MY-BASIC Quick Reference

1.   Introduction

MY-BASIC is a tiny cross-platform easy extendable BASIC interpreter written in pure C with about 5000 lines of source code. Its grammar is similar to structured BASIC in early era, but without line number. It is aimed to be either an embeddable scripting language or a standalone interpreter. The core is pretty light; all in a C source file and an associated header file. You can combine MY-BASIC with an existing C / C++ / Objective-C project easily, that can make them more powerful. This manual is a quick reference on how to program with MY-BASIC, what it can do and what cannot, how to use it and extend it as a scripting programming language.

For the latest info, see http://code.google.com/p/my-basic; or contact with author from mailto:hellotony521@gmail.com.

2.   Programming with BASIC

The well known programming language BASIC is an acronym for Beginner's All-purpose Symbolic Instruction Code; when we mention BASIC today, we often refer to the BASIC family, not a specific one. The BASIC family has a long history since an original BASIC was designed in 1964 by John George Kemeny and Thomas Eugene Kurtz

at Dartmouth College in New Hampshire; and BASIC is famous because it is easy to learn and use all the time.

MY-BASIC has an early structured BASIC like grammar, similar as GW-BASIC but without line number. It would be familiar to you if you ever had programmed with a BASIC dialect.

Getting started

You can download the latest MY-BASIC package from http://code.google.com/p/my-basic/downloads/list or check out the source from http://code.google.com/p/my-basic/source/checkout first if you don't have a standalone interpreter yet. It is recommended to check out the source code rather than downloading a package because the ones in the download list are often a little out of date.

In this part let's get started using the MY-BASIC command line interpreter which comes as below:
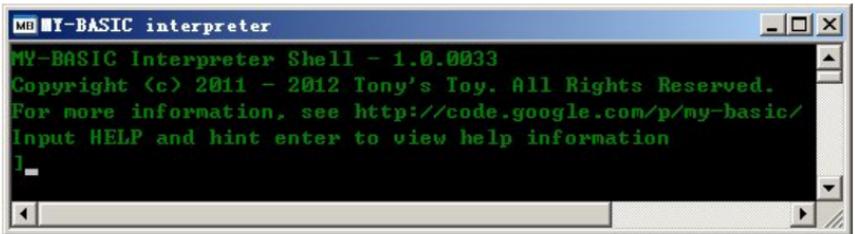
The close square bracket is an input prompt. Let's begin rock it by typing a classical 'hello world' tutorial as below:

```
' Hello world tutorial
a$ = "hello "
a$ = a$ + "world"
PRINT a$
```

You would get the response text after giving it a RUN command and hinting the enter key. Any text begins with a single quote and end to that line wouldn't be parsed to any interpretable structure but treated as comment; a comment does not perform anything, it's just a short explanation of statements. Like other BASIC dialects, MY-BASIC is not case-sensitive; it means PRINT A$ or Print a$ will have the same response.

Keywords

There are twenty two keywords and twenty four function reserved words in MY-BASIC as below:

Keywords	MOD, AND, OR, NOT, LET, DIM, IF, THEN, ELSE, FOR, TO, STEP, NEXT, WHILE, WEND, DO, UNTIL, EXIT, GOTO, GOSUB, RETURN, END
Reserved words	ABS, SGN, SQR, FLOOR, ROUND, RND, SIN, COS, TAN, ASIN, ACOS, ATAN, EXP, LOG, ASC, CHR, LEFT, LEN, MID, RIGHT, STR, VAL, PRINT, INPUT

It is not allowed to use these words for user-defined identifier; in addition there are two more TRUE and FALSE predefined ones besides these words, which represent Boolean value true and false, you are capable of redefining these two words but it's not

recommended. Meaning of each keyword will be mentioned latter in this manual.

Operators

There are eleven operators in MY-BASIC as below:

Operators	+, -, *, /, ^, =, <, >, <=, >=, <>

All these operators could act in calculation or comparison. Besides these operators, keywords MOD, AND, OR, NOT can also perform operator behaviors. An operator priority level shown in the table below indicates execution order in an expression:

Level	Operation
1	^
2	*, /, MOD
3	+, - (minus)
4	<, >, <=, >=, <>, = (equal comparison)
5	NOT
6	AND, OR
7	- (negative)
8	= (assignment)

The priority of level 1 is the highest and level 8 is the lowest. Higher level operations are treated before the lower ones. An expression is processed from left to right; operations in a same level are dealt in the same way. Brackets '(' and ')' are used in pair to process expression between them before operations which are not surrounded by them.

MOD means modulus, it is usually signified by percent symbol '%' in some other programming languages. The caret symbol stands for power operation thus 2 ^ 3 results 8.

Data and operation

There are four kinds of built-in data type in MY-BASIC: Integer, Real, Boolean, String.

Integer and Real are defined as int and float of C types which are 32bit size under most compiler architectures nowadays. And you could redefine them as other types like long and double by modifying a few lines of code if you need. Boolean's only instances are TRUE and FALSE, and can be assigned from a Boolean expression or an Integer. Actually Boolean is implemented and treated the same as Integer; zero means FALSE and none zero means TRUE.

MY-BASIC accepts numbers in HEX and OCT representation. A hexadecimal number begins with a 0x prefix, and an octonary one begins with a 0. For instance 0x10(HEX) equals to 020(OCT) equals to 16(DEC).

A variable identifier can contain alphabet and number, but must begin with a letter. A variable does not require declaration before using, so pay attention to spelling mistakes avoid strange errors. You don't need to take care of Integer / Float type conversion of a variable, an Integer variable can become a Float automatically if it's assigned with a Float value. Notice that a String variable must end with a dollar character $. An assignment statement consists of a beginning keyword LET and a following assignment expression, but the word LET

is optional. See below:

```
LET a = 1 ' Assignment statement begins with LET
pi = 3.14 ' Another assignment statement
```

MY-BASIC supports array up to four dimensions by default (defined by a macro). Array is a kind of collection data structure common in programming. An array can store a set of data which can be accessed via the array name and subscripts. A MY-BASIC array can hold either Real or String data. An array must be declared by a DIM (short for dimension) statement before using like this:

```
DIM nums(10)
DIM strs$(2, 5)
```

The naming rule for array identifiers is similar to variable rule. A dimension definition field followed an identifier begins with an open bracket and ends with a close one. Dimensions are separated by commas. Array indexes begin from zero in MY-BASIC thus nums(0) is the first element of array num. An array index could be a positive Integer, a variable of Integer or an expression; an invalid index would lead to an out of bound error.

MY-BASIC allows you to concatenate two Strings together using operator plus '+' and get a concatenate String. So be aware of that each String concatenate operation would generate a new String object. Numeric comparison operators can also apply to String. These operators start comparing the first character of each String, if they

are equal to each other, it continues with the following pairs until the characters differ or until a terminating null-character is reached; then return Integer values indicating the relationship between the Strings: a zero value if both Strings are equal, a positive one if the first is greater than the second one, a negative one if the first is less than the second one.

Sub program

It is recommended to break a program into small sub programs. Sub programs can bring duplicate coding and complicacy shielding. A label is used to define the enter point of a sub program. You can use a GOSUB statement wherever in the program to call a defined sub program and transfer control to it. A RETURN statement is used to exit a sub program and transfer control back to its caller.

Control structures

There are three types of control structures in common structured programming languages, and MY-BASIC is of no exception.

Serial structure that executes statements one by one is the most basic but also fundamental structure. MY-BASIC supports GOTO statement that provides unconditional control transfer ability. You can execute it like GOSUB as GOTO label, but that control cannot be returned back. An END statement can be placed anywhere in source code to terminate the whole execution.

Conditional structure consists of some condition statements (like IF, THEN, ELSE). These statements check a condition then perform an action in the case of true condition and in the case of false they

perform something else as specified by you. You can write multi statements in a single line by separating them with colons.

Loop structure statements check a loop condition and do the loop body in the case of true loop condition until it comes to the false case.

The FOR TO STEP NEXT loop statement is deemed as fixed step loop. See below that prints number one to ten:

```
FOR i = 1 TO 10 STEP 1
    PRINT i
NEXT i
```

The STEP segment is optional if the increment is one. The loop variable after NEXT is optional if it's continuing the closest FOR segment.

We don't know how many steps a loop has, sometimes. For this reason, unfixed step loops are necessary. There are two kinds of unfixed loops in MY-BASIC, WHILE WEND and DO UNTIL loop. See the code below:

```
a = 1
WHILE a <= 10
    PRINT a
WEND
```

```
b = 1
DO
     PRINT b
UNTIL a > 10
```

Just as their names imply, WHILE WEND loop do the loop body while the condition is true, and DO UNTIL loop do that until the condition is false. The difference is WHILE WEND checks the condition first before executing other codes, however, DO UNTIL checks the condition after the loop body has been executed once.

EXIT statement in MY-BASIC is used to interrupt current loop and continue to execute the program after it. It is the same as break statement in some other programming languages.

3.   Core and Standard Libraries

MY-BASIC supplies a set of frequently used function libraries which provides some fundamental numeric and string functions. These function name words couldn't be used as a user-defined variable identifier either. For details of these functions, see the figure bellow:

Type	Name	Description
Numeric	ABS	Returns absolute value of a number
	SGN	Returns sign of a number
	SQR	Returns arithmetic square root of a

		number
	FLOOR	Returns the greatest integer not greater than a number
	CEIL	Returns the least integer not less than a number
	FIX	Returns the integer trimmed format of a number
	ROUND	Returns the specified value to the nearest integer of a number
	RND	Returns a random float between 0.0 and 1.0
	SIN	Returns the sine of a number
	COS	Returns the cosine of a number
	TAN	Returns the tangent of a number
	ASIN	Returns the arcsine of a number
	ACOS	Returns the arccosine of a number
	ATAN	Returns the arctangent of a number
	EXP	Returns the base-e exponential of a number
	LOG	Returns the base-e logarithm of a number
String	ASC	Returns the integer ASCII code of a character
	CHR	Returns the character of an integer ASCII code

	LEFT	Returns a given number of characters from the left of a string
	LEN	Returns the length of a string
	MID	Returns a given number of characters from a given position of a string
	RIGHT	Returns a given number of characters from the right of a string
	STR	Returns the string format of a number
	VAL	Returns the number format of a string
Input & Output	PRINT	Outputs number or string to the standard output stream
	INPUT	Inputs number or string from the standard input stream

Be aware that all those functions besides PRINT and INPUT require a pair of brackets to envelop arguments.

4.   Application Programming Interface

There are a few but adequate exposed MY-BASIC APIs (Application Programming Interface) for C / C++ / Objective-C programs. MY-BASIC is written with pure C, what you need to do before programming with MY-BASIC is just copy and add my_basic.h and my_basic.c to the target project; all interfaces are declared in my_basic.h. Most APIs which return an int value return MB_FUNC_OK if there is no

execution error.

Interpreter structure

MY-BASIC uses an interpreter structure to store some necessary data structures during parsing and running period; like local / global function directory, global variable dictionary, abstract syntax tree, parsing / running context, error information etc. An interpreter structure is a unit of MY-BASIC environment. Invoking between MY-BASIC script and host program also works through this structure.

Initializing and disposing

int mb_init(void);

This function must and must only be called once before any other operations with MY-BASIC to initialize the entire system.

int mb_dispose(void);

This function must and must only be called once after operations with MY-BASIC to dispose the entire system.

int mb_open(mb_interpreter_t** s);

This function opens an interpreter structure to get ready for parsing and running.

Common usage of this function likes this:

```
mb_interpreter_t* bas = 0;
mb_open(&bas);
```

int mb_close(mb_interpreter_t** s);

This function closes an interpreter structure when it is no longer used. mb_open and mb_close must be matched in pair.

int mb_reset(mb_interpreter_t** s, bool_t clrf);

This function resets an interpreter structure to defaults as it was just opened.

Function registration

These functions are called to register or remove extended functions.

int mb_register_func(mb_interpreter_t* s,
                     const char* n,
                     mb_func_t f);

This function registers a function pointer into an interpreter structure using a given name. The function to be registered must be a pointer as int (* mb_func_t)(struct mb_interpreter_t*, void**). A registered function can be called in script.

int mb_remove_func(mb_interpreter_t* s,
                   const char* n);

This function removes a registered function out of an interpreter structure by a given name.

Invoking

These functions are called in extended functions to invoke with script.

int mb_attempt_func_begin(mb_interpreter_t* s,
                          void** l);

This function checks whether script is invoking an extended function in a legal begin format.

int mb_attempt_func_end(mb_interpreter_t* s,
                        void** l);

This function checks whether script is invoking an extended function

in a legal end format.

```
int mb_attempt_open_bracket(mb_interpreter_t* s,
                            void** l);
```

This function checks whether script is invoking an extended function in a legal format that begins with an open bracket before arguments list.

```
int mb_attempt_close_bracket(mb_interpreter_t* s,
                             void** l);
```

This function checks whether script is invoking an extended function in a legal format that ends with a close bracket after arguments list.

```
int mb_pop_int(mb_interpreter_t* s,
               void** l,
               int_t* val);
```

This function tries to pop an argument in int_t from an interpreter structure.

```
int mb_pop_real(mb_interpreter_t* s,
                void** l,
                real_t* val);
```

This function tries to pop an argument in real_t from an interpreter structure.

```
int mb_pop_string(mb_interpreter_t* s,
                  void** l,
                  char** val);
```

This function tries to pop an argument in char* from an interpreter structure.

```
int mb_pop_value(mb_interpreter_t* s,
                 void** I,
                 mb_value_t* val);
```

This function tries to pop an argument in mb_value_t from an interpreter structure. Use this function instead of mb_pop_int, mb_pop_real and mb_pop_string if an extended function accepts argument of generics types.

```
int mb_push_int(mb_interpreter_t* s,
                void** I,
                int_t val);
```

This function tries to push an argument in int_t format to an interpreter structure.

```
int mb_push_real(mb_interpreter_t* s,
                 void** I,
                 real_t val);
```

This function tries to push an argument in real_t format to an interpreter structure.

```
int mb_push_string(mb_interpreter_t* s,
                   void** I,
                   char* val);
```

This function tries to push an argument in char* format to an interpreter structure.

```
int mb_push_value(mb_interpreter_t* s,
                  void** I,
                  mb_value_t val);
```

This function tries to push an argument in mb_value_t format to an interpreter structure. Use this function instead of mb_push_int, mb_push_real and mb_push_string if an extended function returns value of generics types.

Parsing and running

```
int mb_load_string(mb_interpreter_t* s,
                   const char* l);
```

This function loads a string into an interpreter structure; then parses script source to executable structures and appends them to the abstract syntax tree.

```
int mb_load_file(mb_interpreter_t* s,
                 const char* f);
```

This function loads a string into an interpreter structure; then parses script source to executable structures and appends them to the abstract syntax tree.

```
int mb_run(mb_interpreter_t* s);
```

This function runs a parsed abstract syntax tree.

```
int mb_suspend(mb_interpreter_t* s,
               void** l);
```

This function suspends and saves current execution point. Some extended functions need this ability and resume that point after some other operations. Call mb_run to resume a suspended point.

Error handling

```
mb_error_e mb_get_last_error(mb_interpreter_t* s);
```

This function returns the latest error information of an interpreter

structure.

`const char* mb_get_error_desc(mb_error_e err);`

This function returns the description string of error information.

`int mb_set_error_handler(mb_interpreter_t* s,`
                                    `mb_error_handler_t h);`

This function sets the error callback handler of an interpreter structure.

`int mb_set_printer(mb_interpreter_t* s,`
                       `mb_print_func_t p);`

This function sets the PRINT handler of an interpreter structure. Use this to customize an output handler for the PRINT statement. The function to be set must be a pointer as `int (* mb_print_func_t)(const char*, ...)`. `printf` is set by default.

5.   Scripting with MY-BASIC

As to source code portability, the C programming language is most outstanding, because C compilers are available on almost every platform; that is why MY-BASIC is written in pure clean C so it can be compiled for PC, Mac, Cell phone, Game console etc. with none or a few modifications. It would be pretty easy to bind MY-BASIC in the existing project by just adding the MY-BASIC core which consists of a C source file and a header file into the target project.

First of all, you should recognize which parts in your project require speed and low level control, and which require flexibility and

augmentability. It's not wise to code kernel computation-intensive modules in script; script is appropriate for volatile parts of an entire program. Scripting programming languages are not omnipotent.

If it is explicit to you that using a scripting language would bring the benefit to your project then you should make and expose some interfaces correctly. More details on how to create your own scripting interfaces will be dealt with in the next chapter. After that you may complete your program with script invoking those scripting interfaces and pack them together into a publishable version.

6.   Customizing MY-BASIC

MY-BASIC is a free and open source software released under the MIT license which allows you to use, modify and extend the software for either commercial or noncommercial cases. You might need more scripting libraries according to your specific requirement though MY-BASIC has already provided some functions.

The first step is to define the function in your host program. All C / C++ / Objective-C functions that will be called from MY-BASIC script will be called using a pointer of type int (* mb_func_t)(struct mb_interpreter_t*, void**). Since an interpreter structure is used as the first argument of an extended function, the function actually can pop any number of arguments from the interpreter structure and push none or one return value back into the structure. The int return value indicates an execution status of an extended function which

always returns MB_FUNC_OK for no error. Let's make a maximum function that returns the maximum value of two integers as tutorial; see code below:

```
int maximum(mb_interpreter_t* s, void** l) {
    int result = MB_FUNC_OK;
    int m = 0;
    int n = 0;
    int r = 0;

    assert(s && l);

    mb_check(mb_attempt_open_bracket(s, l));
    mb_check(mb_pop_int(s, l, &m));
    mb_check(mb_pop_int(s, l, &n));
    mb_check(mb_attempt_close_bracket(s, l));

    r = m > n ? m : n;
    mb_check(mb_push_int(s, l, r));

    return result;
}
```

The second step is to register defined functions as: mb_reg_fun(bas, maximum) (supposing we have mb_interpreter_t* bas defined).

After that you can use a registered function as any other scripting interfaces like:

```
i = MAXIMUM(1, 2)
PRINT i
```

To perform a user defined abort, just return an integer value greater equal than a macro MB_EXTENDED_ABORT. It is recommended to add an abort value like:

```
typedef enum mb_user_abort_e {
    MB_ABORT_FOO = MB_EXTENDED_ABORT + 1,
    /* more... */
};
```

Then code return MB_ABORT_FOO; in your customized function.

7.   Using MY-BASIC as a Standalone Interpreter

You would be familiar with then MY-BASIC interpreter since you typed the hello world tutorial. There are some useful commands under interpreter mode:

Command	Summary	Usage
HELP	Shows help information.	
CLS	Clears screen.	

NEW	Clears current program.	
RUN	Runs current program.	
BYE	Quits interpreter.	
LIST	Lists current program.	LIST [l [n]], l is start line number, n is line count.
EDIT	Edits a line in current program.	EDIT n, n is line number.
LOAD	Loads a file as current program.	LOAD *.*.
SAVE	Saves current program to a file.	SAVE *.*.
KILL	Deletes a file.	KILL *.*.

Type a command (maybe also with several arguments) and hint enter to execute it. Command is only an aspect of the interpreter other than keyword, that is to say it is valid to use them as variable identifiers in a program; but to avoid reading confusion and conflict, you must consider it carefully.

8.   Extra Information

Document version
Version: 1.0 Dec. 2012
Author: Tony Wong
First edited date: Mar. 8, 2011
Last edited date: Dec. 10, 2012