MY-BASIC
Quick Reference

Copyright (C) 2011 – 2017 Wang Renxin
https://github.com/paladin-t/my_basic

1. Introduction

MY-BASIC is a lightweight cross-platform easy extendable BASIC interpreter written in pure C with about twenty thousand lines of source code. MY-BASIC is a dynamic typed programming language. It supports structured grammar, and implements a style of OOP called prototype-based programming paradigm, furthermore it offers a functional programming ability with lambda abstraction. It is aimed to be either an embeddable scripting language or a standalone interpreter. The core is very lightweight; all in a C source file and an associated header file; simpleness of source file layout and tightness dependency make it extraordinarily tough. Anyone even C programming newbies could learn how to use it and add new scripting interfaces in five minutes. It's able to easily combine MY-BASIC with an existing project in C, C++, Java, Objective-C, Swift, C# and many other languages. Scripting driven can make your projects more powerful, elegant and neat. It's also possible to learn how to build an interpreter from scratch with MY-BASIC, or build your own dialect easily based on it.

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 other information, see https://github.com/paladin-t/my_basic; or contact with the author through mailto:hellotony521@qq.com to get support.

2. Programming with BASIC

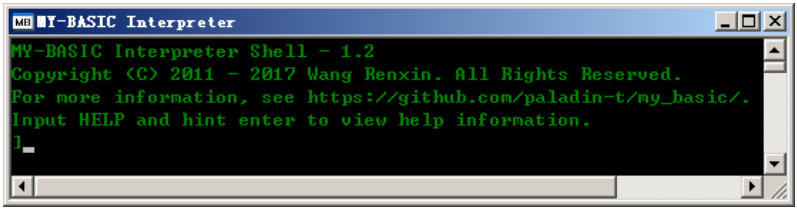
The well-known programming language BASIC is an acronym for Beginner's All-purpose Symbolic Instruction Code; when we mention BASIC today, we often refer to the BASIC family, not any specific one. The BASIC family has a long history since an original BASIC was designed in 1964 by John George Kemeny and 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, your passion affected many people like me, and guided us to the computer science.

MY-BASIC has a structured BASIC like grammar, and with many other retro and modern features. It would be familiar to you if you ever had programmed with another 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 code to make a build manually, if you didn't have the standalone interpreter yet by using *git clone https://github.com/paladin-t/my_basic.git*. It is recommended to get a MY-BASIC executable and have a quick glance.

In this part let's get start 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$;
```

Like other BASIC dialects, MY-BASIC is case-insensitive; it means *PRINT A$* or *Print a$* will perform the same behaviour. You would get the response text after giving it a *RUN* command and hinting the enter key. Any text begins with a single quote until the end of that line is called a comment (or remark) which won't influence the program logic; a comment does not perform anything, it's just a short explanation of code. It's also able to use the retro *REM* statement to start a comment as well.

Multi-line comment

MY-BASIC also supports multi-line comment, which is an advantage comparing to other dialects. It's surrounded by "*'[*" and "*']*", for example:

```
print “Begin”;
‘[
These lines won’t be executed!
print “Ignored”;
‘]
print “End”;
```

MY-BASIC will ignores these comments in this code. The convenience is it’s able to simply modify “*’[*” to “*’’[*” to uncomment lines.

Support for Unicode

Unicode is widely used nowadays for international text representation; MY-BASIC supports both Unicode represented code identifier and worldwide string manipulation. For instance:

```
print "你好" + "世界";

日本語 = "こんにちは"
print 日本語, ", ", len(日本語);
```

Keywords

There are a few keywords and some reserved function words in MY-BASIC as below:

Keywords	REM, NIL, MOD, AND, OR, NOT, IS, LET, DIM, IF, THEN, ELSEIF, ELSE, ENDIF, FOR,

		IN, TO, STEP, NEXT, WHILE, WEND, DO, UNTIL, EXIT, GOTO, GOSUB, RETURN, CALL, DEF, ENDDEF, CLASS, ENDCLASS, ME, NEW, VAR, REFLECT, LAMBDA, MEM, TYPE, IMPORT, END
Reserved words	Standard library	ABS, SGN, SQR, FLOOR, CEIL, FIX, ROUND, SRND, RND, SIN, COS, TAN, ASIN, ACOS, ATAN, EXP, LOG, ASC, CHR, LEFT, LEN, MID, RIGHT, STR, VAL, PRINT, INPUT
	Collection library	LIST, DICT, PUSH, POP, BACK, INSERT, SORT, EXISTS, INDEX_OF, GET, SET, REMOVE, CLEAR, CLONE, TO_ARRAY, ITERATOR, MOVE_NEXT

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 obviously represent Boolean value true and false, it's not allowed to reassign these two symbols with other value. 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 can perform in calculation or comparison. Besides these operators, keywords *MOD*, *AND*, *OR*, *NOT*, *IS* are operators as

well. An operator priority level shown in the table below indicates execution order in an expression:

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

The priority of level 1 is the highest and level 7 is the lowest. Higher level operations are taken before 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

A variable doesn't have a type, but a value does, so I can claim MY-BASIC is a dynamic type language. It's helpful to write code properly after learning some details about data types in MY-BASIC, although you could just assign whatever you want to a variable

without taking care about data types most of the time. There are some kinds of built-in data types in MY-BASIC: Nil, Type, Integer, Real, Boolean, String, Array, List, List Iterator, Dictionary, Dictionary Iterator, Prototype (a.k.a. Class), and Sub Routine (including Lambda). Besides, MY-BASIC also supports user defined data types (Usertype and Referenced Usertype) to let you to customize your own data structures, it will be explained later.

Nil is a special type which includes only one value: *NIL*. A *NIL* represents nothing, similar to *null*, *none*, *nothing*, etc. in other programming languages. Assigning a variable with a Nil value will release/unreference the previous value it holds.

A Type typed value represents the type of a value, it will be explained later with the *TYPE* statement.

Integer and Real are defined as *int* and *float* in C types which are 32bit size under most compiler architectures nowadays (*int* is 16bit size under some embedded systems). And you could redefine them with other types like *long* and *double* by modifying a few lines of code according to your requirement. 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 processed in a similar way as Integer; zero means *FALSE* and non-zero means *TRUE*.

NIL, *FALSE*, and *0* all mean false in a Boolean condition; on the other hand all other value including a blank String “” mean true.

MY-BASIC accepts literal numbers in HEX and OCT format. A

hexadecimal number begins with a *0x* prefix, and an octadic begins with a *0*. For instance *0x10* (HEX) equals to *020* (OCT) equals to *16* (DEC).

It's not allowed to use explicit type specifier, because of MY-BASIC's dynamic nature. A variable identifier is formed with letters, underline and numbers, but it must begin with a letter or an underline. You don't need to declare a variable before using it, so pay attention to spelling mistakes to avoid unexpected behaviours. You don't need to take care of conversion between Integer and Float values, an Integer variable can be converted to a Float automatically if it's assigned with a Float value, and a Float value will be converted to an Integer if it doesn't has a real component. Notice that dollar characters *$* are optional for String variables, which maybe a little different from some early BASIC dialects. An assignment statement consists of a beginning keyword *LET* and a following assignment expression, of course 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 by a macro), you can definitely redefine this default value. 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 by the array name and subscript. A MY-BASIC array can hold either Real or String data; furthermore, it's able to store every type of data in an

array by modifying a macro. 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 naming variables, actually all user identifiers do the same. 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 BASIC, 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 may cause an out of bound error.

MY-BASIC allows you to concatenate two Strings together using operator plus "*+*" and get a concatenated result. So be aware of that each String concatenating operation would generate a new String object with memory allocation. 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 relativity 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 lexically defined after the calling statement. A sub routine returns the value of the last expression back 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 routine scope.

MY-BASIC supports recursive sub routines and tail recursion optimization.

Besides, the *CALL* statement is used to get a routine value itself as:

```
routine = call(fun) ' Get a routine value
routine() ' Invoke a routine value
```

Be aware it requires a pair of brackets to get the value, or it's a calling execution.

Instructional sub routine

Whatever, instructional sub routine is a reserved 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. Use a *RETURN* statement 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 can be returned back from a callee while unconditional *GOTO* cannot. Without doubt, you cannot use both structured sub routine and unconditional *GOTO* 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 (such as *IF*/*THEN*/*ELSEIF*/*ELSE*/*ENDIF*). These statements check condition expressions then perform an action in a case of true

condition branch, otherwise in a case of false it performs something else as you write.

You can write conditional *IF* statements in two ways. The first is single line which the whole conditional chunk is written in one compact 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 it";
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 number is one. The loop variable after *NEXT* is optional if it is associated with the associated *FOR* layer.

MY-BASIC supports loop on collections by using *FOR*/*IN*/*NEXT* statements. It's able to iterate a list or a dictionary in this way. The loop variable is assigned with the value of the element which an iterator is currently pointing to. For example, this counts one to five:

```
for i in list(1 to 5)
    print i;
next
```

Sometimes, we don't know how many times a loop should repeat. Variable step loops are quite essential for this purpose. 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;
    a = a + 1
wend
```

```
a = 1
do
    print a;
    a = a + 1
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 main difference is that *WHILE*/*WEND* checks the condition first before executing the loop body, while, *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 a "*break*" statement in some other programming languages.

Use a class

MY-BASIC supports prototype-based programming paradigm which is a kind of OOP (Object-Oriented Programming) paradigm. We mean the same thing, when we mention "class instance" or "prototype" in MY-BASIC. This programming paradigm can also be known as "prototypal", "prototype-oriented", "classless", or "instance-based" programming. Use a pair of *CLASS*/*ENDCLASS* statements to define a class (a prototype object). Use *VAR* to declare a member variable in a class. It's able to define member function (a.k.a. Method) in a prototype with the *DEF*/*ENDDEF* statements as well. Write another

prototype surrounding with a pair of parentheses after a declaration statement to inherit from it (which means using it as a meta class). Use *NEW* to clone a new instance of a prototype.
See below for example to use a class in MY-BASIC:

```
class foo
    var a = 1
    def fun(b)
        return a + b
    enddef
endclass

class bar(foo) ' Use Foo as a meta class (inheriting)
    var a = 2
endclass

inst = new(bar) ' Create a new clone of Bar
print inst.fun(3);
```

"*bar*" will simply link "*foo*" as a meta class. But "*inst*" will create a new clone of "*bar*" and keep the "*foo*" meta linkage.
MY-BASIC supports reflection on an instance of a prototype with the *REFLECT* statement. It iterates all variable fields and sub routines in a class and its meta class as well, and stores the *name*/*value* of a variable and the *name*/*type* of a sub routine to a dictionary. You need to have a binary build with the collection library enabled to use

reflection. Let’s have a look at this example:

```
class base
        var b = "Base"
        def fun()
                print b;
        enddef
endclass
class derived(base)
        var d = "Derived"
        def fun()
                print d;
        enddef
endclass
i = new(derived)
i.fun();
r = reflect(i)
f = iterator(r)
while move_next(f)
        k = get(f)
        v = r(k)
        print k, ": ", v;
wend
g = get(i, “fun”);
g()
```

Use Lambda

According to Wikipedia's description, a Lambda abstraction (a.k.a. anonymous function or function literal) is a function definition that is not bound to an identifier. Lambda functions are often:

1. Arguments being passed to higher order functions, or
2. Used for constructing the result of a higher-order function that needs to return a function.

A Lambda becomes a closure after it captured some values in outer scope.

MY-BASIC has a full support for Lambda, including invokable as a value, higher order function, closure and currying, etc.

A Lambda abstraction begins with a *LAMBDA* keyword.

```
' Simple invoke
f = lambda (x, y) (return x * x + y * y)
print f(3, 4);
```

```
' Higher order function
def foo()
    y = 1
    return lambda (x, z) (return x + y + z)
enddef
l = foo()
print l(2, 3);
```

```
' Closure
s = 0
def create_lambda()
	v = 0
	return lambda ()
	(
		v = v + 1
		s = s + 1
		print v;
		print s;
	)
enddef
a = create_lambda()
b = create_lambda()
a()
b()
```

```
' Currying
def divide(x, y)
    return x / y
enddef
def divisor(d)
    return lambda (x) (return divide(x, d))
enddef
half = divisor(2)
third = divisor(3)
print half(32); third(32);
```

```
' Return as a value
def counter()
    c = 0
    return lambda (n)
    (
        c = c + n
        print c;
    )
enddef
acc = counter()
acc(1)
acc(2)
```

Lambda enhanced MY-BASIC with functional programming abilities.

Check the type of a value

You can use a *TYPE* statement to tell what the type of a value is, or use a predefined string argument to generate a type information. It returns a type value. Eg:

```
print type(123); type("Hi"); ' Get type of values
print type("INT"); type("REAL"); ' Get type objects
```

It's also able to check whether a value match a specific type by using the *IS* operator as follow:

```
print 123 is type("INT"); "Hi" is type("STRING");
print inst is type("CLASS");
```

You can also use *IS* to tell whether an instance of a prototype is inherited from another prototype:

```
print inst is foo;
```

Pass a type value to the STR statement to get the type name in string.

Import another script file

It's necessary to separate modules to multiple files in large programs. MY-BASIC provides an *IMPORT* statement to let a script import another script file. Eg. assuming we got an "a.bas" which is:

```
foo = 1
```

And another “b.bas” which is:

```
import a.bas

print foo;
```

When MY-BASIC is parsing “b.bas”, it will load “a.bas” as if it was written in “b.bas”.

Import a module

It’s able to put some symbols in a module (or called “namespace” in other programming languages) to avoid naming pollution.

Use *IMPORT “@xxx”* to import a module, and all symbols in that module could be used without the module prefix. Please note for the moment a module can be defined in native code when registering scripting interfaces.

3. Core and Standard Libraries

MY-BASIC offers 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 identifier as well. 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
	SRND	Sets the seed of random number
	RND	Returns a random float number between [0.0, 1.0] by RND, or [0, max] by RND(max), or [MIN, MAX] by RND(min, max)
	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
	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, or format a class instance with the TO_STRING function
Common	VAL	Returns the number type value of a string, or the value of a dictionary iterator, overridable for referenced usertype and class instance
	LEN	Returns the length of a string or an array, or the element count of a LIST or a DICT, overridable for referenced usertype and class instance
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 these functions besides *PRINT* and *INPUT* require a pair of brackets to surround arguments; the *RND* statement is a little special, it supports both with and without brackets, see the figure for detail.

4. Collection Libraries

MY-BASIC supplies a set of *LIST*, *DICT* manipulation function libraries which provide creation, accessing, iteration, etc. as below:

Name	Description
LIST	Creates a list
DICT	Creates a dictionary
PUSH	Pushes a value to the tail of a list, overridable for referenced usertype and class instance
POP	Pops a value from the tail of a list, overridable for referenced usertype and class instance
BACK	Peeks the value of a tail of a list, overridable for referenced usertype and class instance
INSERT	Inserts a value at a specific position of a list, overridable for referenced usertype and class instance
SORT	Sorts a list increasingly, overridable for referenced usertype and class instance

EXISTS	Tells whether a list contains a specific **value** (**not index**), or whether a dictionary contains a specific key, overridable for referenced usertype and class instance
INDEX_OF	Gets the index of a value in a list, overridable for referenced usertype and class instance
GET	Returns the value of a specific index in a list, or the value of a specific key in a dictionary, or a member of a class instance, overridable for referenced usertype and class instance
SET	Sets the value of a specific index in a list, or the value of a specific key in a dictionary, or a member variable of a class instance, overridable for referenced usertype and class instance
REMOVE	Removes the element of a specific index in a list, or the element of a specific key in a dictionary, overridable for referenced usertype and class instance
CLEAR	Clears a list or a dictionary, overridable for referenced usertype and class instance
CLONE	Clones a collection, or a referenced usertype
TO_ARRAY	Copies all elements from a list to an array
ITERATOR	Gets an iterator of a list or a dictionary, overridable for referenced usertype and class

	instance
MOVE_NEXT	Moves an iterator to next position for a list or a dictionary, overridable for referenced usertype and class instance

For example which shows how to use collections:

```
l = list(1, 2, 3, 4)
set(l, 1, "B")
print exists(l, 2); pop(l); back(l); len(l);

d = dict(1, "One", 2, "Two")
set(d, 3, "Three")
print len(d)
it = iterator(d)
while move_next(it)
    print get(it);
wend
```

MY-BASIC supports accessing elements in a list or dictionary collection using brackets as well:

```
d = dict()
d(1) = 2
print d(1);
```

A list begins from zero as well as array does in MY-BASIC.

5. Application Programming Interface

There are a few but adequate exposed MY-BASIC APIs (Application Programming Interface) for C, C++, Java, Objective-C, Swift, C#, and other 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 *int* values representing for execution states, most of them should return *MB_FUNC_OK* if there was no execution error, check the *MB_CODES* macro in *my_basic.h* for details. Yet there are some exceptions, I'll describe them below.

Interpreter structure

MY-BASIC uses an interpreter structure to store necessary data during parsing and running period; like local, global function dictionary, variable scope dictionary, AST (Abstract Syntax Tree) list, parsing context, running context, error information, etc. An interpreter structure is a unit of MY-BASIC environment context. It also works through this structure when invoking between MY-BASIC script and host program.

Meta information

unsigned long mb_ver(void);

Returns the version number of current interpreter.

const char* mb_ver_string(void);

Returns the version number string of current interpreter.

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. This function is optional, it will clear all variables, and also all registered global functions if *tclrf* is *true*.

Fork

These functions are used to fork and join a running environment.

int mb_fork(struct mb_interpreter_t** s,
struct mb_interpreter_t* r,
bool_t cklv);

This function forks a new running environment, from *r* to *s*. All forked environments share the same registered functions, parsed code, etc. but uses its own running context. Pass *true* to *cklv* to let it check alive objects with this forked environment.

int mb_join(struct mb_interpreter_t** s);

This function joins a forked running environment.

int mb_get_forked_from(struct mb_interpreter_t* s,
struct mb_interpreter_t** src);

This function gets the source running environment of a forked one.

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 with a specific 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. This function returns how many entries has been influenced, thus non-zero means success.

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 specific name the same as it was registered. This function returns how many entries has been influenced, thus non-zero means success.

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 specific name. Do not use this function unless you really need to, for example, remove or replace built-in interfaces. This function returns how many entries has been influenced, thus non-zero means success.

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

This function begins a module with a module name. All functions registered after a module began will be put in that module. A module is as known as namespace in some other programming languages.

int mb_end_module(struct mb_interpreter_t* s);

This function ends the current module.

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 beginning format. Call it when beginning an extended function without parameters.

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 ending format. Call it when ending an extended function without parameters.

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 variadic arguments interface function. It returns zero for no more argument, non-zero for more.

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. Then stores the result to **val*.

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. Then stores the result to **val*.

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

This function tries to pop an argument of *char** (String) from an interpreter structure. And stores the pointer to **val*. You don't need to know how and when a popped string will be disposed, but keep in mind that a popped string may be disposed during next string argument popping, so, just process it or cache it in time.

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

This function tries to pop an argument of *void** (Usertype) from an interpreter structure. Use *mb_pop_value* if a usertype is bigger than *void**.

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 different types. Or you are popping other advanced data 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** (String) to an interpreter structure. Value of *char* val* must be allocated and disposable by MY-BASIC, use *mb_memdup* to make it before pushing.

For example:

mb_push_string(s, l, mb_memdup(str, (unsigned)(strlen(str) + 1));.

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

This function pushes an argument of *void** (Usertype) to an

interpreter structure. Use *mb_push_value* if a usertype is bigger than *void**.

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. Or you are pushing other advanced data types.

Class definition

These functions are used to define a class manually in native side.

int mb_begin_class(struct mb_interpreter_t* s,
void** l,
const char* n,
mb_value_t** meta,
int c,
mb_value_t* out);

This function begins a class definition with a specific name. *n* is the class name. *meta* is an array of *mb_value_t**, which are meta classes; *c* is the element count of the array meta. The generated class will be returned into *out*.

int mb_end_class(struct mb_interpreter_t* s,
void** l);

This function ends a class definition.

int mb_get_class_userdata(struct mb_interpreter_t* s,
void** l,
void** d);

This function gets the userdata of a class instance. The returned data will be stored into *d*.

int mb_set_class_userdata(struct mb_interpreter_t* s,
void** l,
void* d);

This function sets the userdata of a class instance with data *d*.

Value manipulation

These functions manipulate values.

int mb_get_value_by_name(struct mb_interpreter_t* s,
void** l,
const char* n,
mb_value_t* val);

This function gets the value of an identifier with a specific name. *n* is the expected name text. A pointer to an internal variable structure will be stored in *val*.

int mb_add_var(struct mb_interpreter_t* s,
void** l,
const char* n,
mb_value_t val,
bool_t force);

This function adds a variable with a specific identifier name and a

value. *n* is the name text. *val* is the value of the variable. *force* indicates whether overwrite existing variable.

```
int mb_get_var(struct mb_interpreter_t* s,
                    void** l,
                    void** v);
```

This function gets a token literally, and stores it in an argument *v* if it's a variable.

```
int mb_get_var_name(struct mb_interpreter_t* s,
                    void** v,
                    char** n);
```

This function gets the name of a variable, then stores it in an argument *n*.

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

This function gets the value of a variable into *val*.

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

This function sets the value of a variable from *val*.

```
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_init_coll(struct mb_interpreter_t* s,
void** l,
mb_value_t* coll);

This function initializes a collection; you need to pass a valid *mb_value_t* pointer with a specific collection type you'd like to initialize.

int mb_get_coll(struct mb_interpreter_t* s,
void** l,
mb_value_t coll,
mb_value_t idx,
mb_value_t* val);

This function gets an element in a collection. You can pass *LIST* index or *DICT* key to *mb_value_t idx*.

int mb_set_coll(struct mb_interpreter_t* s,
void** l,
mb_value_t coll,
mb_value_t idx,
mb_value_t val);

This function sets an element in a collection. You can pass *LIST* index or *DICT* key to *mb_value_t idx*.

int mb_remove_coll(struct mb_interpreter_t* s,

***void** l,*
mb_value_t coll,
mb_value_t idx);

This function removes an element from a collection. You can pass *LIST* index or *DICT* key to *mb_value_t idx*.

int mb_count_coll(struct mb_interpreter_t* s,
***void** l,*
mb_value_t coll,
int* c);

This function returns the count of elements in a collection.

int mb_keys_of_coll(struct mb_interpreter_t* s,
***void** l,*
mb_value_t coll,
mb_value_t* keys,
int c);

This function retrieves all keys of a collection. It gets indices of a *LIST* or keys of a *DICT*; and stores them in *mb_value_t* keys*.

int mb_make_ref_value(struct mb_interpreter_t* s,
void* val,
mb_value_t* out,
mb_dtor_func_t un,
mb_clone_func_t cl,
mb_hash_func_t hs,
mb_cmp_func_t cp,
mb_fmt_func_t ft);

This function makes a referenced usertype *mb_value_t* object which holds *void* val* as userdata. Note you need to specify some functors.

```
int mb_get_ref_value(struct mb_interpreter_t* s,
                     void** l,
                     mb_value_t val,
                     void** out);
```

This function gets the raw userdata from a referenced usertype.

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

This function increases the reference count of a referenced usertype.

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

This function decreases the reference count of a referenced usertype.

```
int mb_set_alive_checker(struct mb_interpreter_t* s,
                         mb_alive_checker_t f);
```

This function sets a global alive object checker.

```
int mb_set_alive_checker_of_value(struct mb_interpreter_t* s,
                                  void** l,
                                  mb_value_t val,
                                  mb_alive_value_checker_t f);
```

This function sets an alive object checker of a referenced usertype.

```
int mb_override_value(struct mb_interpreter_t* s,
                      void** l,
```

mb_value_t val,
mb_meta_func_e m,
void f);*

This function overrides a meta function of a referenced usertype.

int mb_dispose_value(struct mb_interpreter_t s,*
mb_value_t val);

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

Sub routine manipulation

int mb_get_routine(struct mb_interpreter_t s,*
*void** l,*
const char n,*
mb_value_t val);*

This function gets a sub routine object by its name.

int mb_set_routine(struct mb_interpreter_t s,*
*void** l,*
const char n,*
mb_routine_func_t f,
bool_t force);

This function sets a sub routine with a specific name using a native functor.

int mb_eval_routine(struct mb_interpreter_t s,*
*void** l,*
mb_value_t val,

mb_value_t* args,
unsigned argc,
mb_value_t* ret);

This function evaluates a subroutine object. *mb_value_t* args* is a pointer of an argument array, *unsigned argc* is the count of arguments. The last parameter *mb_value_t* ret* is optional; pass NULL to it if it's not necessary for further processing.

int mb_get_routine_type(struct mb_interpreter_t* s,
mb_value_t val,
mb_routine_type_e* y);

This function gets the sub type of a subroutine object. *mb_value_t val* is a value of a subroutine object, and the result will be assigned to *mb_routine_type_e* y*.

Parsing and running

int mb_load_string(struct mb_interpreter_t* s,
const char* l
boo_t reset);

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

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 (list).

int mb_run(struct mb_interpreter_t* s, bool_t clear_parser);

This function runs a parsed abstract syntax tree (list) 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* again to resume a suspended point. This is a reserved function for compatibility reasons to provide a limited suspending (from simple program) ability.

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. This is a reserved function for compatibility reasons to provide a limited suspending (from simple program) ability.

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_get_stack_trace(struct mb_interpreter_t* s,
void** l,
char** fs,
unsigned fc);

This function gets current stack frame trace. It requires the *MB_ENABLE_STACK_TRACE* macro enabled to use this function.

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

Type handling

const char* mb_get_type_string(mb_data_e t);

This function returns type information in string of a given type.

Error handling

```
int mb_raise_error(struct mb_interpreter_t* s,
                   void** l,
                   mb_error_e err,
                   int ret);
```

This function raises an error manually.

```
mb_error_e mb_get_last_error(struct mb_interpreter_t* s,
                             const char** file,
                             int* pos,
                             unsigned short* row,
                             unsigned short* col);
```

This function returns the latest error information of an interpreter structure, and detail location.

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)(const char*, char*, int)*. *mb_gets* is set by default. The first parameter is an optional prompt text if you are doing *INPUT "Some text", A$*. Just ignore it if you don't need it.

Miscellaneous

bool_t mb_get_gc_enabled(struct mb_interpreter_t* s);

This function gets whether garbage collection is enabled.

int mb_set_gc_enabled(struct mb_interpreter_t* s,
bool_t gc);

This function sets whether garbage collection is enabled. Can be used to pause and resume GC.

int mb_gc(struct mb_interpreter_t* s,
int_t* collected);

This function tries to trigger a garbage collection.

int mb_get_userdata(struct mb_interpreter_t* s,
void** d);

This function gets the userdata of an interpreter instance.

int mb_set_userdata(struct mb_interpreter_t* s,
void* d);

This function sets the userdata of an interpreter instance.

int mb_set_import_handler(struct mb_interpreter_t* s,
mb_import_handler_t h);

This function sets a customized module importing handler.

int mb_gets(const char* pmt,
char* buf,
int s);

A more safety evolvement of the standard *gets*. Returns the length of input text.

char* mb_memdup(const char* val,
unsigned size);

This function duplicates a piece of memory to a MY-BASIC manageable buffer; use this function before pushing a string argument. Note this function only allocates and copies bytes with a specific *size*, thus you have to add an extra byte to *size* for ending "*\0*".

For example:

mb_push_string(s, l, mb_memdup(str, (unsigned)(strlen(str) + 1));.

int mb_set_memory_manager(mb_memory_allocate_func_t a,
mb_memory_free_func_t f);

This function sets a memory allocator and a freer globally to MY-BASIC.

6. Scripting with MY-BASIC

The C programming language is most outstanding, as to source code portability, 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, Tablet, Pad, Mobile Phone, PDA, Video Game Console, Raspberry Pi, Intel Edison, Arduino and even MCU, 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, then which parts require flexibility and augmentability. It's not wise to code kernel computation-intensive modules in script; scripting is appropriate for volatile parts of an entire program. There is no one-fits-all solution in computer science; scripting programming languages are cool but 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 explained in the next section. I hope MY-BASIC driven scripting could help you to make your project elegant and enjoyable when developing it. Besides the scripting driven benefits, playing with a scripting language itself is really an enjoyable thing.

7. 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[64];
	char* ptr = buf;
	size_t len = sizeof(buf);
	int result = 0;
	va_list argptr;
	va_start(argptr, fmt);
	result = vsnprintf(ptr, len, fmt, argptr);
	if(result < 0) {
		fprintf(stderr, "Encoding error.\n");
	} else if(result > (int)len) {
		len = result + 1;
		ptr = (char*)malloc(result + 1);
		result = vsnprintf(ptr, len, fmt, argptr);
	}
	va_end(argptr);
	if(result >= 0)
		printf(ptr); /* Change me */
	if(ptr != buf)
		free(ptr);

	return ret;
}
```

Customizable input handler:

```
int my_input(const char* pmt, 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 customized printer and inputer handlers would be invoked instead of the standard ones. Note you still need to use *PRINT* and *INPUT* in script after redirecting them. See follow for making new statements as scripting functions.

Write scripting APIs

MY-BASIC is a free and open source software released under the MIT license which allows you to use, modify, extend and distribute 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 add new facilities.

The first step is defining the function in your host program. All native callee functions which 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 this functions as: *mb_reg_fun(bas, maximum)* (assuming we already have *struct mb_interpreter_t* bas* initialized).

After that you can use a registered function as any other scripting interfaces in MY-BASIC as:

```
i = maximum(1, 2)
print i
```

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

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

Then use *return MB_ABORT_FOO;* in your customized function when something uncontainable happened.

Use usertype values

It may be not enough for all purposes with only MY-BASIC built-in types. It's easy to use Usertype in MY-BASIC. It can accept whatever data you give it.

MY-BASIC doesn't care what a Usertype really 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 an array.

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. This brings more than

using only built-in types.

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*, etc. Besides, array of *string* is still another type. Note non simple array requires extra memory to store type mark of each element.

MB_ENABLE_ARRAY_REF

Enabled by default. Compiles with referenced array if this macro defined, otherwise compiles as value type array.

MB_MAX_DIMENSION_COUNT

Defined as 4 by default. Change this to support arrays of bigger maximum dimensions. Note it cannot be greater than the maximum number which an *unsigned char* precision can hold.

MB_ENABLE_COLLECTION_LIB

Enabled by default. Compiles including *LIST* and *DICT* libraries if this macro is defined.

MB_ENABLE_USERTYPE_REF

Enabled by default. Compiles with referenced usertype support if this macro defined.

MB_ENABLE_ALIVE_CHECKING_ON_USERTYPE_REF

Enabled by default. Compiles with alive object checking functionality on referenced usertype if this macro defined.

MB_ENABLE_CLASS

Enabled by default. Compiles with class (prototype) support if this macro defined.

MB_ENABLE_LAMBDA

Enabled by default. Compiles with Lambda (anonymous function) support if this macro defined.

MB_ENABLE_MODULE

Enabled by default. Compiles with module (namespace) support if this macro defined. Use *IMPORT "@xxx"* to import a module, and all symbols in that module could be used without the module prefix.

MB_ENABLE_UNICODE

Enabled by default. Compiles with UTF8 manipulation ability if this macro defined, to handle UTF8 string properly with functions such as *LEN*, *LEFT*, *RIGHT*, *MID*, etc.

MB_ENABLE_UNICODE_ID

Enabled by default. Compiles with UTF8 token support if this macro defined, this feature requires *MB_ENABLE_UNICODE* enabled.

MB_ENABLE_FORK

Enabled by default. Compiles with fork support if this macro defined.

MB_GC_GARBAGE_THRESHOLD

Defined as 16 by default. It will trigger a sweep-collect GC cycle when such number of deallocation occurred.

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_ENABLE_STACK_TRACE

Enabled by default. MY-BASIC will record stack frames including sub routines and native functions if this macro defined.

MB_ENABLE_FULL_ERROR

Enabled by default. Prompts detailed error message. Otherwise all error types will prompts a uniformed "*Error occurred*" message. However, it's always able to get specific error type by checking error code in the callback.

MB_CONVERT_TO_INT_LEVEL

Describes how to deal with Real numbers after an expression is evaluated. Just leave it a Real if it's defined as *MB_CONVERT_TO_INT_LEVEL_NONE*; otherwise try to convert it to an Integer if it doesn't contains decimal part if it's defined as *MB_CONVERT_TO_INT_LEVEL_ALL*. Also you could use the *mb_convert_to_int_if_posible* macro to deal with an *mb_value_t* in your own scripting interface functions.

MB_PREFER_SPEED

Enabled by default. Prefers running speed over space occupation as possible. Disable this to reduce memory footprint.

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 embedded system compilers). Try disabling this if you met any strange bugs.

_WARNING_AS_ERROR

Defined as 0 by default.

Define this macro as 1 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 most of the time (except cross scope identifier lookup).

_SINGLE_SYMBOL_MAX_LENGTH

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

8. 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, but this extra footprint is not counted.

Comment the *MB_ENABLE_SOURCE_TRACE* macro in *my_basic.h* to disable source trace to reduce some memory occupation, but you will reserve error prompting only without source code locating.

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, for most parsing mechanism.

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 these functors only affect things going inside *my_ basic.c*, but *main.c* still uses the standard C library.

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 pool in the tutorial 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.

9. Using MY-BASIC as a Standalone Interpreter

You would be familiar with the MY-BASIC interpreter if you have tried the hello world tutorial. Execute the binary directly without any argument to launch interactive mode. There are some useful

commands for the interactive interpreter mode:

Command	Summary	Usage
HELP	Views 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 to a file.	SAVE *.*.
KILL	Deletes a file.	KILL *.*.
DIR	List all files in a directory.	DIR p, p is a directory path.

Type a command (maybe with several necessary arguments) then hint enter to execute it. Commands are only operations of the interpreter other than keyword, which means it's able to use them as

identifiers in a program, if a command isn't a keyword or reserved word neither (eg. LIST is a reserved word, and a command too). Whatever, it's better to avoid using commands as identifiers to get rid of reading confusion.

Pass a file path to the binary to load and run that script file in file execution mode.

Pass an argument *–e* and an expression to evaluate and print it, eg. *–e "2 * (3 + 4)"*, note the double quotation marks are required when an expression contains space characters.

Pass an option *–p* and a number to set the memory pool threshold size of an interpreter, eg. *–e 33554432* to set the threshold to 32MB. MY-BASIC will tidy the memory pool when the free list reached this size.