MLC II - Language Definition and Documentation

1) Definition of commands and structures in MLC II
1.1) The file structure of MLC II

File are started with a keyword, as known from MLC Version 1, in order to make it possible to recognize MLC files for the Compiler or an IDE. This keyword however is no longer #MLC but has changed to **#ML2** to identify the files with new syntax since these can no longer be compiled with the old compilers. Be aware that, as soon as the backwards compatibility feature is introduced into the MLC compilers they will support the old MLC Version 1 syntax. You should, however, be aware that you will not be able to mix MLC 1 and MLC II Syntax. The syntax model is defined by the starting keyword and can not be changed during program compilation or execution. Some obsolete MLC Version 1 commands will simply cause the compiler to exit with an error if compiler in MLC II mode.

As in MLC Version 1 the main program is included in the #FNCT Main function that is run at the start of the program by the virtual machine. The code is executed until the command #FEND is found which ultimately terminates the execution of the program as it indicates the end of then main loop.

From within the main programs all functions, that have before been defined by a #FNCT command can be called using the #FRUN command. This command can be issued in any place of the program.

This is the main file structure of an MLC II file. The detailed description of commands like the new #FOR and #LOOP commands that are used for the structuring of the program can be found below. A description of the DATA structures as well as the description of the MLC command set are found in chapters 2 and 3 of this document.

1.2) Functions in MLC II

Functions can, like in MLC Version, be defined anywhere in the source of the program except within another function or the main function. The syntax for a function definition is:

#FNCT <*NAME*> <*PARAMETER COUNT*>
    (one or several commands)
#FEND

While the general structure is known from MLC the command has been extended with a possibility to pass parameters to it.
This way the old #FNCT structure can be resembled by using the structure

#FNCT <*NAME*> 0
    (one or several commands)
#FEND

Basically you pass the parameters to a function by writing them after the functions name when calling it. If there are several arguments to be given to the function they are separated by a comma (,). Lets say you want to write a function that would enable you to count down from the defined number. Then you could write

#FNCT count 1

```
        #CSTR $TMP,%1
        #TEXT 1,1,$TMP
        #DRAW
        #WAIT 10
        #CLRS
        %1-
        #IIFF %1>-1
        #FGOB
#FEND
```

This function can be called using the following command:

#FNCT count 15

This will count down from 15.

Currently only the passing of numbers is allowed. Functions can return one integer value since their name, preceded by a % sign is a valid integer variable and can be used in all statements. Be, however, aware that passing the function return variable to itself will generate a compiler error.

1.3) New language structuring features

Another new thing to MLC are the commands #FORL and #LOOP which are used to make coding in MLC more structured and the programs easier to read. The #FORL command resembles a for loop. The command has only a starting term and no closing statement and is, in its structure leaning to a BASIC like syntax. Here is the code definition:

#FORL *<FROM>,<TO>,<STEP>,<FUNCTION NAME>*

The structure should look familiar for BASIC coders except that you do have a function name instead of a code block finished with #NEXT and no counter variable. The counter variable if instead given as an integer parameter to the function. The FROM and TO values need to be given. The STEP value needs to be given too and can not be avoided. However if you want as simple loop you . can just set it to 1. The FROM value must be smaller than the TO value and the STEP value has to be greater than nil. In all other cases the compiler reports an error and stops the compilation process. If the TO value can no longer be reached since the variable has already exceeded it the interpreter breaks the FOR loop and returns to the next command to execute it. The definition of a function name instead of a code block resembles the MLC1 type of how this could be achieved only that in MLC II the compiler takes a lot of standard coding work away from the coder enabling him to concentrate on the essentials. The function has to be defined, else an error is generated at compile time. Be aware that the function that is used for the MLCII for command MUST accept (exactly) one parameter because this is
the replacement for the variable generally used in for-loops. This parameter can, like all other parameters to functions not be changed by the function.

Sample:

Count from 15 down to 1:

```
#FNCT FLOOP 1
        #CSTR $TMP,16-%1
        #TEXT 1,1,$TMP,clblack
        #DRAW
        #WAIT 10
        #CLRS
#FEND
```

#FORL 1,15,1,FLOOP

The next new structure that is introduced with MLC 2 is the loop function.
It is similar to the BASIC function Repeat. Here is the code definition:

#LFCT (function name),(conditional statement)

The loop is started by the #LFNCT command that expects the arguments function name and conditional statement separated by a comma. The first argument gives the name of the function that should include the commands that should be executed repeatedly.

The last structure new to MLC2 is the while do – type function. This is also adopted to the old MLC1 style. It is quite similar to the #LFCT function except that the condition is checked BEFORE the execution of the loop. This means the loop is only executed if the statement is not true when the execution is at the start of the loop. The syntax is:

#WFCT (conditional statement),(function name)

Developers note:

This function type can be easily extended by reusing the standard MLC1 function type. The only difference is, that in this version at the end the conditional statement is checked and if it is false then the function is simply restarted. This way it should be pretty simply to implement this new function type.

1.4)The new mouse interface

This new way of control for the user is intended to make the creation for strategy games like Command&Conquer easier for the developers to write. Since such games would be impossible without a mouse these new functions have been introduced. There are several new commands involved in the handling of the mouse feature and these will be explained here in detail.

The first command each MLC2 program should execute if it wants to use the mouse feature is the command #MFNC this command needs the name of a function as parameter. The function give to this command must not include its parameter names but the function must provide 2 integer parameters and one char parameter in this order. The mouse buttons (control) are fixed by the programmer and may be different for each model. But there a problem arises. What if the program previously assigned a function to the mouse control keys. This would lead to a conflict and so the engine test the keyboard assignment table for these keys and if it finds them in use by another function it prompts for new assignments for this function (not for the mouse keys!). The buttons include an at least 4 direction control for the mouse and 2 buttons (resembling left click and right click).

The first two parameters of the mouse function are the 2 position values of the mouse cursor. The third parameter is a char that indicates which event triggered the mouse handler. There are several possibilities to trigger the mouse function and they have the following values. Alternatively they may also be accessed by their constant name:

mClick	=	0					
mOver	=	1					
mrclick	=	2					
mSL	=	3	(scroll left)	MSLU	=	7	(scroll left/up)
MSU	=	4	(scroll up)	MSRU	=	8	(scroll right/up)
MSR	=	5	(scroll right)	MSRD	=	9	(scroll right/down)
MSD	=	6	(scroll down)	MSLD	=	10	(scroll left/down)

Before each if these values might be send several actions must be taken. In order to enable the mouse at all you must first execute the #MSET command which enables the mouse cursor. But this only enables you to catch the two click events. In order to catch the mOver event that is triggered once the cursor passes over a certain area you first have to set this area with the #MOVR command. This command accepts 4 parameters which are the four corners of the triggering area. The same applies to the Scroll commands which are one of the most impressive features of MLC2. Before you can use them you must call the #MSCR command (without any parameters). Be aware that the scroll events have a higher priority than the MouseOver events. This means if the MouseOver area is on the border of the screen and the mouse cursor passes over this area then the scroll events will be passed to the mouse handler. This is quite tricky so watch out! Please be also aware that the MouseOver and the scroll handlers take up a lot of processor time of the VM so do only use them if there is no other way to achieve what you want to do.

Another command introduced is the #MCUR command which sets the moue cursor. Here quite strict rules apply. The only parameter of this command has to be a bitmap variable which contains the new mouse cursor image. The image MUST have the size 6x6 pixels. This size has to be enforced by the VM. This means, that if a bitmap with the wrong size is given the VM is interrupted with a runtime error.

Developers note:

Here are some suggestions on how the internal handling of the mouse in the VM should be done. I would suggest to hook a function to the keyboard interrupt upon the execution of the #MSET command. This interrupt should watch for the predefined 6 keys which control the mouse cursor. If one of these occurs then the appropriate actions have to be taken. In order to react accordingly the address in the code segment of the VM where the handler function is stored should be preserved at some place. Upon a click this function is called with the parameters specified in the text above. If a movement of the cursor occurs then the first thing the VM has to do is to check whether the Scroll event and the over event are enabled. If the scroll event is enabled then the next thing to check is whether the mouse is in a range from 5 pixels distance of the virtual screen (which is defined by the size of 128x64 pixels, counting starts at 0, outside parts are not reachable by programs but may be used to show statistics or setting (like caps look or so) of the VM if there are borders). If all of these requirements are fulfilled then the mouse handler is called again with the appropriate parameters. If the scroll event is not enabled or the mouse cursor is not within a scrolling zone, then the VM should skip to querying whether the mouseover event is enabled. If it is then the position of the cursor has again to be checked against the window in which the mouseover event should be captured. If the cursor is within this window then the mouse handler functions should again be called with the corresponding parameters. Please be aware that only one event can occur at a time. So if a button press event is processed the over and scroll events are not processed. Likewise if the scroll event has been processed then the handler is done and the check of the mouse over event should not be done.
This is actually enough information for you to implement this feature.

2) The DATA structures of MLC II

MLC II knows two kinds of DATA structures. The

2.1 Traditional DATA structures

The traditional data structures of MLC version 1 are completely reused in MLC but have been extended to a certain extend ;) There are:

Type	Prefix	Range	Notes	Size in byte
Integers	%	-32768..32767	-	2
Strings	$	up to 255 chars	can be joined with chars	256
Char	.	0 .. 255	can be joined to strings	1
Colours	%	0 .. 8	colour codes available.	2
Arrays	@	complex	must be defined	<2^15
Bitmaps	[	complex	must be initialized	6
Pointers	*	0-65535	unsigned integer	2
Sprites	^	complex	-	0

Integers: that are whole numbers from -32768 to 32767
Strings: are encapsulated by the " sign and may be up to 255 characters long.
single elements can only be addressed by pointers.
Char: Single characters that have a range from 0 to 255. They can be used in conjunction with pointers to alter single characters of a string.
Format is either *#value* or *'single-character'.*
Colours: colours are defined by their names (unlike MLC Version 1) *or* their number:

Name	=	Colour	=	Number	=	Inverted
clWhite	=	White	=	0	=	6
clNWite	=	Nearly white	=	1	=	5
clLGray	=	Light Gray	=	2	=	4
clGray	=	Gray	=	3	=	3
clDGray	=	Dark Gray	=	4	=	2
clNBlack	=	Nearly Black	=	5	=	1
clBlack	=	Black	=	6	=	0
clInvert	=	Invert	=	7	=	/
clTrans	=	Transparent	=	8	=	/

Developers note:
The range of the colourcodes must be enforced by the compiler values smaller than 0 or greater than 8 will result in an error!

Please note, that the colour-code for invert is no actual colour but only represents the inverse colour of the current pixel that is drawn at. There for this colour also has no inverse representative. The same is valid for Transparent pixels. Since the transparency simply means that nothing is drawn you can't inverse it. This is usefully for rectangles if you don't them to be filled or something like that. If you use the inverse colour for drawing commands like #LINE the compiler will simply ignore them and display a warning since nothing actually needs to be drawn if the whole line is transparent.

Arrays: An array is used to store a List of items. Due to the memory limitations of the used hardware the array is, in fact a list. It may contain up to 32768 items (in theory) of

type integer. Or up to 255 Items of type string. An array can contain nearly all possible types from the above table. The whole Object (Array) may never be bigger than 2^16 Bytes and may never have more than 2^15 elements since the addressing is impossible using integers else.. An array may also contain Sprites (see below) or Pointers, as well as Bitmaps. There may be up to 32768 Pointers in an array. The amount of possible elements in an array is defined by the possible address space of the interpreters data segment. This way all DATA is being limited to 2^16 Bytes if added together. (or by the calcs memory). Arrays need to be initialized with the #ARRY command (see below) and can not be resized once they are defined. Access to undefined arrays is not possible. The elements of arrays are addressed by giving the index in brackets (1) denotes the second item of the array since counting starts by 0. The prefix for an array is the @ sign but in order to define the type of the elements that are contained the array needs a second prefix behind the @ defining this type. An array of Integers looks the following way:

```
@$IntArray
```

If the second prefix is omitted the compiler generates an error when the #ARRY command is called!

Developers note:

The compiler should look for #ARRY statements and if it finds multiple ones for one name then it should print an error. Else it should include the array in the DATA segment. This technique allows the coders to define arrays anywhere in the source code and even allows them to access arrays that are defined much later. Local arrays are forbidden.

Bitmaps: Bitmaps do no longer exist like in MLC with a variable holding the bitmaps data. Instead they are now represented by pointers that point to an array of integers. Each integer can hold 4 pixels so that the minimum size for a bitmap is 2x2 pixels. The new thing for the bitmaps is, that they now are used similar to sprites. The length and with is defined using properties. The new Bitmap variable is composed of these properties:

width:integer	-	holding the width of the bitmap in pixels
height:integer	-	holding the height of the bitmap in pixels
ptr:pointer	-	points to the array that contains the pictures data

each of them needs to be assigned before the bitmap can be used. They are all defaulted with 0 and any operations with bitmaps where any property is 0 will lead the interpreter to a failure. All bitmap sizes are possible as long as they fit in the 2^16 byte of the DATA segment of the program.

Pointers: They are mainly used by sprites and bitmaps and can, in theory refer, to any variable in the calculators memory. They are treated like integers, so you can calculate with them and do nearly anything you like. They don't have any special type, only a special sign to show that they are unsigned but else they are the same as integers and can, as a type even be used as unsigned integers if you like. You simply create a pointer by assigning any variable to an integer and use for this action, instead of the = sign the ^= sign to show that the address is needed. This, of course, bears the risk that you assign wrong memory positions so be really careful when using pointers! Pointers are dereferenced the in a similar fashion using the .= operator. The difference is, that the variable that should contain the value at the memory address has to be left of the .= sign and the pointer has to be right.

Developers note:

please take care that now overflows occur and that no negative values are assigned to the unsigned integer type (e.g. decrease from 0 or

direct assignment. This should IN ANY CASE end the interpretation process since anything else leaves us in an unpredictable machine state! Please do also note that the pointers are always given relative to the start of the DATA segment of the bytecode. The interpreter should also check that the dereferencing is never done beyond the end of the DATA segment. The pointer variable can, however take all values the unsigned 16-Bit Integer type offers since it may also be used as unsigned integer variable instead of the pointer. The conversions of the signed and unsigned integer types are done by setting the 15th bit to 0 when converting in either direction.

Sprites: Sprites are similar to bitmaps but can be animated, rotated and moved across the screen automatically. For these actions they do have certain properties:

x:char - vertical position on screen 0-127
y:char - horizontal position on screen 0-63
dx:char - floating value (1/100 of position vertically) 0-99
dy:char-floating value (1/100 of horizontal position) 0-99
vx:char-movement on y-axis per call 0-99
vy:char-movement on x-axis per call 0-99
vmode:(disp,*1-254*,off)

- if char then time in char/10 second that a change should occur. This issues an automatic #DISP call every time this occurs. All events are synchronized and begin with the first call to disp. User calls to disp do re-initialize the loop (e.g. issue the next disp call and set the timer to the next event)
- if disp(=255) then this occurs only with a disp command called by the user this or another call from another sprite.
- if off (=0) then no automatic movement is used by the object.

Rot:0-3 - 0=0° / 1=90° / 2=180° / 3=270° rotation
rmode:(disp,1-254,off) -see the vmode information
bmp:bitmap/array -a bitmap or an array of bitmaps (overloaded)
ani:(disp,1-254,off) -see vmode shows the next bitmap of the array if on.

Sprites have to be initialized with the #SPRT <variable name> call before they can be used. This assigns them a unique ID for internal handling. Uninitialized Sprites will generate an error at compile time.

Developers note:

This command is one of the most difficult of the whole language. But it is also its strength. Sprites are in fact more commands than data types. The following internal handling is recommended. Add an internal variable to the properties as a unique identifier. Add this identifier as well as the update intervals (for each of the 3 update properties!) to a table. Then you write a timer interrupt that is called every 1/10 second (hardware timer) and implements a counter for each call of the interrupt. On every call to the timer interrupt you have to check the table if any ID matches the current update interval. This is done by calculating:

Mod (Counter/Interval)

If the result is 0 then you have to update the sprite and let the engine issue a #DISP command (that also updates all other sprites/properties set on DISP mode). The compiler should optimize the code as much as possible. If all 3 timed modes use the timer mode and the SAME time then the compiler should set 2 of them to DISP in order to reduce the size of the internal table and therefore shorten the runtime of the timer interrupt. It is recommended to turn of the timer during the recalculation and the redraw process to avoid writing a re-entrant

interrupt. This should only include a small lag. Please note that this type is overloaded and that the ani variable must only be interpreted if an array is assigned to the bmp property.

Keycodes: New Keycode variable are identified by a] sign. Please do not mistake them for bitmaps which have a quite similar type identifier. For more details on the keyboard system please look into the keyboard handling section at the end of this chapter. The keycodes may only be used in assignments but it is not possible to use them in calculations. This would generate a compiler error.

Maps: maps are 2 dimensional arrays that contain a certain amount of bitmaps (NOT sprites). They are only limited by the size of the memory (2^16 bytes) and by the integer address space (not more than 32768 elements in every dimension). In theory the largest square map possible is sqrt(2^16/6-4) = 104x104 bitmaps big but then you have no space left to define the bitmaps ;) The map type implements 2 properties

Appendix for 2.2)

Problem: The keyboards of all supported architectures and calculator brands are vastly different. The names of the key that are used during the programming therefor may differ from the actual names that are printed on them due to reorganisation because of better ergonomics or missing key on other calculators.
The problem is to find a layout or mecanism that is easy to implement, easy for the user to understand and for the programmer to implement. Further it would be best if all of the keys and/or key combinations which are offered by the different calculators would be supported and accessible. The only exception should be the MLC2 Menu key which will let the user enter into the setup menu of the Virtual Machine.

Solution: The only solution that comes to my mind and solves all of the above mentioned problems and fits all the requirements is the implementation of an abstraction layer and key variables. For this the folowing new commands, that will be explained below are, beside the new variable type *Key* introduced:

#AKEY <key variable>,<description>,<prompt>	-	Assign Key
#RKEY <key variable>,<description>,<prompt>	-	Reassign Key
#WKEY <key variable>	-	Wait for Key
#GKEY <key variable>	-	Get currently pressed key
#SKEY	-	Stop for Key

The concept of the new keyboard handlings is quite simple. The programmer is no longer confronted with the hardware architecture of the platform but can use the abstraction provided by the VM which takes care of the exact hardware handling. This is also in the sense of an platform independant language. Since the commands are also bytecode commands the exact handling is left to the programmer of the virtual machine.

The key to the new concept is the internally kept KAT - the Key Assignment Table. There are 3 ways for the programmer to implement this concept. Each of the 3 are compatible (e.g. the pogram does not depend on which method the VM programmer has implemented.

1) *Statically*: This means that. whenever a game executes the #AKEY command for a description it looks up in a table that keeps the values of all the already assigned keys. If it does not find the specified description it prompts the user to set a key for the action that is described by the descrition string. After the key is set the VM test wether the key is already used by another descritiption and if it is the user is warned that this will overwrite the old description. Whenever the same descritiption is asked for again the keycode from the table is returned. The table

is stored for each program in a file with the name of the program and the headder string "KAT". It is saved upon program termination and loaded again when satarting the new program.
The size is changed dynamically.

2) *Simple*: Like the statical version, but the table is not saved and restored. Instead the assignments are only remembered during the time of execution but forgotten afterwards. So you have to reassign the keys each time the app is run. This mode is only recommended for initial versions and for bug testing.

3) *Dynamically*: This is similar to the static version again but has a great advantage. Namely this, that the KAT is not stored in one file for each program but in one single file that holds all the descriptions and their corresponding key values. This has the great advantage, that the most common keys like "Left" or "Down" or "Fire" don't have to be assigned for each program over and over again. Instead they all read from 1 global table. In order to limit memory usage the VM programmer should include a usage counter which increments on each AKEY assignment of the variable. The size of the table is up to the VM programmer. Once it is reached new asignments overwrite the old, seldom used assignments. This should be pretty handy

I would strongly advise you to use the dynamic mode since it is the easiest mode for the user. However you can also give the user an option in the setup menu of the vm where you let him choose which model he wants to use.

The folowin rules are needed in order to make this feature a real success and easy to use:

--> Ignore cases in descriptions
--> White spaces are forbidden and the compiler should print an error and exit if it finds one in a description
--> Double assignments "steal" the key in this sense, that the description is replaced with the new one. The compiler should warn if such a stealing occurs.

The SKEY commands does, btw., wait for anykey and returns it in the variable given to it.

2.2 Advanced DATA structures

There have been some completely new data structures introduced to MLC.
Here is a descritption of them:

Dynamic and static variables:

Imagine the following scenario: you have 2 maps in your game that need to be changed dynamical (terrain destruction etc) during playing. There are only 2 possible ways to go with the standard data structures. Either each map is copied into the data segment of the program and changed there, leaving the original version untouched but wasting twice the amount of memory the map needs or you do only keep one copy in the code segment and modify it directly and do destroy the original this way during runtime. If the user wanted to redo the current map the whole game would have to be exited and recompiled what results in quite long load times. Imagine a 100x100 map that takes 10KB take 2 of them for the 2 maps we do have so you got 20KB and using the copy model you

would get 40KB of memory only filled with these 2 maps. On small models this can easily force the calculator to its knees. Another drawback that the compiler would have to decide whether the data should be changed or not. The new implementation in MLC II solves this problem by using all variables not specially declared with the copy mode (the memory wasting thing). However 4 new commands have been introduced:

#STAT, #DYNA, #FREE, #MEMO

that deal with the problem.
The first 3 can be set before any variable name and do the following:

#STAT -	means the variable is static. This way the compiler arranges the program in this way that only 1 copy is held in the DATA segment that can not be changed. All attempts to change the variable are detected during compile time and generate a 'Static violation error'.
#DYNA -	This defines a variable as dynamic. The data that is defined is copied to a special dynamic part in the DATA segment and the interpreter keeps track of its position.
#FREE -	This frees up the memory taken up by the variable destroying all changes that have been made in the dynamic data segment.

In order to make work more easy for the coders of the interpreter system the fourth command was designed to define the size of the dynamic DATA segment. This guarantees that no excessive data movements have to be made during runtime which would result in noticeable lag. On the downside this gives to MLC programmer a lot of responsibility since he has to watch that old objects (maps) are removed from the DynDATA section in order to free it up and make room for new objects. If the DynDATA section runs out of memory the interpreter will generate a 'DynDATA Error' and terminate execution. It is best to remove all possible maps from DynDATA memory previous to storing new big objects. Calling the #FREE command with the name of a variable that does not reside in the Dynamic memory will just continue execution with the next command without issuing an error or a warning.
Examples:

```
#STAT %PI=3

#MEMO 10
#DYNA $TMP
#CSTR $TMP,%counter
#TEXT 1,1,$TMP,clblack
#DISP
#FREE $TMP
```

File variables do replace the file access commands #WRTE and #READ that are known from MLC Version 1. They are no longer accessible with their old function in the new MLCII programming language. The new variable type is identified by its prefix : the variable only needs to be assigned values. There is no function call needed to create it. In fact the compiler should take care of calling the FileOpen or Close functions of the interpreter whenever the properties change. The following properties are associated with the File type:

RO	- 0/1	(synonym: false/true)

Open	- 0/1 (synonym: false/true)
Filename	- String[8] (fixed length / unterminted)
Size	- Integer (-->See #MEMX)

beware that the values Size and filename must be set before setting open to true. Else the interpreter will generate an exception and exit the program with an error message. If the file doesn't already exist the interpreter will take care of creating it. In any other case the file is just opened and made available under the new variable for access. If any of the parameters is changed while the files is open the interpreter will react accordingly. If the size is changed then the file will be resized to match the new value. However the interpreter will not check if data in the file will be lost upon decreasing its size. This is the responsibility of the programmer. When the filesize is set to 0 the file will be deleted and the file close (Open = 0).
When the read-only (RO) flag is changed all further write access will be denied and generate an exception. When the filename is changed the file will be renamed to match the new filename. The compiler should be aware of these things and call the FileUpdate function of the interpreter after every change to the parameters.
This method of handling files avoids the implementation of additional commands for resize/close/open/rename and delete.
However the using of filenames does need the implementation of the new function #EXIS (see below). Please note that the maximum file size is 16384 since the first 2 bits are used internally to indicate the RO/Open flags for the compiler. They are, however not stored in the file system.

Just on a side note:
Please be responsible with the tools given to you. Be aware that the program you write is not the only that needs memory on the call and so you should decide REALLY carefully weather a value needs to be stored or if it could be recalculated. You also should inform the users of the size of the files since, at least on the Casio calculators the files will not be visible in the standard explorer. (TOUCHE can view them however)

3) The Command set of MLC II

most commands of the MLC1 command set (of the TI calculators) have been ported to MLC2 however, some have been slightly changed, some heavily and some are completely new so the give you a complete overview here is the complete feature set of MLC2 with all its commands, development specifications, syntaxes and some examples. All commands are explained in depth in order to make programming for you as easy as possible. However, if you are completely new to programming also one of the tutorials to be published on http://dysfunction.earthforge.com might server your purpose better than this reference.
Commands are sorted in alphabetical order. To see whether these functions are already supported in the latest MLC2 versions on the various calcs I would recommend you to take a look at the release schedule and the manual with annotations which are kept current and up to date. To visit them please use the links at http://dysfunction.earthforge.com.

Each command is first noted in a standardized way. Here an example:

#TEST <_%cmd1>,<$string>

the parameters are always enclosed in brackets (< and >) and all parameters are always given

with their type. In the example the $string denotes, that the value has to be of the type string. Another way to give parameters is the notation with the underscore before the parameters name. In the example above the _%cmd1 means, that the parameter can either be given as an intermediate value or as a variable.
In the example this means that the following ways to call the function are valid:

```
%t1 = 3
#TEST %t1,$test1
```

or

```
#TEST 3,$test1
```

but not

```
#TEST 3,”Test string”
```

or

```
#TEST %t1,”Test string”
```

got it? It is easy isn't it? :D

All of the commands include examples with an information on the expected output. The commands affecting fonts or delays or similar technical details which are important to have specific information for in order to maintain platform compatibility. This means that, for example, with the fonts, an image for the exact style of the standard fonts is given. You may not include different / additional fonts in the specific versions of the VM in order to not mess with the programs and let the coders create incompatible code.

For the argument types please look into section 2 above where all of the variable types are explained in detail together with their valid ranges. The following list represents the command set in the final version of MLC. Since no platform has released a final version yet you should also look at http://dysfunction.earthforge.com/forum where detailed information of the progress and the available commands will be available.

All commands appear in alphabetical order. The examples do not include complete programs. They merely are the code inside a main function and should be encapsulated in a real program unless information stating otherwise are given.

#ABSL

Syntax: #ABSL <_%number>,<%absnumber>

This command returns the absolute value of the number given by _%number.
This means that the minus in front of the number is removed (if there was one, else the same number is returned again). The number returned by the command is given to the variable %absnumber. This function does only work with integers and integer variables.

Samples:

```
#ABSL -10,%abs          ;Returns 10 in %abs

#ABSL 5,%abs            ;Returns 5 in %abs

%number1=-10
#ABSL %number1,%abs     ;Returns 10 in %abs
```

```
%number1=5
#ABSL %number1,%abs          :Returns 5 in %abs

#ABSL -10,10                 ;Compiler error
```

#AKEY

Syntax: #AKEY <]keyvariable>,<_$descript>,<_$prompt>

This command is explained in the key variables section in detail. In short it does assign The key known to the VM by the name $descript to the variable]keyvariable. However, if this key is not known to the VM then it opens a pop-up with the message specified in $prompt and asks the user to input a key for this action described by $descript. This key code is remember and not asked for again on program restart.

The exact code is never known to the programmer and the key variables may neither be written to now read from. They may only be used in the appropriate key commands. Any other usage will generate a compiler error.

I would really recommend to always give the description in english since this greatly enhances compatibility of the programs and makes the KAT (described above) smalled. This also ensures that the user does not have to give the key names for “FIRE”,”FEU”,”FEUER” and other things.

In order to make this possible the KAT allows duplicate assignments of keys but the VM checks that NEVER is the same key assigned for different actions in the same program. This means that the VM note which keys have been requested by the VM and whenever the same key is requested again with a different description then the VM generates a compiler error. Be, however, aware that this is only implemented in order to provide other language coders an easy way to get into MLC2. It is generally considered as bad coding practice to use other languages than English in the descriptions. However, the variable names and the prompt text may be in every language you wish since they do not matter for the internal processing of the variables. The description is NOT case sensitive!!

The prompt parameter may also be left out. Then the standard text “Press Key for action $description” is outputted (where $description is replaced by the respective text).

Samples:

```
#AKEY ]fire,”FIRE”,”Please press button for fire”      ;OK
#AKEY ]feuer,”FIRE”,”Bitte Feuer Taste drücken”       ;OK
#AKEY ]feuer,”FEUER”,”Bitte Feuer Taste drücken”      ;BAAAD coding practice!Avoide!
#AKEY ]fire,”FIRE”                                    ;OK

$DESCRIPT=”FIRE”
AKEY ]fire,”DESCTRIPT                                 ;OK
```

#ARRY

Syntax: #ARRY <@arrayname>,<_%elements>
#ARRY <@$arrayname>,<_%elements>,<_%maxstring>

This command defines an new array with the number of elements defined by the %elements parameter. The name that has to be given as first parameter is the one the array is addressed with after the initialisation is done. The number of elements can not be changed during runtime. A special thing are string arrays. They are used in such a way

that there is a third parameter %maxstring which defines the maximum length of the strings. This parameter may not be omitted! If it is not given the compiler will generate an error and abort the compilation process. The same will happen if the third parameter is given on any other tyoe than string arrays. The sense of this is, to save memory in the programs data segment. Since I want to avoid dynamic data movement because of enlarging strings and also wanted to avoid wasting free space by setting the maximum string length as default size (255) I choose this way. So you can calculate the amount of memory needed by the array by calculating:

```
%elements*(%maxstring+3)
```

For all other data types you'll have to look up the specific size of the element in the table in the data types section and do the following calculation:

```
%elements*TypeSize
```

Please choose your memory layout carefully especially if you want to use dynamic memory (mentioned in the second chapter) since you may run into “out of dataspace” errors if you do waste your memory.

Samples:

```
#ARRY @%IntArray,10
@%IntArray[0] = 10              ;OK
@%IntArray[10]=3                ;Compiler error since counting starts @0

#ARRY @%IntArray,10,20          ;Error since the second parameter may not be
                                ;used with non-string arrays
#ARRY @$StrArray,10,10          ;OK
@$StrArray[0] = “Hallo Welt”    ;OK (exactly 10 chars)
@$StrArray[2] = “The first test” ;Error since not more than 10 chars allowed
@$StrArray[10]=”Test”           ;Error since out of bounds

#ARRY @$StrArray,10             ;Error since string length missing
```

#BITM

This command has become obsolete because of the new way bitmaps are handled. Bitmaps can simply be used by assigning the variable with values. If uninitialized bitmaps are given as parameters to command the interpreter will give an error.

#CINT

Syntax <%integerVar>,<$StringVar>

This command converts a string to an integer. Be aware that only the chars 0 – 9 are accepted for conversions and that numbers exceeding the maximum range of the integer type will generate a runtime error.

Please do also note that both parameters may NOT be immediate values. This would also make no sense because an immediate value could directly be assigned to the integer in the right way.

Samples:

```
$Str1=”1239”
#CINT %Int1,$Str1       ;OK %Int1 is now 1239
```

```
#CINT %Int1,"1239"      ;Error! Use %Int1=1239. Why not?

$Str1="Hallo1"
#CINT %Int1,$Str1       ;Error! $Str1 is no number.
```

#CSTR

Syntax <$StringVar>,<%integerVar>

This command is quite similar to the CINT command. It only works vice versa and converts an integer to a string. If you use a string array as target be aware that the number must fit into the maximum string length defined upon the creation of the array.

Please do also note that both parameters may NOT be immediate values. This would also make no sense because an immediate value could directly be assigned to the integer in the right way.

Samples:

```
%Int1 = 129
#CINT $Str1,%Int1       ;OK %Str1 is now "129"

#CINT $Str1,1239        ;Error! Use $Str1="129". Why not?

#ARRY @$Astr,10,2
%Int1=1239
#CINT @$Astr[0],%Int1 ;Error! Number does not fit!

#ARRY @$Astr,10,10
%Int1=1239
#CINT @$Astr[0],%Int1 ;OK.
```

#CLRS

Syntax: #CLRS

Clears the buffer of the screen (no the screen itself). The clearing of the screen is only done if afterwards an #DRAW command is issued. Be aware, that the screen is by no drawing command affected directly. The actions do always show when #DRAW is executed.

Sample:

```
#CLRS           ;Clear the buffer
#DRAWs                  ;And write it to the screen.
```

#CLSN
#CMPA
#DATA

This command has become obsolete since there is no more such thing like a data section. All data is now defined directly in the source code. This enhances the readability of the program.

#DBMP

Syntax: #DBMP <[bitmapname>,<_%xPos>,<_%yPos>

This command draws the bitmap with the name given as first parameter at the position %xPos,%yPos. The drawing is, as always only done in the buffer and not directly visible on the screen.

Samples:

//!TODO!

#DEND

Since this #DATA command has been removed this command has also become obsolet and is no longer recognized by the compiler. See the notes on #DATA for more details. Execution of obsolete commands leads to a compiler error!

#DISP

#DMAP

Syntax: #DMAP <@

#DRAW
#DYNA
#FEND
#FGOB
#FNCT
#FORL
#FREE
#FRUN
#GKEY
#HALT
#IIFF
#INPT
#LFCT
#LINE
#MCUR
#MEND
#MFNC
#MOVR
#MSCR
#MSET
#MWND
#NMAP
#PAUS
#PIXL
#PIXT
#POLR
#RCLP
#READ
#RECT
#RKEY
#RNDM
#RSET
#SHFT
#SIZE
#SKEY
#SPRT
#STAT
#STOP

#SUBS
#TCHK
#TEXT
#TILE
#WAIT
#WFCT
#WKEY
#WRTE

Changelog:

V 1.0

09.09.2005	-	First version has been published

V 1.0.1

Keyboard information have been added
Array information have been updated
Command list has been included
Information on mouse handling have been added.