MY-BASIC Quick Reference

1. Introduction

MY-BASIC is a tiny cross-platform easy extendable BASIC interpreter written in pure C with about 7000 lines of source code. Its grammar is similar to structured BASIC. 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, etc. 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 revision or information, see https://github.com/paladin-t/my_basic; or contact with the author through mailto:hellotony521@qq.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

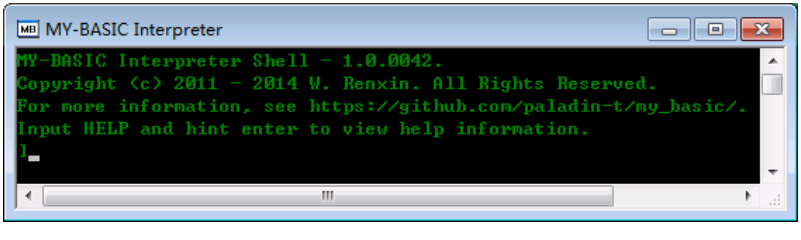
designed in 1964 by John George Kemeny and Thomas Eugene Kurtz at Dartmouth College in New Hampshire; and BASIC is famous because it is easy to learn and use all the time. Thank you all BASIC dedicators and fanatics.

MY-BASIC has an early structured BASIC like grammar. 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 https://github.com/paladin-t/my_basic/archive/master.zip or check out the source use *git clone https://github.com/paladin-t/my_basic.git* first if you didn't have a standalone interpreter yet. It is recommended to get a latest MY-BASIC package and have a quick blast with it first.

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 interpretable structure because it is a comment which won't influence the logic; a comment does not perform anything, it's just a short explanation of statements. Like other BASIC dialects, MY-BASIC is case-insensitive; it means *PRINT A$* or *Print a$* will perform the same behavior.

Keywords

There are thirty one keywords and twenty six reserved function words in MY-BASIC as below:

Keywords	MOD, AND, OR, NOT, LET, DIM, IF, THEN, ELSEIF, ELSE, ENDIF, FOR, TO, STEP, NEXT, WHILE, WEND, DO, UNTIL, EXIT, GOTO, GOSUB, RETURN, CALL, VAR, DEF, ENDDEF, CLASS, ENDCLASS, MEM, END
Reserved words	ABS, SGN, SQR, FLOOR, CEIL, FIX, 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 symbols 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 perform in calculation or comparison. Besides these operators, keywords *MOD*, *AND*, *OR*, *NOT* are operators as well. 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 direction. Brackets '(' and ')' are used in pair to tell the interpreter to process expression between them before operations outer.

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 types in MY-BASIC: Integer, Real, Boolean, String. Besides, MY-BASIC supports user defined data types as well, it will be explained later.

Integer and Real are defined as *int* and *float* in 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 wish. The only instances of Boolean are *TRUE* and *FALSE*, and can be assigned from a Boolean expression or an Integer expression. Actually Boolean is implemented and treated the same way as Integer; zero means *FALSE* and non-zero means *TRUE*.

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

A variable identifier is formed with alphabet and numbers, but it

must begin with a letter. A variable does not require declaration before using, so pay attention to spelling mistakes to avoid unexpected behaviours. You don't need to take care of conversion between Integer/Float values, an Integer variable can be changed to a Float automatically if it's assigned with a Float value. Notice that a String variable doesn't need to end with a dollar character *$*, which maybe a little different from some BASIC dialects. An assignment statement consists of a beginning keyword *LET* and a following assignment expression, practically the word *LET* is optional. See below:

```
LET a = 1 ' Assignment statement begins with LET
pi = 3.14 ' Another assignment statement without LET
```

MY-BASIC supports array up to four dimensions by default (defined with a macro), without doubt you can redefine the limitation. Array is a kind of regular collection data structure in programming aspect. An array can store a set of data that each element can be accessed via the array name and subscript. 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 the same as variable naming rule. A dimension definition field followed an array identifier begins

with an open bracket and ends with a close bracket. Dimensions are separated by commas. Array indexes begin from zero in MY-BASIC therefore *nums(0)* is the first element of array *nums*, note for this difference from other BASICs, but it's more common in most modern programming languages. An array index could be a non-negative Integer value formed as a constant, a variable of Integer or an expression which evaluation results an Integer; an invalid index would cause an out of bound error.

MY-BASIC allows you to concatenate two Strings together using operator plus "*+*" and get a concatenated String. So be aware of that each String concatenating operation would generate a new String object with memory occupation. Comparison operators can also apply to Strings. These operators start comparing the first character of each String, if they are equal to each other, it continues looking at the following ones until a difference or a terminating null-character "*\0*" is reached; then return Integer values indicating the relationship between the Strings: a zero value if both Strings are equal, a positive value if the first is greater than the second one, a negative value if the first is less than the second one.

Structured sub routine

It is recommended to break a program into small sub routines. Sub routines can reduce duplicate and complicacy code. MY-BASIC supports both structured sub routine with *CALL*/*DEF*/*ENDDEF* and instructional sub routine with *GOSUB*/*RETURN*, but you cannot use

them both in one program. It's recommended to use structured *CALL*/*DEF*/*ENDDEF* to write more elegant programs.

A sub routine begins with a *DEF* statement and ends with *ENDDEF*, you can add any numbers of parameters to a sub routine. It's quite similar to call a sub routine with calling a scripting interface, note you need to write an explicit *CALL* statement, if you were calling a sub routine which was defined below the calling statement. A sub routine returns the value of the last expression to its caller, or you may use an explicit *RETURN* statement. See below for example:

```
a = 1
b = 0
DEF FUN(d)
   d = CALL BAR(d)
   SIN(10)
   RETURN d ' Try comment this line
ENDDEF
DEF FOO(b)
   a = 2
   RETURN a + b
ENDDEF
DEF BAR(c)
   RETURN FOO(c)
ENDDEF
r = FUN(2 * 5)
PRINT r; a; b; c;
```

As you may see, a variable defined in a sub routine is only visible inside the local routine scope.

Instructional sub routine

Whatever, instructional sub routine is a valid option as well. A label is used to define the entry point of an instructional sub routine. You can use a *GOSUB* statement wherever in the program to call a labeled sub routine and transfer control to it. A *RETURN* statement is used to exit

a sub routine and transfer control back to its caller.

Control structures

There are three kinds of control structure in common structured programming languages, MY-BASIC supports them as well.

Serial structure that executes statements one by one is the most fundamental structure. MY-BASIC supports *GOTO* statement that provides unconditional control transfer ability. You can execute it like *GOSUB* as *GOTO label*, note instructional sub routine control proprietary cannot be returned back from a callee but unconditional *GOTO* cannot. Also, you cannot user both structured sub routine and unconditional jumping in one program. An *END* statement can be placed anywhere in source code to terminate the whole execution of a program.

Conditional structure consists of some condition jump statements (like *IF*, *THEN*, *ELSEIF*, *ELSE*, *ENDIF*). These statements check condition expressions then perform an action in a case of true condition branch and in a case of false it performs something else as specified by you.

You can write conditional *IF* statements in two ways. The first is single line format which the whole conditional chunk is written in a single line:

```
IF n MOD 2 THEN PRINT "Even" ELSE PRINT "Odd"
```

The other way is multi line statements:

```
INPUT n
IF n = 1 THEN
    PRINT "One"
ELSEIF n = 2 THEN
    PRINT "Two"
ELSEIF n = 3 THEN
    PRINT "Three"
ELSE
    PRINT "More than three"
ENDIF
```

It supports nested *IF* in multi line conditional statements.

Loop structure statements check a loop condition and do the loop body in a case of true until it comes to a 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 is associated with the closest *FOR* segment.

Sometimes, we don't know how many steps a loop would repeat. For this reason, variable step loops are quite essential. There are two

kinds of variable loops in MY-BASIC, *WHILE WEND* and *DO UNTIL* loops. 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. A key difference is *WHILE WEND* checks the condition first before executing the loop body, 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 names 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 the absolute value of a number
	SGN	Returns the sign of a number
	SQR	Returns the 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 number 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 type value of a number
	VAL	Returns the number type value of a string
Input & Output	PRINT	Outputs number or string to the standard output stream, user redirectable
	INPUT	Inputs number or string from the standard input stream, user redirectable

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

4. Application Programming Interface

There are a few but adequate exposed MY-BASIC APIs (Application Programming Interface) for C, C++, Objective-C, etc. programmers. MY-BASIC is written with pure C, what you need to do before scripting with MY-BASIC is just copy *my_basic.h* and *my_basic.c* to the target project, then add them to the project build configuration; all interfaces are declared in *my_basic.h*. Most APIs return an *int* value, they should return *MB_FUNC_OK* if there was no execution error most time.

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 (list), 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(struct mb_interpreter_t** s);

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

Common usage of this function does like this:

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

int mb_close(struct 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 sequentially.

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

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

Function registration/unregistration

These functions are called to register or remove extended functions.

int mb_register_func(struct 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 of *int (* mb_func_t)(struct mb_interpreter_t*, void**)*. A registered function can be called in MY-BASIC script.

int mb_remove_func(struct mb_interpreter_t* s,

const char* n);

This function removes a registered function out of an interpreter structure by a given name the same as it was registered.

int mb_remove_reserved_func(struct mb_interpreter_t* s,
const char* n);

This function removes a reserved function out of an interpreter structure by a given name. Do not use this function unless you really need to.

Invoking

These functions are utilities called in extended functions.

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

This function detects whether there is any more argument at current execution position in an interpreter structure. Use this function to implement a variable parameters interface function.

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

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

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

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

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

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

int mb_pop_usertype(struct mb_interpreter_t* s,
***void** l,*

***void** val);*

This function tries to pop an argument of *void** from an interpreter structure.

int mb_pop_value(struct mb_interpreter_t* s,
void** l,
mb_value_t* val);

This function tries to pop an argument of *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 arguments of generics types.

int mb_push_int(struct mb_interpreter_t* s,
void** l,
int_t val);

This function pushes an argument of *int_t* to an interpreter structure.

int mb_push_real(struct mb_interpreter_t* s,
void** l,
real_t val);

This function pushes an argument of *real_t* to an interpreter structure.

int mb_push_string(struct mb_interpreter_t* s,
void** l,
char* val);

This function pushes an argument of *char** to an interpreter structure.

int mb_push_usertype(struct mb_interpreter_t* s,
void** l,

void* val);

This function pushes an argument of *void** to an interpreter structure.

int mb_push_value(struct mb_interpreter_t* s,
void** l,
mb_value_t val);

This function pushes an argument of *mb_value_t* 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.

int mb_init_array(struct mb_interpreter_t* s,
void** l,
mb_data_e t,
int* d,
int c,
void** a);

This function initializes an array which MY-BASIC can use. The parameter *mb_data_e t* means what's the type of elements in the array, you can pass *MB_DT_REAL* or *MB_DT_STRING*; you need to disable the *MB_SIMPLE_ARRAY* macro to use a complex array and pass *MB_DT_NIL*. The *int* d* and *int c* stand for ranks of dimensions and dimension count. The function will put a created array to *void** *a*.

int mb_get_array_len(struct mb_interpreter_t* s,
void** l,
void* a,

int r,
int* i);

This function gets the length of an array. *int r* means which dimension you'd like to get.

int mb_get_array_elem(struct mb_interpreter_t* s,
void** l,
void* a,
int* d,
int c,
mb_value_t* val);

This function gets the value of an element in an array.

int mb_set_array_elem(struct mb_interpreter_t* s,
void** l,
void* a,
int* d,
int c,
mb_value_t val);

This function sets the value of an element in an array.

int mb_dispose_value(mb_interpreter_t* s,
mb_value_t val);

This function disposes a value popped from an interpreter. Now used for strings only.

Parsing and running

int mb_load_string(struct 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(struct 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(struct mb_interpreter_t* s);

This function runs a parsed abstract syntax tree in an interpreter structure.

int mb_suspend(struct 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* as well to resume a suspended point.

int mb_schedule_suspend(struct mb_interpreter_t* s,
int t);

This function schedules a suspend event, and it will trigger the event after the active statement execution is done. It's useful to do so when you need to do something else during the whole execution.

A). *mb_schedule_suspend(s, MB_FUNC_SUSPEND);* It's re-enterable which means next *mb_run* will resume execution from where you

suspended. B). *mb_schedule_suspend(s, MB_FUNC_END);* Terminate an execution normally, no error message. C). *mb_schedule_suspend(s, MB_EXTENDED_ABORT);* Or pass an argument greater than *MB_EXTENDED_ABORT* to terminate an execution and trigger an error message. You can call *mb_schedule_suspend* either in *_on_stepped* or in a scripting interface function. The difference between *mb_schedule_suspend* and *mb_suspend* is that *mb_suspend* can be called in a scripting interface only, and it cannot trap type B) and C) suspension.

Debugging

int mb_debug_get(struct mb_interpreter_t* s,
const char* n,
mb_value_t* val);

This function retrieves the value of a variable using the identifier in an interpreter structure.

int mb_debug_set(struct mb_interpreter_t* s,
const char* n,
mb_value_t val);

This function sets a variable using the identifier with a given value in an interpreter structure.

int mb_debug_set_stepped_handler(struct mb_interpreter_t* s,
mb_debug_stepped h);

This function sets a single step handler of an interpreter structure. The function to be set must be a pointer of *void (**

mb_debug_stepped_handler_t)(struct mb_interpreter_t, int, unsigned short, unsigned short)*. This function is useful for step by step debugging.

Error handling

mb_error_e mb_get_last_error(struct 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(struct mb_interpreter_t* s,
mb_error_handler_t h);

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

Stream redirection

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

This function sets a *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 of *int (* mb_print_func_t)(const char*, …)*. *printf* is set by default.

int mb_set_inputer(struct mb_interpreter_t* s,
mb_input_func_t p);

This function sets the *INPUT* handler of an interpreter structure. Use

this to customize an input handler for the *INPUT* statement. The function to be set must be a pointer of *int (* mb_input_func_t)(char*, int)*. *mb_gets* is set by default.

Miscellaneous

int mb_gets(char* buf,
int s);

A more safety evolvement of the standard *gets*.

int mb_memdup(char* val,
unsigned size);

This function duplicates a piece of memory to a MY-BASIC manageable buffer structure; use this to generate an argument for strings to be pushed. Note this function only copy bytes in given *size*, thus you have to add an extra byte to *size* for ending '*\0*'.

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, mobile devices, game console, super computers, MCU, smart cards, etc. with none or few porting modifications. It would be pretty easy to bind MY-BASIC in an existing project by just adding the MY-BASIC core which consists of a header declaration file and corresponding C implementation file into the

target project.

First of all, you should recognize which parts in your project require execution speed and low level control, and which parts 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. There is no one fits all solution; scripting programming languages are not omnipotent.

If it is explicit to you that using a scripting language would benefit 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 MY-BASIC script, invoking those scripting interfaces and pack them together into a publishable version.

Besides the scripting benefits, play with a scripting language itself is a really enjoyable thing.

6. Customizing MY-BASIC

Redirect PRINT and INPUT

Include a header file to use variable argument list:

```
#include <stdarg.h>
```

Customizable print handler:

```
int my_print(const char* fmt, ...) {
   char buf[1024];
   va_list argptr;

   va_start(argptr, fmt);
   vsnprintf(buf, sizeof(buf), fmt, argptr);
   va_end(argptr);

   printf(buf); /* Change me. */

   return MB_FUNC_OK;
}
```

Customizable input handler:

```
int my_input(char* buf, int s) {
    int result = 0;
    if(fgets(buf, s, stdin) == 0) { /* Change me. */
        fprintf(stderr, "Error reading.\n");
        exit(1);
    }
    result = (int)strlen(buf);
    if(buf[result - 1] == '\n')
        buf[result - 1] = '\0';

    return result;
}
```

Register handlers to an interpreter:

```
mb_set_printer(bas, my_print);
mb_set_inputer(bas, my_input);
```

Now your printer and inputer would be invoked.

Write scripting APIs

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 uses. You might need more scripting libraries according to your specific requirement though MY-BASIC has already provided some functions. It is really simple in MY-

BASIC to do so.

The first step is to define the function in your host program. All C/C++/Objective-C callee functions that will be invoked from MY-BASIC script is 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 a tutorial; see code below:

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

	mb_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;
}
```

Quite simple, isn't it.

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

After that you can use a registered function as any other scripting interfaces in MY-BASIC 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 write *return MB_ABORT_FOO;* in your customized function when something uncontainable happened.

Use usertype values

MY-BASIC building types are quite few. It's easy to use usertype in MY-BASIC. It can accept whatever type you give it.

MY-BASIC doesn't care what the usertype is; it just holds a usertype value at a variable or an array element. Note *MB_SIMPLE_ARRAY* macro must be disabled when you wish to store usertype in arrayes.

There are only two essential interfaces to get or set a usertype: *mb_pop_usertype* and *mb_push_usertype*. You can push a *void** to an interpreter and pop a value as *void** as well.

Macros

Some features of MY-BASIC could be customized with macros.

MB_SIMPLE_ARRAY

Enabled by default. An entire array uses a unified type mark, which means there are only two kinds of array: *string* and *real_t*.

Disable this macro if you would like to store generic type values in an array including *int_t*, *real_t*, *usertype*. Besides, array of *string* is still another kind. Note non simple array requires extra memory to store type mark of each element.

MB_MAX_DIMENSION_COUNT

Defined as 4 by default. Change this to support arrays of bigger maximum dimensions.

MB_ENABLE_ALLOC_STAT

Enabled by default. Use *MEM* to tell how much memory in bytes is allocated by MY-BASIC. Note statistics of each allocation takes *sizeof(intptr_t)* more bytes memory.

MB_ENABLE_SOURCE_TRACE

Enabled by default. MY-BASIC can tell where it goes in source code when an error occurs.

Disable this to reduce some memory occupation. Only do this on memory sensitive platforms.

MB_COMPACT_MODE

Enabled by default. C *struct* may use a compact layout.

This might cause some strange pointer accessing bugs with some compilers (eg. Some MCU compilers). Try disable this if you met any

strange bugs.

_WARING_AS_ERROR

Disabled by default.

Enable this in *my_basic.c* to treat warnings as error, or they will be ignored silently.

Something like divide by zero, wrong typed arguments passed will trigger warnings.

_HT_ARRAY_SIZE_DEFAULT

Defined as 193 by default. Change this in *my_basic.c* to resize the hash tables. Smaller value will reduce some memory occupation, size of hash table will influence tokenization and parsing time during **loading**, won't influence **running** performance.

_SINGLE_SYMBOL_MAX_LENGTH

Defined as 128 by default. Max length of a lexical symbol.

7. Memory Occupation

In some memory limited environments, memory occupation is often a sensitive bottleneck. MY-BASIC provides a method to count how much memory has an interpreter context allocated. Write script like below to tell how much memory in bytes does MY-BASIC allocated:

```
PRINT MEM ' The keyword MEM is right for this
```

Note that it will take *sizeof(intptr_t)* bytes more of each allocation if this statistics is enabled.

Comment the *MB_ENABLE_SOURCE_TRACE* macro in *my_basic.h* to disable source trace to reduce some memory occupation, but you will lose the error locating feature as well.

Redefine the *_HT_ARRAY_SIZE_DEFAULT* macro with a smaller value minimum to *1* in *my_basic.c* to reduce memory occupied by hash tables in MY-BASIC. Value *1* means a linear lookup.

The memory is limited in embedded systems which can run for years and cause a severe waste of memory due to fragmentation. Besides, it's efficient for MY-BASIC to customizing a memory allocator, even on systems, with a plenty of memory. MY-BASIC provides an interface that let you do so.

An allocator need to be in form of:

typedef char (* mb_memory_allocate_func_t)(unsigned s);*

And a freer:

typedef void (mb_memory_free_func_t)(char* p);*

Then you can tell MY-BASIC to use them globally instead of standard malloc and free by:

MBAPI int mb_set_memory_manager(mb_memory_allocate_func_t a, mb_memory_free_func_t f);

Note the functors only affect things going inside *my_basic.c*, but *main.c* still uses the standard pair.

There is already a simple memory pool implementation in *main.c*. You need to make sure the *_USE_MEM_POOL* macro is defined to enable this pool, and undefine it to disable the mechanism.

There are four functions in this implementation as a tutorial:

_open_mem_pool opens the pool when setting up an interpreter; *_close_mem_pool* closes the pool when terminating; a pair of *_pop_mem* and *_push_mem* will be registered to MY-BASIC. Note *_pop_mem* will call the standard *malloc* if an expected size is not a common size in MY-BASIC; and it will take *sizeof(union _pool_tag_t)* extra bytes to store meta data with each common size allocation.A typical workflow may looks like below:

```
_open_mem_pool(); // Open it.
mb_set_memory_manager(_pop_mem, _push_mem); // Register them.
{
  mb_init();
  mb_open(&bas);
  // Other deals with MY-BASIC.
  mb_close(&bas);
  mb_dispose();
}
_close_mem_pool(); // Finished.
```

Strictly speaking, the tutorial pool doesn't guarantee to allocate continuous address memory, it is an object pool other than a memory pool, which pops a free chunk of memory with an expected size to user, and pushes it to the stack back when user frees it instead of freeing it to system. This could be a good start if you would like to implement your own memory pool algorithm optimized for a specific

system.

8. Using MY-BASIC as a Standalone Interpreter

You would be familiar with the MY-BASIC interpreter if you have tried the hello world tutorial. There are some useful commands in 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 (modify/insert/remove) a line in current program.	EDIT n, n is line number. EDIT -I n, insert a line before a given line, n is line number. EDIT -R n, remove a line, n is line number.
LOAD	Loads a file as current program.	LOAD *.*.
SAVE	Saves current program	SAVE *.*.

	to a file.	
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, and you may consider different identifier naming.

9. Extra Information

Document version

Version: 1.1.3 Sep. 2015

Author: Wang Renxin

License: the MIT license

First edited date: Mar. 8, 2011

Last edited date: Sep. 2, 2015