User's Guide
Embedded Script Language for I/G

- HummingBird Virtual Machine -

Jinhong Kim <Redperf@gmail.com>

HAANGILSOFT Co., LTD.

Revision: 110806, 081030, 070620

영상발생기팀

CONTENTS

Prologue

HBVM has been designed for Image Generator since 2007. A scripting system was required to replace the hard-coded Host-to-IG interface module. Interface module implementers can now easily re-code the script and run it as desired without the compilation and build process.

HBVM scripts can be programmed. It also shares variables, functions, and structures and interacts with host applications using events. HBVM can control host applications and vice versa (two-way operation).

The VM after the abbreviation HB stands for the virtual machine you know. Therefore, object code is generated internally rather than compiled. And object code can be disassembled.
Object code can also be generated in binary format and used by HBVM.

1. Prerequisites

<Syntax>

- Designed to have C/C++ similar syntax.
- Script is NOT case-sensitive.
- A word(token) can be 256 characters max in length.
- ASCII characters only.

<Statement>

- All the statements are ended by semicolon mark.
- Scope keywords '{', '}' should not be end-up by a semicolon.

<Operators>

- You can calculate equations using arithmetic operators.
- Arithmetic operators:

*+, +=, -, -=, *, *=, /, /=, =, ++(Increment by 1), --(Decrement by 1)*

Note:

++ or – must be placed after a variable.

Ex) A++; ➔ OK, ++A; ➔ Wrong

*The complex operators such like +=, -=, *=, /= allows only to have constant numbers, not a variable.*

Ex) A += A / 3 ➔ Wrong, It should be A = A + A / 3;

- Unary operator:

    - (Negative)

    The positive sign is not used.

- Calculation sample:

```
c = c / ((t + 20.0) * sin (3.14/2.0));
v = 3/(1+2/(-3-4))/5;
v = 1 * sin ( cos(1/3) );
```

<Type classification>

"Hello, world"	-> Double quoted	-> STRING
10.0	-> A number with a dot	-> DOUBLE
10	-> A number without a dot	-> INT

<Program main>

- Basic structure:

```
int         v;
function main
{
            v           = 7;
            echo        v;
}
```

As like main() in the C/C++, HBVM requires the entry point also. In the HBVM calls a ‘main’ as entry-point by default. The entry function can be replaced with the other function. A function body is scoped by braces. 'END' is not a label. It commands HBVM to exit now the loop.

2. COMMENTS

Syntax:

/* */	-> Block comment.
//	-> Line comment.

Remarks:

The comment style follows the C/C++ things.

You can use block-comments: **/* */** or, line comment: **//**

HBVM does NOT support nested comment.

3. PREPROCESSORS

3.1 #DEFINE

Syntax:

#DEFINE <name> <string|number>

Remarks:

Use #DEFINE for string replacement. This is the same to preprocessor '#define' in C/C++.

It is pre-processed by HBVM, and does not affect any performance at runtime.

3.2 #INCLUDE

Syntax:

#INCLUDE <filename>

Remarks:

Sample:

```
#include "c:/igs/advcon_opcode.igs"
#include "c:/igs/advcon_setting.igs"
#include "c:/igs/advcon_main_stub_init.igs"
#include "c:/igs/scene_model.igs"

function main
{
#include "c:/igs/advcon_main_stub.igs"          // including in function may possible also.
…
```

3.3 #STOP

Syntax:

#STOP

Remarks:

Stops the script parsing. It comment out the entire section after the #STOP keyword. If you just use END keyword instead of #STOP, VM continues parsing although it stops running at the END. So they are different.

3.4 #PORTPACK

Syntax:

#PORTPACK <const-int>

Remarks:

Arranges the memory alignment of the host-side ‘struct’ data and the VM’s STRUCT one.

Default alignment size is 4.

Sample:

```
// C/C++ (Host code)
#pragma pack(push, 2)
struct     time_control_type
{
           int Year;
           int Month;
           short Day;
           int Second;
};
#pragma pack(pop)

// SCRIPT
struct time_control_type
{
           int Year;
           int Month;
           short Day;
```

```
            int Second;
        }

        #PORTPACK 2              // set to 2-byte memory alignment.
        Time_control_type        tct1;
        #PORTPACK 4
        Time_control_type        tct2; // Same design to tct1 but different alignment.
```

As you see in the sample code, the main difference between '#pragma pack" in C/C++ and "#PORTPACK" in VM is that the #pragma is placed before the structure definition while the #PORTPACK is placed before the structure instance variable. It's because the struct(in script) is purposed only to specify the structure design. So, in VM, you can make various alignment instances for one structure.

3.5 ENTRY

Syntax:

#ENTRY [entryname]

Remarks:

Like in C/C++, HBVM script has a default entry point – a 'main'. In the script, you can specify the other entry point. This is done by #ENTRY preprocessor. The script can have no entry point. If you need no ENTRY point then write '#ENTRY NONE' or VM will annoy you with error message.

This also can be done programmatically by using provided API – 'init(…)'

4. DEFINITIONS

4.1 VARIABLES

Syntax:

<BYTE|SHORT|INT|FLOAT|DOUBLE|STRING> <varname> [const];

Remarks:

Types:

There are 6 types in the variable:

BYTE	= unsigned char
SHORT	= unsigned short int
INT	= int
FLOAT	= float
DOUBLE	= double
STRING	= char[256]

- Numbers or strings are recognized as constant items.
- Variables can be initialized during definition.

ex)

```
int v1;              // -> No initial value.  Set to 0 by default.
int v1 = 7;          // -> Set to 7 the v1.
int v1 = 7.8;        // -> Set to 7(cut and casting).
int a; int b; int c;
a = b = c = 7;       // Multiple substitution is OK
int v1 = 3.14 * 6;   // Wrong; Expression in definition is not allowed.
```

Scope:

In HBVM, all the variables have global scope through the whole script. The variable can be used after the definition.

4.2 STRUCT

Syntax:

```
STRUCT   <f-name>
{
        <BYTE|SHORT|INT|FLOAT|DOUBLE|STRING|STRUCT>      <e-name>;
        ...
}
```

Remarks:

The STRUCT keyword is very similar with 'struct' in C/C++, but it does not provide bit operator or union. And you cannot use enumerators in the struct definition.

Note that when you map the port data, you have to concern the memory alignment of the port data and the structure of struct definition. Please refer to '#PORTPACK' section.

- STRUCT can refer another STRUCT.
- STRUCT elements can be accessed using DOT(.) relation operator.
- f-name(STRUCT name) can have 256 characters max.
- e-name(element name) can have 64 characters max.

Sample:

```
struct   WheelInfo
{
          Int        Size;
          int        BoltNum;
          int        MetalType;
}

struct CarInfo
{
          int                  DoorNum;
          int                  BaseColor;
          WheelInfo            wheel;
          int                  Price;
          float                Scale;
}
                    ...

CarInfo             theCar;             // STRUCT instance cannot have initial value.
int                 WMType;
WMType = theCar.wheel.MetalType;
```

4.3 ARRAY

Remarks:

For performance reason, VM supports only 1-dimension array. You can specify the array type as following:

Ex)

```
int pos[3];          // 3 integer variables. Indices from 0 to 2.
```

- In the array definition, only a positive value is allowed. And can't put a variable or use an expression.

```
int pos[-3];          // error
int pos[k];           // error, too !
…
echo pos[ k/20 ];     // The expression is OK in the statement.
```

```
struct      time_control_type
{
            int Year;
            int Month;
            int Day;
            int Second[50];
};
struct      host_to_visual_data_type
{
            int                 Host_Checksum;
            int                 Host_Error_Code[30]; // OK
            time_control_type   Mission_Time;
            double              Pitch_Angle;
            ...
}
```

5. BRANCH AND LOOP

5.1 GOTO, LABEL

Syntax:

```
LABEL:
@<LabelName>;

GOTO:
GOTO   <LabelName>
```

Remarks:

GOTO jumps to the pointing label position unconditionally.

Don’t forget a statement mark a semicolon at the end.

Sample:

```
@again;
        ...
GOTO again;                 --> Loops infinitely.
```

5.2 IF, ELSE

Syntax:

```
IF <condition>
{
..
[ELSE;]
..
}
```

Remarks:

In <condition>, there are comparison operators such as '==', '!=', '<', '<=', '>', '>='.

Limits:

You can specify only one <condition>. There is no '&&' or '||' operators for multiple conditions. To solve this, you can use nested IF blocks.

5.3 WHILE, BREAK

Syntax:

```
WHILE <condition>
{
..
[BREAK;]
}
```

Remarks:

<condition> is the same to IF’s one.

To escape from the loop, use BREAK on anywhere meets the condition. You can use nested while.

Sample:

```
while q < 10
{
        echo "q = ";   echo q; echo "\n";
```

```
            q += 1;
            if   q == 3
            {
                        break;
            }
}
```

5.4 END

Syntax:

    END;

Remarks:

    Exits VM immediately.

5.5 RETURN

Syntax:

    RETURN   [var|expression];

Remarks:

Exits function immediately with/without a returning value.

- RETURN can have expression.
- RETURN with no argument returns without returning value.

6. FUNCTION DEFINITON

Syntax:

    FUNCTION   <name> [var,...]

Remarks:

Parameters to be passed via stack can be retrieved(or mapped) within the FUNCTION by using POP command. The variables are must be defined before use. Because the passing parameters are on the stack, you can retrieve anywhere within the proc, or to make simple, if you specify the parameter variables explicitly after the name of the procedure, the HBVM automatically retrieves the passing parameters without after-pop efforts.

Sample:

```
ex-1)

CALL SomeCalc a, b, 10.0;
...
int arg1;    int arg2;    float scale
// HBVM automatically retrieves each parameter respectively from the stack.
function SomeCalc arg1, arg2, scale
{
...
}

ex-2)

function SomeCalc
{
        int arg1;    int arg2;    float scale; // Can be placed before the function like ex-1.
        pop scale;    // -> User can retrieve each parameter from the stack by self.
        pop arg2    // -> User have to consider the pop sequence;
        pop arg1;    // The parameters are on the stack in LIFO sequence.
}

Ex-3)
Int Ret;
Ret = Call SomeCalc 1, 2;  // Let Ret to have the result.
Call SomeCalc 1, 2;      Pop Ret;    // This is OK too.
function SomeCalc p1, p2
{
…
Return p1 * p2;     // return the value.
}
```

- You can return a result by using RETURN(or PUSH)
- The parameters are CALL-BY-VALUE by default. If you need CALL-BY-REFERENCE process, please read the CALL-BY-REFERENCE section.

- As you see Ex-3, you can directly take the returning value from a function only in case of the returning thing is a value not a string.

7. FUNCTION CALL

7.1 CALL

Syntax:

CALL <proc name> [parameters,...];

Remarks:

You can arrange and reuse the repeated commands or statements in a function. The parameters are retrieved and pushed their value onto the internal stack. The stacking sequence is right to left. Because the HBVM does not support pointer conception, parameter passing method is call-by-value only. CALL saves the current IP. So if the process finishes, it restores the IP and executes from very after the CALL instruction. Note that all the variable-types passing by call will be converted into double.

7.2 CALL-BY-REFERENCE

By default, argument-passing-convention of VM is CALL-BY-VALUE. But as you know, making Swap function without pointer Ref/De-ref facility is impossible. HBVM does not provide pointer expression. So, we support CALL-BY-REFERENCE method.

Look at the following sample:

```
// Swap<int>    a with b.
function Swapi _a, _b                    // A, B
{
        int t;
        t = _a;
        _a = _b;
        _b = t;
}

function main
{
        int a = 1;
        int b = 2;
        call swapi a, b;    // Are they swapped expectedly ?    ➔    Not!!
```

```
            echoln a;
            echoln b;
}
```

You already know the code will not work correctly. And you might think the fault is in Swapi function because there's no ref or deref. In HBVM, the Swapi has no error in syntax. To make the variable a reference, you should put '&' just before the passing parameter.

```
            Call swapi &a, &b;     // They are swapped expectedly.
```

You need not to modify the function prototype anymore. You can put a '&' before the variable only when you need to modify. A variable with '&' will be modified.

Partial modification is allowed.

```
            Call swapi a, &b;      // It's ok. But only 'b' will be modified.
```

7.3 RECURSIVE

HBVM supports recursive call. Sample 1 shows how to code a recursive function and its escape condition.

Sample 1

```
int recur_count;
function recur
{
            echo "recur... ";   echoln recur_count++;
            if recur_count == 5
            {
               return;
            }
            call recur;
            echoln "return";
}
```

```
<RESULT>
recur... 0.00
recur... 1.00
recur... 2.00
recur... 3.00
recur... 4.00
return
return
return
return
```

Unfortunately, in the world of HBVM, there’s no local variable concept. So, you can try to code a function with parameters to be recursive by pushing the parameters before and popping them after the call instruction. Refer to Sample 2.

Sample 2

```
int recur_count;
int in;

function recur in
{
        echo "recur... ";    echoln recur_count++;
        echo "in = "; echoln in;

        if recur_count == 5
        {
            return;
        }

        push in;
        in++;
        call recur in; // push
        pop in;
        echo "return = ";    echoln in;
}
...
call recur 1;
```

```
<RESULT>
recur... 0.00
```

```
in = 1.00
recur... 2.00
in = 2.00
recur... 3.00
in = 3.00
recur... 4.00
in = 4.00
recur... 5.00
in = 5.00
return = 4.00
return = 3.00
return = 2.00
return = 1.00
```

7.4 EVENT CALL

Syntax:

EVENT <signum(INT)> <parameter(INT)>;

Remarks:

Through EVENT, the script can make a signal into the employer application in the same thread of a process. The employer thread have to offer signal listening function – a callgate. A callgate is a.k.a callback function. You can provide a parameter value for some purpose when you need. Otherwise, put a zero instead of parameter.

User can offer a callgate function where VM's initializer:

The initial function is:

void vm_init(char script, void (_stdcall *usercb)(int, int), char* entry);*

As you can see, the callee must have 'void (_stdcall *usercb)(int, int)' prototype.

```
void _stdcall vmCallGate( int funcid, int param )
{
        switch (funcid)
        {…
        }
}
```

```
vm_init( "c:/script.txt", vmCallGate, NULL );
```

If there's no callgate, provide a NULL to the second parameter place.

```
vm_init( "c:/script.txt", NULL, NULL );
```

7.5 PARAMETER PASSING ON EVENT

EVENT can pass one parameter value. When the parameter is insufficient, you can pass more values through the stack. You can push any variables or values before the EVENT. In the callgate, it pops up the value by using:

```
int        vm_pop_i();
double     vm_pop_f();
void       vm_pop_s(char*);
```

The callgate can return some value in the same way:

```
void       vm_push_i( int v );
void       vm_push_f( double v );
void       vm_push_s( char* v );
```

So, there’s a communication line between script and employer thread.

Sample:

```
void _stdcall vmCallGate( int signum, int param )
{
        if ( 2007 == signum )
        {
                double ang[3];
                ang[2] = vm_pop_f();
                ang[1] = vm_pop_f();
                ang[0] = vm_pop_f();
                double pos[3];
                pos[2] = vm_pop_f();
```

```
                        pos[1] = vm_pop_f();
                        pos[0] = vm_pop_f();

                        printf( "x: %f, y: %f, z: %f, pitch: %f, pitch: %f, roll: %f\n",
                                    pos[0], pos[1], pos[2], ang[0], ang[1], ang[2] );
            }
            else
            {
                        printf( "VM called: signum = %d, param = %d\n", signum, param );
                        vm_push_i( param * 10 );     // return a result.
            }
}
```

8. STACK OPERATIONS

8.1 PUSH

Syntax:

    PUSH   <var|const>;

Remarks:

8.2 POP

Syntax:

    POP   <var>;

Remarks:

8.3 PEEK

Syntax:

    PEEK   <var>;

Remarks:

8.4 READSTACK

Syntax:

    READSTACK   <level:var> <var>;

Remarks:

Reads stack value from the specified level.

9. OUTPUT

VM supports limited output method. VM can output a message into a debugger. Or it can display a message using MessageBox. If you have to make an output into a console or the other display equipment, then you have to extend VM by writing a library that communicates with a DLL which supports other output mechanism.

Actually, you can find the console output method from system.lib (offering library.) See Library section.

9.1 ECHO, ECHOLN

A message which is generated from ECHO or ECHOLN can be seen only in a debugger. If you don't know what is debugger or didn't find suitable one, I recommend you a DebugView which is free. It can be downloaded from Microsoft Download Center.

Syntax:

```
ECHO        <var|const|expression>;
ECHOLN      <var|const|expression>;
```

Remarks:

'ECHO' makes outputs into the debugger.

'ECHOLN' is the same to ECHO "…\n".

9.2 MSG

Syntax:

```
MSG    <var|const|expression>;
```

Remarks:

'MSG' makes output to be popped up like a MessageBox.

10. DATA MAPPING

10.1 MAPPORT

Syntax:

```
MAPPORT     <port:var|const> <output:struct-var>
```

Remarks:

MAPPORT maps the format structure into the binary block buffer which is provided by the external application.

10.2 READPORT

Syntax:

READPORT <port:var|const> <offset:var|const > <output:var>

Remarks:

READPORT is similar to MAPPORT. The difference is that a user can specify the offset in byte to read block buffer. You can directly put an offset value to the second variable. READPORT reads a value from the buffer according to the size of output variable and copies it into the output variable.

10.3 GETOFFSET

Syntax:

GETOFFSET <in:struct> <out:int-var>

Remarks:

GETOFFSET returns STRUCT item's offset in byte. Note that the parameter 'in' is not an instance of struct but is struct itself.

11. LIBRARY

In many cases, the often used routines are composed into the common functions. The functions can be grouped in a script file and it is called a library. HBVM cannot provide a library in binary form script file yet. So, the common routines are always free to modify by a user. So you need to care about it.

11.1 LOADMODULE

Syntax:

LOADMODULE <modname:string> <hnd:int-var>

Remarks:

The DLL module can be loaded into the VM. You can use system programming because you can call functions in all modules. The handle generated by LOADMODULE is the input to GETPROC.

11.2 UNLOADMODULE

Syntax:

UNLOADMODULE <hnd:int-var>

Remarks:

You must unload the module to conserve unnecessary system resources. Specify the module handle loaded in the parameter.

11.3 GETPROC

Syntax:

GETPROC <modhnd:int-var> <procname:string> <out:int-var>

Remarks:

Returns the procedure handle retrieved from the module. You can use CALLPROC to call the procedure. And you can get the value returned by GETRET.

11.4 CALLPROC

Syntax:

$ <prchnd:int-var> [args...]

Remarks:

The loaded module calls an internal binding function or function. In fact, the VM does not care about the type being used, and does not automatically do the type conversion like this inner call. Therefore, the user must use a variable of the correct type that corresponds to the parameter of the function to be called. Type matching is a very important issue. Incorrect typing can lead to system errors or instability, which can lead to inaccurate results or uncertain behavior.
Check what exact type you want to use.

Sample:

```
// To use MessageBox in the USER32.DLL
// You must know the prototype of the API completely.

int _hb_global_hnd_user32;
int _hb_global_hnd_messagebox;

#define MB_ICONEXCLAMATION          0x00000030

function main
{
        loadmodule "user32.dll", _hb_global_hnd_user32;
        getproc _hb_global_hnd_user32, "MessageBoxA", _hb_global_hnd_messagebox;
        …
        $_hb_global_hnd_messagebox 0,
                        “Hello, World”, “Sample”, MB_ICONEXCLAMATION;
        …
        unloadmodule _hb_global_hnd_user32;
}
```

11.5 GETRET

Syntax:

GETRET [ret:int-var]

Remarks:

After a calling some API, may be there a result value from that. Usually the value has integer type and passed through EAX register. You can read the returned value with GETRET method.

12. BINDING

With binding technology you can call the internal function or access the variables provided by caller. There are exist already data mapping and event processing method and supporting-API to read in script variables. But direct binding will be more faster or effective solution.
VM provides 4 APIs to bind with:

```
vm_bind_i( int* var, const char* varname );                     // BIND INT-VAR
vm_bind_f( double* var, const char* varname );                  // BIND DOUBLE-VAR
vm_bind_str( const char* var, const char* varname );            // BIND STRING-VAR
vm_bind_call( int (_stdcall *_bindcall)(), const char* varname ); // BIND A CALL
```

12.1 VARIABLE BINDING

VM have 3 major variable types: integer(i), float(f), string(str) And you can read all type of values through the major types.
Use vm_bind_i() for binding byte or short variable. And bind float and double variable with vm_bind_f().

```
vm_bind_i( int* var, const char* varname );                     // BIND INT-VAR
vm_bind_f( double* var, const char* varname );                  // BIND DOUBLE-VAR
vm_bind_str( const char* var, const char* varname );            // BIND STRING-VAR
```

In VM, the string variable always has 256 byte-length(null terminated) memory structure. So, to bind a string variable correctly, you have to provide 256 byte-length character array in the application also. If you provide a string variable of insufficient size, VM just tries to copy 256 bytes of string variable buffer into the caller’s memory while raising critical Memory Access

Violation error.

The VM does not take into account the size of the buffer. The VM is not secure.

12.2 FUNCTION BINDING

Basically, VM supports event method to make a communication. An event is essentially safe to use.

call	script-function
event	event-call

13. MISCELLANEOUS

13.1 MATHEMATICS

Syntax:

```
SQRT    <var|const|expression>
SIN     <var|const|expression>
COS     <var|const|expression>
EXP     <var|const|expression>
```

Remarks:

13.2 SLEEP

Syntax:

```
SLEEP   <var|const>
```

Remarks:

Sleeps (or delay) the VM. Unit is millisecond.