are byte table pointers)
    DIMTABLE F 255                  ; the output string reserved here with 255 max chars
    GETPARAM 1 X                    ; get the value to be converted in X
    GETPARAM 3 T                    ; flag for classic/compact
    CLEAR K                         ; will point into output string (0 to 254)
    CLEAR I               ; points into A() (from 0 to 25, the last A(25) is null for marker)
    CLEAR J                         ; points into E() string (from 0 to 49)
    DO
        COMPARE T 0                 ; compact form?
        IF=                         ; YES
            GETTABLE A(I) Y         ; one value from table A() into Y
            ABS Y                   ; and accept >0 or <0, no restriction
        ELSE                        ; here classic form
            DO
                GETTABLE A(I) Y     ; read a value
            WHILE<                  ; if negative, rejected
                INC I               ; next
                ADD J 2
            LOOP
        ENDIF
    WHILE<>
        DO
            COMPARE X Y             ; does Y fit in X?
        WHILE>=
            SUB X Y                 ; if YES, remove Y from X
            GETTABLE E(J) C         ; get one symbol
            PUTTABLE F(K) C         ; and put it into output string
            INC J                   ; see next
            INC K                   ; idem
            GETTABLE E(J) C         ; get next symbol
            COMPARE C 32            ; is it a SPACE? (32)
            IF<>
                PUTTABLE F(K) C     ; if not, it's a pair, add second symbol
                INC K               ; one more char
            ENDIF
            DEC J                   ; back on 1st symbol if to be repeated
         LOOP
```

```
         INC I                          ; if Y doesn't fit in X anymore, next value in A()
         ADD J 2                        ; and next symbols in E()
     LOOP
     LET U 2                            ; second paramater in CALL LINK (U, V, W, X are string
pointers in CALL LINK)
     LET Z K                            ; get lenght of string in Z
     PUTTABLE U(0) F                    ; return Z chars from F table to string R$ in CALL
LINK
$$
$END
```

Talking Scroll, speech, sprites and graphics

This programs shows an big horizontal scroll with three bouncing sprites (letters MLC) and speech with a stupid sentence repeated.

```
100 CALL CLEAR::DIM A$(10),C$(8)

$MLC F 110 10 3000

290 GOSUB 2500
300 RESTORE 1000::I=1
310 READ A$(I):: IF A$(I)="*" THEN A$(I)="":GOTO 330
320 I=I+1::GOTO 310
330 CALL CLEAR::GOSUB 2000
340 CALL LINK("SCROL1",A$(),C$(),1000)
350 CALL CLEAR::END

; here the text to be scolled

1000 DATA "This is a Scroll made with MLC (My Little Compiler) "
1010 DATA "FULLY INTEGRATED INTO Extended BASIC and that SPEEDS "
1020 DATA "UP your programs!    "
1500 DATA "*"

;

2000 CALL MAGNIFY(4)
2010 CALL SPRITE(#1,132,9,50,50,1,0)
2030 CALL SPRITE(#2,136,12,50,82,1,0)
2040 CALL SPRITE(#3,140,6,50,100,1,0)
2050 RETURN

; here the definitions of the three sprites

2500 CALL CHAR(132,"0000381C1C1C16262223232171710000000000E1C1C1C2C2E0E464686870F0000")
2510 CALL CHAR(136,"000078303030303030303030317F0100000000000000000000000080800000000")
2520 CALL CHAR(140,"000007183020606060606030180F00000000C06020000000000000002040000000")
2530 CALL SCREEN(1)::FOR I=1 TO 7::CALL COLOR(I,4,1)::NEXT I
2540 RETURN



$SCROL1
      STARTDATA
        WORDS &wI &wAM &wA1 &wTEXAS_INSTRUMENTS
        WORDS &wCOMPUTER &wUHOH &wWHAT_WAS_THAT
        WORDS &wI &wDO &wNOT &wKNOW &wFOR
        WORD 0
      ENDDATA D
      CLEAR K           ; pointer in D
       DIMTABLE C 6   ; 6 words for sprite motions (x,y)
       FOR I 0 5
              RND
              DIV Z 64
              SUB Z 32
              PUTTABLE C(I) Z
       NEXT
       LET R 1               ; first string index for message
       LET U 1        ; string table is parameter "1"
       GETPARAM 3 W   ; wait loop
       IF=
              INC W   ; at least W=1 !
       ENDIF
```

```
        DIMTABLE E 256 ; enough for 8 lines
        BMOVEVTOC 448 256 E   ; load 8 lines (adress 480) from screen to my buffer (E)
        DIMTABLE F 8   ; to load one char definition
        LET B E
        INC B          ; B = E+1 for one shift
        LET G 1
        COLOR 16 1 16
        FOR I 1 32
               PUTCHAR 14 I 128
               PUTCHAR 24 I 128
               PUTCHAR 1 I 129
               PUTCHAR 12 I 129
        NEXT
        FOR I 2 11
               PUTCHAR I 1 129
               PUTCHAR I 32 129
        NEXT
        LET I 5
        LET J 3
        INTERRUPT
        REPEAT
               GETTABLE C(I) X
               DEC I
               GETTABLE C(I) Y
               SMOTION J X Y
               DEC J
               DEC I
        UNTIL<
        DO
               GETTABLE U(R) M       ; string address in M and len in Z compared to zero
        WHILE<>                      ; while len not null
            DEC Z
            FOR I 0 Z
               GETTABLE M(I) S              ; one character from string
               ADD S 96
               MUL S 8               ; s= VDP ram address of char definition
               BMOVEVTOC S 8 F              ; take definition
               NDO N 8               ; 8 columns
                      BMOVE B 255 E         ; shift one position to the left
                      LET P 31       ; fist position in table
                      FOR J 0 7
                             INTERRUPT
                             GETTABLE F(J) A       ; one def byte
                             COMPARE A 128
                             IF>=
                                    SUB A 128
                                    LET Q 138      ; "*"
                             ELSE
                                    LET Q 128      ; " "
                             ENDIF
                             ADD A A ; shift A for next bit
                             PUTTABLE F(J) A ; and stores it
                             PUTTABLE E(P) Q       ; correct char in table
                             ADD P 32       ; next line
                      NEXT
                      NEG G
                      IF<
                             COLOR 16 1 16
                      ELSE
                             COLOR 16 16 1
                      ENDIF
                      DEFCHAR 2(N) 42
                      BMOVECTOV E 256 448   ; copy to screen !!
                      GOSUB c               ; sprite managment
                      NDO J W
                      NLOOP                 ; wait loop
               NLOOP
            NEXT
            INC R             ; next string
        LOOP
        SMAX 0                ; stop all sprites
        GOTO x                ; end

; this is the subroutine to verify the positions of the sprites and
; make them bounce

SLABEL c
```

```
      TALKING?
      IF<>
            GETTABLE D(K) H           ; next word to say
            IF=
                CLEAR K
                GETTABLE D(K) H
            ENDIF
            INC K
            SAY H
      ENDIF
       LET L 5                        ; index into C() table
       NDO H 3                        ; for 3 sprites
              LET Q 0                  ; default is no change
              GETTABLE C(L) T          ; current vertical motion
              SPOSITION H X Y          ; potition of sprite H (3, 2 or 1)
              LIMIT 8 60 X             ; vertical position into limits?
              IF<>
                     NEG T              ; if not, change vertical speed
                     PUTTABLE C(L) T    ; and store new motion
                     INC Q              ; mark a change
              ENDIF
              DEC L
              GETTABLE C(L) V          ; current horizontal motion
              LIMIT 8 216 Y            ; horizontal position into limits?
              IF<>
                     NEG V              ; if not, change horizontal motion
                     PUTTABLE C(L) V    ; stores it
                     INC Q              ; mark a change
              ENDIF
              DEC L
              ABS Q                    ; just to test Q
              IF<>
                     SLOCATE H X Y      ; if a change, new position
                     SMOTION H T V      ; and new motion
              ENDIF
              INTERRUPT                ; to enable auto motion
       NLOOP
RETURN
LABEL x
$$
$CHAR
        ; the 8 chars are the growing points that form letters in the scroll

        0000001818000000 0000183C3C180000 003C7E7E7E7E3C00 7EFFFFFFFFFFFF7E
        7EFFFFFFFFFFFF7E 003C7E7E7E7E3C00 0000183C3C180000 0000001818000000

        ; those two chars to surround the text and the bouncing MLC

        S128.0F1E3C78F0783C1EE7E7E7FFFFE7E7E7
$$
$END
; A = one def byte
; B = adress E+1 for scroll
; C = table for sprite motions
; D = table for speech sentence
; E = CPU RAM buffer for 8 lines (bytes)
; F = CPU RAM buffer for one char def (bytes)
; G = flag +/-1 to swap colors of char 128
; H =
; I = character loop from 0 to len(string)-1
; J = loop for 8 definitions bytes when creating one column
; K = pointer in speech table D
; L =
; M = VDP RAM address of string (bytes)
; N = loop for 8 columns per character
; O =
; P = pointer into E buffer for the 8 new characters when creating a column
; Q = char "*" or " " / flag in subroutine
; R = string index
; S = VDP RAM address of one char definition
; T =
; U = 1 to access the first parameter in CALL LINK as a string array
; V =
; W = wait loop
; X =
; Y =
; Z = len of string
```

PONG game, sprites, graphics and sounds

This is the PONG game that you all (should) know.

```
100 CALL INIT::CALL CLEAR::DIM S$(3)
; load compiler and compiles game and sounds

110 GOSUB 1000

; ball, paddle, net and field definitions

140 CALL CHAR(96,"60F0F0F0F0F0F060")::CALL CHAR(100,"60F0F06000000000")
150 CALL CHAR(97,"8855225588552255")::CALL CHAR(98,"FFFFFFFFFFFFFFFF")::CALL
CHAR(99,"0000000000000000")

; prepares screen

160 CALL CLEAR::CALL SCREEN(1)::CALL COLOR(9,16,10)
170 FOR I=1 TO 8::CALL COLOR(I,4,1)::NEXT I
180 RESTORE 900::READ M$,N$,P$::READ SP(1),SP(2),SP(3),SP(4)
190 DISPLAY AT(1,13):"PONG"
200 DISPLAY AT(2,1):M$
210 FOR I=3 TO 19::DISPLAY AT(I,1):N$::NEXT I
220 DISPLAY AT(11,1):P$::DISPLAY AT(20,1):M$
230 DISPLAY AT(21,2):"LEFT (X-E)     RIGHT (I-M)"
240 SC(1)=0::SC(2)=0::START=1::SPEED=2::CALL MAGNIFY(2)

; display little menu and wait for SPACE to start

250 DISPLAY AT(24,1):"SPACE=START  S=SPEED  Q=QUIT"
260 DISPLAY AT(22,7):SC(1)::DISPLAY AT(22,21):SC(2)::GOSUB 400::IF K$<>" " THEN 250

; sprites 2 and 3 are the paddles, sprite 1 the ball

270 CALL SPRITE(#2,96,5,16,24)::CALL SPRITE(#3,96,14,136,224)
280 CALL SPRITE(#1,100,13,120*START-102,184*START-152)

; call assembly routine to play

290 CALL LINK("PLAY",S$(),START,SP(SPEED),WIN)

; upon return, WIN is the winner!

300 SC(WIN)=SC(WIN)+1:: START=3-START::GOTO 260

; quit game, the assembly routine is deleted from ram

310 CALL LINK("POP",A)::PRINT A
320 END

; menu key

400 CALL KEY(0,K,S)::K$=CHR$(ABS(K))::IF K$=" " THEN RETURN
410 IF K$="S" OR K$="s" THEN 420
415 IF K$="Q" OR K$="q" THEN 310 ELSE 400
420 DISPLAY AT(24,1):"SELECT SPEED FROM 1 TO 4:";SPEED
430 ACCEPT AT(24,27)SIZE(-1)BEEP:SPEED
440 RETURN

; field definition

900 DATA "bbbbbbbbbbbbbaabbbbbbbbbbbbbb"
910 DATA "bccccccbccccaaccccccbccccccb"
920 DATA "bccccccbbbbbbaabbbbbbbccccccb"

; speed table 1 to 4

930 DATA 10,20,35,50

; includes here the loader from line 1000 and DATA from line 2000
```

```
$MLC F 1000 10 2000
1900 RETURN

; sound definitions

$SND 1  ; ball touches paddle 1
       FA440VA0VN15D2 VA2D3 VA4D4 VA6D5 VA8D6,VA12D7 VA14D8 VA15D0
$$
$SND 2  ; ball touches paddle 2
       FA220VA0VN15D2 VA2D3 VA4D4 VA6D5 VA8D6,VA12D7 VA14D8 VA15D0
$$
$SND 3  ; ball touches border
       FN5VN8VA15D1 VN6D1 VN4D1 VN6D1 VN8D2,VN12D2 VN15D0
$$

; game routine

$PLAY
       GETPARAM 2 S                        ; S=start player (1/2)
       GETPARAM 3 H                        ; horizontal speed
       RND                                 ; random number in Z
       DIV Z H                             ; reminder (so Z<H)
       LET G Z                             ; vertical speed!
       LET M 1                             ; default player 1
       COMPARE S 1
       IF<>                                ; if start player is not 1
              NEG G                        ; modifies motion and M=2
              NEG H
              INC M
       ENDIF
       SMAX 1                              ; one sprite with auto motion
       SMOTION 1 G H                       ; ball starts !
       SOUND M                             ; with a paddle sound
       SPOSITION 2 A B                            ; get positions of both paddles
       SPOSITION 3 C D
       DO
              INTERRUPT                    ; enables interrupt for auto motion

              ; here player ONE

              KEY 1                 ; read keyboard left, key in K and COMPARE K 0 performed
              IF>=                         ; a key pressed!
                     IF=                   ; if equal 0, it is X
                            INC A          ; here "X"=down, A=A+1
                     ELSE
                            COMPARE K 5    ; is it "E"?
                            IF=
                                   DEC A   ; here "E"=up, A=A-1
                            ENDIF
                     ENDIF
                     LIMIT 16 136 A        ; ensure A is in the range
                     SLOCATE 2 A B         ; and set new paddle position
              ENDIF

              COMPARE K 18                 ; is it "Q"
       WHILE<>                             ; if so QUIT !

              GOSUB b                      ; manages ball movement

              ; here player TWO

              KEY 2                 ; read keyboard right, key in K and COMPARE K 0 performed
              IF>=                         ; a key pressed!
                     IF=                   ; if 0, it is "M"
                            INC C          ; if "M"=down, C=C+1
                     ELSE
                            COMPARE K 5    ; is it "I"?
                            IF=
                                   DEC C   ; if "I"=up, C=C+1
                            ENDIF
                     ENDIF
                     LIMIT 16 136 C        ; ensure C is in the range
                     SLOCATE 3 C D         ; new position
              ENDIF

              GOSUB b                      ; manages ball movement
```

```
        LOOP                                    ; and back to paddle one !!!

        GOTO x                                  ; here if K=18, "Q" key, QUIT

; Subroutine for ball movement

SLABEL b
        SPOSITION 1 E F                         ; get ball position
        LIMIT 16 144 E                  ; is the vertical position in the field?
        IF<>                            ; no, so modifications!
                NEG G                   ; reverse motion
                SMOTION 1 G H           ; reflexion
                SLOCATE 1 E F           ; new location
                SOUND 3                 ; and border sound
        ENDIF

        LIMIT 24 224 F                  ; is the horizontal position in the field?
        IF<>THEN x                      ; if not, game has ended!

        LIMIT 32 216 F                  ; else, are we far from the paddles?
        IF<>                            ; not so far, verify contact
                IF<                     ; if under 32 then work with paddle 1
                        LET G A         ; take vertical position of...
                        LET M 1         ; ...paddle 1
                ELSE
                        LET G C         ; else take vertical position of...
                        LEt M 2         ; ...paddle 2
                ENDIF
                SUB G E                 ; vertical distance G-E
                LIMIT -16 8 G           ; is it in -16,8 ?
                IF=                     ; yes so, contact!
                        ADD G 4         ; ball touches the paddle M
                        ADD G G         ; G=2*(vertical distance+4) new vertical speed
                        NEG G           ; reflexion
                        NEG H           ; idem
                        SMOTION 1 G H   ; new ball motion
                        SLOCATE 1 E F   ; new location
                        SOUND M         ; and sound for paddle contact
                ENDIF
        ENDIF
        RETURN                          ; back to players keys

; end of game

LABEL x
        SMAX 0                  ; end of game, stop every sprite
        LET R 1                 ; default winner
        COMPARE F 124
        IF<                     ; if position under 124, winner is 2
                INC R           ; R=2
        ENDIF
        PUTPARAM 4 R            ; return winner
$$
$END
```

CHAR ROLL, assembler example

This program makes "roll" every upper case letter from right to left. It uses some assembly instructions to simplify the work.

```
100 CALL CLEAR
$MLC F 110 10 3000
300 INPUT "NUMBER OF ROLLS : ":N
310 CALL LINK("ROLL",N)
320 GOTO 300
$ROLL
    DIMTABLE E 208          ; definitions for 26 letters (26*8)
    BMOVEVTOC 1288 208 E    ; copy defs from VDP ram (1288 = address for A)
    GETPARAM 1 N            ; number of rolls
    NDO N N
        NDO I 8             ; eight bits roll
            $[
            MOV @E,R0       ; R0 = base address E
            LI R1,104       ; R1 = count in words (208 bytes)
            ;
            ; ..."here"...
            ;
            MOV *R0,R2      ; take two bytes ABCDEFGH abcdefgh
            MOV R2,R3       ; a copy of two bytes
            SLA R2,1        ; R2 = BCDEFGHa bcdefgh0 shift left
            ANDI R2,>FEFE   ; R2 = BCDEFGH0 bcdefgh0
            ANDI R3,>8080   ; R3 = A0000000 a0000000
            SRL R3,7        ; R3 = 0000000A 0000000a
            SOC R2,R3       ; R3 = BCDEFGHA bcdefgha (SOC = OR)
            MOV R3,*R0+     ; store circular result
            DEC R1          ; decrement count
            JNE -12         ; not zero, so jump "here"
            $]
            BMOVECTOV E 208 1288    ; new rolled defs in vdp
        NLOOP
    NLOOP
$$
$END
```

GPU program, how to run the F18A coprocessor

This program show how to run the second processor located into the F18A. The GPU routine fills the screen with "A", then "B", until "Z". The MLC program just runs N times the GPU one.

```
$MLC F 100 10 3000
300 CALL CLEAR
310 INPUT "Number of loops:":N
320 CALL LINK("TEST",N)
330 GOTO 310

;
; here the MLC program that will run "n" times the GPU one
;

$TEST
    GETPARAM 1 N            ; number of loops
    GPURUN &h4000           ; runs GPU program that stops immediately
    NDO I N
        GPUWAKE             ; wakes up the GPU program
        REPEAT              ; little loop to wait for the GPU to finish
            GPUSTATE
        UNTIL=              ; state will be "=" when GPU has finished
    NLOOP
    KEYWAIT 0               ; wait for a key
$$

;
; here the GPU program in assembly code
; the compiler sends it to VDP location >4000, the new F18A area
; this example fills the screen with the capital letters from A to Z
;

$GPU
    idle                    ; stops the program and wait for WAKE to run
    li r0,26                ; 26 letters from A to Z
    li r1,>A1A1             ; start with double >A1 = "A" in XB

        ; letter loop
        clr r2              ; screen address
        li r3,384           ; 384 words = 768 bytes = 24*32 positions

            ; fill loop
            mov r1,*r2+     ; write two letters on screen
            dec r3          ; decrement counter
            jne -3          ; if not finished, jump to fill loop

        ai r1,>0101         ; next letter (>A2A2, >A3A3 etc...)
        dec r0              ; decrement counter
        jne -10             ; if not finished, jump to letter loop

    b @>4000                ; back to idle state
$$
$END
```

BML graphics (F18A only)

This program draws lines filling a square.
Note the use of `XREGENABLE` *as there is no $GPU section.*

```
$MLC F 100 10 3000
300 INPUT "How many runs : ":N
310 CALL LINK("TEST",N)
320 END
$TEST
    GETPARAM 1 N                ; how many rectangles said the user
    XREGENABLE                  ; access enabled to extended registers
    STARTDATA
        byte &hE3               ; BML def bloc, flags with palette 3
        byte &h40               ; then address (here 64*64 = 4096)
        bytes 96,64,64,64       ; then x,y,w,h
    ENDDATA E
    BMLSET 0 E                  ; set my BML without display
    BMLPLOT 0 0 0               ; upper corner
    BMLFILLRECT 64 64           ; clear all
    BMLSET 1 E                  ; and display the BML
    LET C 3                     ; color
    NDO I N
        LET Z 63
        FOR X 0 63            ; first group of lines up to down
            BMLPLOT X 0 C
            BMLDRAWTO Z 63 C
            DEC Z
        NEXT
        LET Z 62
        FOR Y 1 62
            BMLPLOT 63 Y C    ; second group from left to right
            BMLDRAWTO 0 Z C
            DEC Z
        NEXT
        DEC C                   ; previous color
        AND C 3                 ; limited to 0-3
    NLOOP
    PUTTABLE E(0) &h63          ; else, change flags to "BML disabled"
    BMLSET 1 E                  ; and set new parameters
$$
$END
```

7° Table of contents

Compare and Jump

L	set label
_	kill label
B	branch to label
S	branch to subroutine
:	sub label
;	sub end
C	compare
?=	jump if =
?>	jump if >
?<	jump if <
!=	jump if <>
!>	jump if <=
!<	jump if >=
(	check limits
E	end
[	select structure
%	case
]	endselect

Screen Instructions

&G	get character
&P	put character
&C	set color
&S	set screen color
&D	define character
&d	extended def char
&X	extended chars

Pattern discussion

Sprites

#A	set sprite pattern
#C	set sprite color
#L	set sprite location
#M	set sprite motion
#P	get sprite position
#&	change coordinates
#D	get sprites distance
#I	enable interrupts
#K	delete sprite
#<	max moving sprites

Logical instructions

HA	logical AND
HO	logical OR
HX	logical XOR
HN	logical NOT
HW	swap bytes
H<	shift left arithmetic
H>	shift right arithmetic
HL	shift right logical
HR	rotate right
HC	carry bit

Set instructions

FD	define set
F+	add element
F-	remove element
F?	test element
FZ	clear set
FO	fill set
FN	complement
FC	cardinal
FF	find next element
FU	set union
FI	set intersection
FX	set difference

Music, Sound, Speech

&M	play sound

Sound discussion

&T	say rom word
&?	is talking?
&U	say user word

Debugging

M	display a variable
[,],\	pseudo variables
#XX	debugging flags

DEBUG_BUFFER
DEBUG_LINE
DEBUG

File access

PAB discussion
Y file operation
Y0 open
Y1 close
Y2 read
Y3 write
Y4 restore
Y5 load
Y6 save
Y7 delete
Y8 scratch
Y9 status

F18A support

QV write VDP reg
QR run VDP prog
QW wake VDP prog
QS sleep VDP prog
Q? GPU state
QB set BML
QP plot a pixel
QD draw to
QF fill rectangle
QT manage timer
&w TEXT mode
&w Multicolor mode
&p plot in multicolor
&l draw in multicolor

4° Precompiler unique features

Pseudo instructions

NDO .. NLOOP
N0DO .. N0LOOP
N-1DO .. N-1LOOP
FOR .. NEXT
REPEAT .. UNTIL*test*
DO .. WHILE*test* .. LOOP
IF*test* .. ENDIF
IF*test* .. ELSE .. ENDIF
BYTES, WORDS
STARTDATA ... ENDDATA
FLAGS

Inline Assembler

$[.. $]
DATA BYTE EVEN
LABEL GOTO
Note on addressing modes
Instruction list
The JMP problem

5° Error Codes

6° Examples

PI calculation
Prime numbers
Roman numbers
Talking Scroll
Pong Game
Char Roll
GPU program (F18A)
BML graphics (F18A)

7° Table of contents