MY-BASIC
Quick Reference

Copyright (C) 2011 – 2026 Tony Wang
https://github.com/paladin-t/my_basic

1. Introduction

MY-BASIC is a lightweight BASIC interpreter written in standard C in dual files. It aims to be embeddable, extendable and portable. It is a dynamic typed programming language, reserves structured syntax, supports a style of prototype-based programming (OOP), also implements a functional paradigm by lambda abstraction. The core is written in a C source file and an associated header file. It's easy to either use it as a standalone interpreter or embed it with existing projects in C, C++, Java, Objective-C, Swift, C#, etc. and totally customizable by adding your own scripting interface. It's also possible to learn how to build an interpreter from scratch, or build your own dialect based on it.

This manual is a quick reference on how to program with MY-BASIC, what it does and what does not, 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.

2. Programming with BASIC

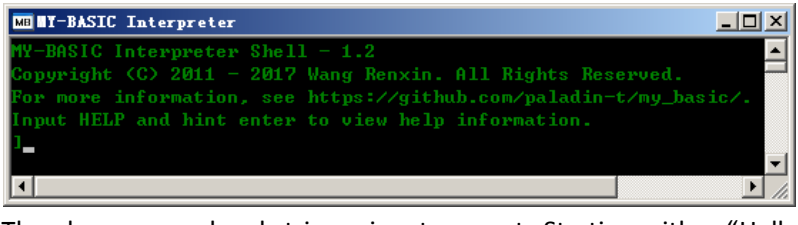
The well-known programming language BASIC is an acronym for Beginner's All-purpose Symbolic Instruction Code; when we mention BASIC today, we often refer to the BASIC family, not any specific one. The BASIC family has a long history since the original BASIC was

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. Thanks to all BASIC dedicators and fanatics, the passion affected many people like me, and guided us to the computer world.

MY-BASIC offers a structured BASIC syntax, and many other retro and modern features. It would be familiar to you if you have ever programmed with another BASIC dialect, or other programming languages.

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. You can get the latest revision by *git clone https://github.com/paladin-t/my_basic.git*. In this part let's get start using the MY-BASIC interpreter, which comes as follow:



The close square bracket is an input prompt. Starting with a "Hello World" convention in MY-BASIC:

```
' Hello world tutorial
input "What is your name: ", n$
def greeting(a, b)
    return a + " " + b + " by " + n$ + "."
enddef
print greeting("Hello", "world");
```

Like other BASIC dialects, MY-BASIC is case-insensitive; in another word *PRINT A$* and *Print a$* mean all the same. You will get a greeting text after input the above code, then input a *RUN* command and hinting Enter. Any text begins with a single quote until the end of that line is a comment (a.k.a. remark) which won't influence the logic of a program; a comment does not perform anything, but just a short explanation of code. It's possible to use the classic *REM* statement to start a comment as well.

MY-BASIC is configurable with macros, this manual is written with default configuration.

Multi-line comment

MY-BASIC also supports multi-line comment, which is a small advantage comparing to other dialects. It's surrounded by *'[* and *']*, eg.

```
print “Begin”;
‘[
print “This line won’t be executed!”;
print “This is also ignored”;
‘]
print “End”;
```

MY-BASIC ignores all comment lines between them. It is convenient to simply modify *‘[* to *“[* to uncomment all lines.

Support for Unicode

Unicode is widely used nowadays for international text representation; MY-BASIC supports both Unicode based identifier and string manipulations. Eg.

```
print "你好" + "世界";

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

Keywords

There are some keywords and reserved functions in MY-BASIC as follow:

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 functions	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 accepted to use these words for user-defined identifiers; in addition there are two more predefined boolean constants, a.k.a. *TRUE* and *FALSE*, as implied literally, represent boolean value true and false, it's not accepted to reassign these symbols with other values. Details of keywords and functions will be explained latter in this manual.

Operators

All operators in MY-BASIC as follow:

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

These operators are used in calculation or comparison expressions. Besides, the keywords *MOD*, *AND*, *OR*, *NOT*, *IS* also work as operators.

An expression is evaluated from left to right, with top down priorities as follow:

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

MOD stands for modulus, a.k.a. *%* in some other programming languages. The caret symbol ^ stands for power operation.

Data types and operations

MY-BASIC is a dynamic programming language, therefore variables don't have types, but values do. The built-in types are: Nil, Integer, Real, String, Type, 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 customize your own data structures.

Nil is a special type which includes only one valid value *NIL*, a.k.a. *null*, *none*, *nothing*, etc. It disposes/unreferences the previous value of a variable by assigning it with nil.

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

Integer and real are defined as *int* and *float* C types, which are both 32bit long under most compilers. You can redefine them as other types such as *long*, *long long*, *double* and *long double* by modifying a few lines of code. Since there is no dedicated boolean type, it's defined as integer also, and can be assigned from any expression. A boolean expression results to false with *NIL*, *FALSE*, and *0*; it results to true with all other values including blank string *""*.

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

A variable identifier is formed with letters, numbers, underline and an optional dollar postfix, but it must begin with a letter or an underline. It's not designed to use type specifier for variable, and you don't need to declare it before using it. You don't need to make conversion between integer and float values manually, generally MY-BASIC stores numbers with proper data type automatically. The dollar sigil *$* is reserved from traditional BASIC dialects as a valid postfix of a variable identifier. But it doesn't denote for type of string in most cases. However, there are special cases that *$* does mean something with the *DIM* and *INPUT* statements. An assignment statement consists of an optional beginning keyword *LET* and a following assignment expression. Eg.

```
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, which is defined by a macro. Array is a kind of regular collection data structure in programming. An array can store a set of data that each element can be accessed by the array name and subscripts. An array must be declared by a *DIM* (short for dimension) statement before using, eg.

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

The common naming rule for an array is same to naming a variable, actually all user identifiers in MY-BASIC follow the same rule. An array can be either a collection of real or string values, depends on whether the identifier ends up with a *$* sigil. Dimensions are separated by commas. Array indexes begin from zero in MY-BASIC therefore *nums(0)* is the first element of array *nums*, it is a bit different from other BASIC, but more common in 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 results an integer; invalid index may cause an out of bound error.

It is possible to concatenate two Strings together using the plus operator *+*. Each string concatenating generates a new string object with memory allocation. It is also possible to apply comparison operators to Strings, which starts comparing the first character of

both string, if they are equal to each other, it continues checking the following ones until a difference occurs or reaching the end of any one; then return an integer value indicating the difference.

Structured routine

It is possible to extract reusable code blocks with sub routines. MY-BASIC supports both structured routine with the *CALL*/*DEF*/*ENDDEF* statements and instructional routine with the *GOSUB*/*RETURN* (and *GOTO*) statements, but they can't be mixed together in one program.

A structured routine begins with a *DEF* statement and ends with *ENDDEF*, you can define routines with any arity. Calling a sub routine is similar to calling a native scripting interface. It requires an explicit *CALL* statement, if a routine is lexically defined after calling. A routine returns the value of the last expression to its caller, or returns explicitly with the *RETURN* statement. Eg.

```
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;
```

Each routine has its own scope for variable lookup.

Furthermore, the *CALL()* statement is used to get an invokable value as:

```
routine = call(fun) ' Get an invokable value
routine() ' Invoke an invokable value
```

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

Instructional routine

Traditional instructional routine is reserved as well. A label is used to tag the start point of an instructional routine. Use a *GOSUB* statement wherever in a program to call that labeled routine. The *RETURN* statement is used to exit a routine and transfer control back to its caller.

Control structures

There are three kinds of execution flows in MY-BASIC.

Serial structure, the most basic one, executes statements line by line. MY-BASIC supports *GOTO* statement which provides unconditional control transfer ability. Use it similar to *GOSUB* as *GOTO label*. The difference is that an instructional routine can return back from a callee, but unconditional *GOTO* cannot. The *END* statement can be placed anywhere in source code to terminate the whole execution of a program.

Conditional structures consist of some condition jump statements: *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.

Conditional *IF* statements in a single line:

```
if n mod 2 then print "Odd"; else print "Even";
```

Or multiple lines:

```
input n
if n = 1 then
	print "One";
elseif n = 2 then
	print "Two";
elseif n = 3 then
	print "Three";
else
	print "More than that";
endif
```

It supports nested *IF* with multi-line conditional statements. Note that single line *IF* doesn't require *ENDIF*, but the other does.

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

Use the *FOR*/*TO*/*STEP*/*NEXT* statements to loop for certain cycles. Eg.

```
for i = 1 to 10 step 1
	print i;
next i
```

The *STEP* part is optional if it increases by *1*. The loop variable after *NEXT* is also optional if it is associated with a corresponding *FOR*.

MY-BASIC also supports loop on collections with the *FOR*/*IN*/*NEXT* statements. It's possible to iterate list, dictionary, iterable class and usertypes. The loop variable is assigned with the value of the element

which an iterator is currently pointing to. Eg, this counts from one to five in list:

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

The *WHILE*/*WEND* and *DO*/*UNTIL* loops are used to loop for uncertain steps, or wait for specific conditions. Eg.

```
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, the *WHILE*/*WEND* statements do the loop body while the condition is true, and the *DO*/*UNTIL* statements do it until the condition is false. The *WHILE*/*WEND* statements check condition before executing loop body, while the *DO*/*UNTIL* statements check condition after loop body has been executed once.

The *EXIT* statement interrupts current loop and continues to execute the program outside loop.

Using class

MY-BASIC supports prototype-based OOP (Object-Oriented Programming). It is also as known as “prototypal”, “prototype-oriented”, “classless”, or “instance-based” programming. Use a pair of *CLASS*/*ENDCLASS* statements to define a prototype object (a class). Use *VAR* to declare a member variable in a class. It’s possible 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 (meaning using it as meta class). Use the *NEW* statement to create a new instance of a prototype. Eg.

```
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 simply links *foo* as meta class. *inst* creates a new clone of *bar* and keep the *foo* meta linkage.

MY-BASIC supports reflection on a prototype with the *REFLECT* statement. It iterates all variable fields and sub routines in a class and its meta class, and stores *name*/*value* pairs of variables and *name*/*type* pairs of sub routines to a dictionary. Eg.

```
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()
```

Using Lambda

A lambda abstraction (a.k.a. “anonymous function” or “function literal”) is a function definition that is not bound to an identifier. Lambda function is often used as:

1. argument passed to other functions, or
2. return value from a function.

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

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

A lambda expression begins with a *LAMBDA* keyword. Eg.

```
' 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);
```

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

Checking the type of a value

The *TYPE* statement tells the type of a value, or generates type information with a predefined type string. Eg.

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

It's also possible to check whether a value matches a specific type with the *IS* operator as follow:

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

The *IS* statement also tells whether an instance of a prototype is inherited from another one:

```
print inst is foo; ' True if foo is inst's prototype
```

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

Importing another BASIC file

It helps to organize source files by separating different parts into multiple files in a big project. The *IMPORT* statement imports other source file just as it were written at where it imports. Eg. assuming we got an "a.bas" as:

```
foo = 1
```

And another "b.bas" as:

```
import “a.bas”
print foo;
```

Now you can use everything imported from “a.bas”. MY-BASIC handles cycle importing properly.

Importing a module

It’s also possible to put some native scripting interfaces in a module (a.k.a. “namespace”) to avoid naming pollution. MY-BASIC doesn’t support making modules in BASIC for the moment. Use *IMPORT “@xxx”* to import a native module, all symbols in that module could be used without module prefix.

3. Core and Standard Libraries

MY-BASIC offers a set of frequently used functions which provide fundamental numeric and string operations. These function names cannot be used for user-defined identifiers. See the figure follow for details of these functions:

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 part of a number
	ROUND	Returns the nearest approximate integer of a number
	SRND	Seeds the random generator
	RND	Returns a random float number between [0.0, 1.0] with RND, or [0, max] with RND(max), or [MIN, MAX] with 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 ASCII code integer of a character
	CHR	Returns the character of an ASCII code

		integer
	LEFT	Returns a specific number of characters from the left of a string
	MID	Returns a specific number of characters from the specific position of a string
	RIGHT	Returns a specific number of characters from the right of a string
	STR	Returns the string representation of a number, or format a class instance with the TO_STRING function
Common	VAL	Returns the number representation 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

The *INPUT* statement is followed with an optional input prompt string, then the variable identifier to be filled; it accepts string when a *$* is specified, otherwise accepts number. Note that all these functions

except *PRINT* and *INPUT* require a pair of brackets surrounding arguments; the *RND* statement is a bit special, it comes either with or without brackets, see the figure for detail.

4. Collection Libraries

MY-BASIC offers a set of *LIST*, *DICT* manipulation functions, which provide creation, accessing, iteration, etc. as follow:

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 at 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, 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 at a specific index in a list, or the value with a specific key in a dictionary, or a member of a class instance, overridable for referenced usertype and class instance
SET	Sets the value at a specific index in a list, or the value with 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 at a specific index in a list, or the element with 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 over a list or a

	dictionary, overridable for referenced usertype and class instance

Using collections, eg.

```
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 using brackets directly:

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

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

5. Application Programming Interface

MY-BASIC is written cleanly in standard C in dual files. All you need to do to embed MY-BASIC with existing projects is just to copy *my_basic.h* and *my_basic.c* to your target project, then add them to the project build pipeline. All interfaces are declared in *my_basic.h*. Most API return *int* value representing for execution states, most of them should return *MB_FUNC_OK* if there is no error, check the *MB_CODES* macro in *my_basic.h* for details. Yet there are some exceptions when a simple *int* doesn't fit.

Interpreter structure

MY-BASIC uses an interpreter structure to store necessary data during parsing and running; including registered function, AST (Abstract Syntax Tree), parsing context, running context, scope, error information, etc. An interpreter structure is a unit of context.

Meta information

unsigned long mb_ver(void);

Returns the version number of the current interpreter.

const char* mb_ver_string(void);

Returns the version text of the current interpreter.

Initializing and disposing

int mb_init(void);

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

int mb_dispose(void);

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

int mb_open(struct mb_interpreter_t** s);

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

It usually comes as:

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

int mb_close(struct mb_interpreter_t** s);

This function closes an interpreter instance after using. *mb_open* and *mb_close* must be matched in pair.

int mb_reset(struct mb_interpreter_t** s, bool_t clear_funcs, bool_t clear_vars);

This function resets an interpreter instance, often for a new load-run cycle. It clears all variables if *clear_vars* is *true*; and also all registered global functions if *clear_funcs* is *true*, but this function does not re-register built-in interfaces.

Forking

These functions are used to fork and join an interpreter.

int mb_fork(struct mb_interpreter_t** s,

struct mb_interpreter_t* r,
bool_t clear_forked);

This function forks a new interpreter, from *r* to *s*. All forked environments share the same registered functions, parsed code, etc. but uses its own running context. Pass *true* to *clear_forked* to let the source instance collects and manages data in the forked one.

int mb_join(struct mb_interpreter_t** s);

This function joins a forked interpreter. Use this to close a forked interpreter.

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

This function gets the source interpreter of a forked one.

Function registration/unregistration

These functions are used to register or unregister native scripting interfaces.

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 with a specific name. The function must have signature as *int (* mb_func_t)(struct mb_interpreter_t*, void**)*. A registered function can be called in MY-BASIC code. This function returns how many entries have been influenced, thus non-zero means success. The specified identifier will be stored in upper case.

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

This function removes a registered function out of an interpreter with a specific registered name. This function returns how many entries have 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 with a specific name. Do not use this function unless you really want to, Eg. remove or replace built-in interfaces. This function returns how many entries have been influenced, thus non-zero means success.

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

This function begins a module context with a name. All functions registered after it will be put into that module. Module is as known as namespace, use the *IMPORT* statement to get shortcut to it.

int mb_end_module(struct mb_interpreter_t* s);

This function ends the current module context.

Interacting

These functions are used in extended functions to communicate with the core.

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

This function checks whether BASIC is invoking an extended function in a legal beginning way. 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 BASIC is invoking an extended function in a legal ending way. 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 BASIC is invoking an extended function in a legal way that begins with an open bracket.

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

This function checks whether BASIC is invoking an extended function in a legal way that ends with a close bracket after argument 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. Use this function to implement a variadic function. Returns zero for no more, otherwise non-zero.

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

This function tries to pop an argument as *int_t* from an interpreter. And 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 as *real_t* from an interpreter. And 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 as *char** (string) from an interpreter. And stores the pointer to **val*. You don't need to know how and when a popped string will be disposed, but note that a popped string may be disposed when popping next string argument, 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 as *void** (usertype) from an interpreter. Use *mb_pop_value* instead if a usertype is larger than *void**.

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

This function tries to pop an argument as *mb_value_t* from an interpreter. Use this function instead of *mb_pop_int*, *mb_pop_real* and *mb_pop_string* if an extended function accepts argument in different types, or popping other advanced data types.

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

This function pushes a value as *int_t* to an interpreter.

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

This function pushes a value as *real_t* to an interpreter.

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

This function pushes a value as *char** (string) to an interpreter. The memory of *char* val* must be allocated and disposable by MY-BASIC. Consider use *mb_memdup* to make it before pushing. Eg. *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 a value as *void** (usertype) to an interpreter. Use *mb_push_value* if a usertype is larger than *void**.

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

This function pushes a value in *mb_value_t* to an interpreter. Use this function instead of *mb_push_int*, *mb_push_real* and *mb_push_string*

if an extended function returns generics types. Or pushing other advanced data types.

int mb_push_managed_value(struct mb_interpreter_t* s,
void** l,
mb_value_t val,
bool_t managed);

This function is similar to *mb_push_value*, in addition it specifies whether a (string) value is managed by MY-BASIC or user codea.

Class definition

These functions are used to define a class manually at 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 in upper case. *meta* is an array of *mb_value_t**, which are optional meta classes; *c* is the element count of the meta array. 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 in upper case. *n* is the expected name text. It returns a value to **val*.

int mb_get_vars(struct mb_interpreter_t* s,
***void** l,*
mb_var_retrieving_func_t r,
int stack_offset);

This function retrieves the name and value of active variables in the specific stack frame, then returns the retrieved variable count. *stack_offset* indicates the stack offset, *0* for the current frame, *-1* for the root, other positive values for offset from the top (active) frame.

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 in upper case and a value to an interpreter. *n* is the name text. *val* is the value of the variable. *force* indicates whether overwrite existing value.

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

This function gets a token literally, and stores it in the parameter **v* if it's a variable. *redir* indicates whether to redirect result variable to any member variable of a class instance.

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 the parameter **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 can be used in BASIC. The parameter *mb_data_e t* stands for the type of elements in the array, can be either *MB_DT_REAL* or *MB_DT_STRING*; disable the *MB_SIMPLE_ARRAY* macro to use a complex array with passing *MB_DT_NIL* to it. The *int* d* and *int c* stand for ranks of dimensions and dimension count. The function stores the 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 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; 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. It accepts *LIST* index or *DICT* key with *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. It accepts *LIST* index or *DICT* key with *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. It accepts *LIST* index or *DICT* key with *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 raw userdata. Note you need to provide some functors to the core to manipulate it.

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 value.

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

This function decreases the reference count of a referenced value.

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

This function sets an object aliveness checker globally.

```
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 object aliveness checker on a specific referenced usertype value.

```
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 specific referenced usertype value.

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

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

Invokable manipulation

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

This function gets a routine value by name in upper case.

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 routine value with a specific name in upper case 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 an invokable. *mb_value_t* args* is a pointer to an array of arguments, *unsigned argc* is the count of it. The last optional parameter *mb_value_t* ret* receives return value; pass *NULL* to it if it's not used.

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 an invokable value. *mb_value_t val* is a value of an invokable value, 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; then parses BASIC source into executable structures and appends it to the AST.

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

This function loads a file into an interpreter; then parses BASIC source into executable structures and appends it to the AST.

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

This function runs a parsed AST in an interpreter.

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

(Obsolete.) This function suspends and saves current execution point. Call *mb_run* again to resume from a suspended point.

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

(Obsolete.) This function schedules a suspend event, and it will trigger the event after finishing active statements. 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 with a specific name in upper case.

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

This function sets the value of a variable with a specific name in upper case.

int mb_debug_count_stack_frames(struct mb_interpreter_t* s);

This function gets and returns the stack frame count.

int mb_debug_get_stack_trace(struct mb_interpreter_t* s,
char** fs,
unsigned fc);

This function traces current call stack. 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 prev,
mb_debug_stepped post);

This function sets a pair of step handlers of an interpreter. The functions to be set must be pointers of *int (* mb_debug_stepped_handler_t)(struct mb_interpreter_t*, void**, const char*, int, unsigned short, unsigned short)*. These functions are useful for step by step debugging, or handling extra stuff during execution, the *prev* and *post* handlers are invoked before and after execution of a statement respectively.

Type handling

const char* mb_get_type_string(mb_data_e t);

This function returns the string value of a specific 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 in an interpreter structure, and detail location. It also clears the latest error information.

const char* mb_get_error_desc(mb_error_e err);

This function returns the string value of a specific error.

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

This function sets an error handler of an interpreter.

IO redirection

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

This function sets a *PRINT* handler of an interpreter. 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)(struct mb_interpreter_t*, const char*, ...)*. Defaults to *printf*.

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

This function sets the *INPUT* handler of an interpreter. 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)(struct mb_interpreter_t*, const char*, char*, int)*. Defaults to *mb_gets*. The first parameter is an optional prompt text as *INPUT "Some text", A$*. Omit it if you don't need.

Miscellaneous

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

This function sets a customized importing handler for BASIC code.

int mb_set_string_measurer(mb_string_measure_func_t m);

This function registers a string length measurer globally.

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 of MY-BASIC globally.

bool_t mb_get_gc_enabled(struct mb_interpreter_t* s);

This function gets whether the garbage collecting is enabled.

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

This function sets whether the garbage collecting 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 garbage collecting manually. And gets how much memory has been collected.

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_gets(struct mb_interpreter_t* s,
const char* pmt,
char* buf,
int s);

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

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

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

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

6. Scripting with MY-BASIC

The C programming language is outstanding at source code portability, because C compilers are available on almost every platform. MY-BASIC is written in standard C, so it can be compiled for different platforms, with none or few modifications. It is also simple to embed MY-BASIC into existing projects by just adding the core in dual files.

It should be realized which part in your project is sensitive to

execution speed, and which part is sensitive to flexibility and configurability. Scripting is appropriate for the volatile part.

7. Customizing MY-BASIC

Redirecting PRINT and INPUT

Include a header file to use variadic:

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

Customize a print handler eg.

```
int my_print(const char* fmt, ...) {
	char buf[128];
	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;
}
```

Customize an input handler eg.

```
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 these handlers to an interpreter:

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

Now your customized printer and inputer will be invoked instead of the standard ones. Use *PRINT* and *INPUT* in BASIC to access it.

Writing scripting API

It is also possible to introduce your own scripting interfaces.

First define the function in a native programming language (often C). An invokable interface by BASIC is a pointer of *int (* mb_func_t)(struct mb_interpreter_t*, void**)*. An interpreter

instance is used as the first argument of an extended function, the function can pop variadic 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 should always return *MB_FUNC_OK* for no error. Let's make a *maximum* function that returns the maximum value of two integers as a tutorial; see the follow code:

```
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;
}
```

The second step is to register this functions by *mb_reg_fun(bas, maximum);* (assuming we already have *struct mb_interpreter_t* bas* initialized).

After that you can use it as any other BASIC functions in MY-BASIC as:

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

Just return an integer value greater than or equals to the macro *MB_EXTENDED_ABORT* to indicate 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 extended function when some error occurred.

Using usertype values

Consider using usertypes, when built-in type doesn't fit. It can accept whatever data you give. MY-BASIC doesn't care about what a usertype really is; it just holds a usertype value, and communicates with BASIC.

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**.

For more information about using referenced usertype, see the interfaces above, or check the website.

Macros

Some features of MY-BASIC could be configured by 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 is defined, otherwise compiles as value type array.

MB_MAX_DIMENSION_COUNT

4 by default. Change this to support more array dimensions. Note it cannot be greater than the maximum number which an *unsigned char* can hold.

MB_ENABLE_COLLECTION_LIB

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

MB_ENABLE_USERTYPE_REF

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

MB_ENABLE_ALIVE_CHECKING_ON_USERTYPE_REF

Enabled by default. Compiles with object aliveness checking on

referenced usertype if this macro is defined.

MB_ENABLE_CLASS

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

MB_ENABLE_LAMBDA

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

MB_ENABLE_MODULE

Enabled by default. Compiles with module (namespace) support if this macro is 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 is 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 is defined, this feature requires *MB_ENABLE_UNICODE* enabled.

MB_UNICODE_NEED_CONVERTING (Windows only)

Disabled by default. Compiles with UTF8 converting for console input/output on Windows if this macro is enabled.

MB_ENABLE_FORK

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

MB_GC_GARBAGE_THRESHOLD

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

MB_ENABLE_ALLOC_STAT

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

MB_ENABLE_SOURCE_TRACE

Enabled by default. To trace where 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 is defined.

MB_ENABLE_FULL_ERROR

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

MB_CONVERT_TO_INT_LEVEL

Describes how to handle real numbers after an expression is evaluated. Just leave it as real when 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 when 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_PRINT_INPUT_PROMPT

Enabled by default. Prefers to output the specified input prompt to the interpreter's output function.

MB_PRINT_INPUT_CONTENT

Disabled by default. Prefers to output the inputted content to the interpreter's output function.

MB_PREFER_SPEED

Enabled by default. Prefers running speed over space occupation as much 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 (for instance, some embedded systems). Try to disable this if you met any strange bugs.

_WARNING_AS_ERROR

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

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 for cross scope identifier lookup).

_SINGLE_SYMBOL_MAX_LENGTH

128 by default. Max length of a lexical symbol.

8. Memory Occupation

Memory footprint is often a sensitive bottleneck under some memory constrained platforms. MY-BASIC provides a method to count how much memory an interpreter has allocated. Write script as follow to tell it in bytes:

```
print mem;
```

Note that it will take *sizeof(intptr_t)* bytes more of each allocation if this statistics is enabled, the extra bytes don't count.

Remove the *MB_ENABLE_SOURCE_TRACE* macro in *my_basic.h* to disable source trace to reduce some memory occupation, it still prompts errors.

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, mostly for parsing mechanisms and dynamic lookup with complex identifiers.

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.

An allocator is in form of:

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

And a freer is in form of:

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

Then you can indicate 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 as *1* to enable it.

There are four functions in this implementation as 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* are registered to MY-BASIC. Note that *_pop_mem* calls the standard *malloc* if an expected size is not a common size in the pool; and it will take *sizeof(union _pool_tag_t)* extra bytes to store meta data with each allocation. A typical workflow looks as follow:

```
_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(); // Finish
```

Strictly speaking, the pool doesn't guarantee to allocate memory at continuous spaces, 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. Replace it with other efficient algorithms to get good performance and balance between space and speed.

9. Using MY-BASIC as a Standalone Interpreter

Execute the binary directly without any argument to launch in the interactive mode. Commands for this mode as follow:

Command	Summary	Usage
HELP	View the help	

	information	
CLS	Clear the screen	
NEW	Clear the current program	
RUN	Run the current program	
BYE	Quit the interpreter	
LIST	List the current program	LIST [l [n]], l is start line number, n is line count
EDIT	Edit (modify/insert/remove) a line in current program	EDIT n, n is line number; EDIT -i n, insert a line before a specific line, n is line number; EDIT -r n, remove a line, n is line number
LOAD	Load a file to the current program	LOAD *.*
SAVE	Save the current program to a file	SAVE *.*
KILL	Delete 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.

Pass a file path to the binary to load and run that BASIC file instantly.

Pass an option *–e* and an expression to evaluate and print it instantly, eg. *–e "22 / 7"*, note the double quotation marks are required when an expression contains spacing characters.

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