7.1 Introduction

Script objects will load the JavaScript built-in objects [ECMA] into the environment and also create objects at the time of interpretation. The script compiler compiles the source script. The compiled source script is represented as an AST and Symbol Table. In JavaScript the, keyword “var” is used to define a variable either in a global scope or function scope. The compiler records the variable name, used inside the function, into the symbol table with offset number which can not be updated during the execution. The script interpreter interprets the AST node by using Symbol table and with the Instruction stack (Non-recursive Stack), Execution Stack and Runtime stack. Traversing the AST will be a typical post-order traversal, which also must be implemented without recursion, for which we may use Instruction Stack. As we move from one execution context to the other, we may require to push them one after the other into a stack called Execution Stack, so that we can come back to the previous execution context with a mere pop. The idea is to emulate the way recursion really works in the existing machine architectures. It involves usage of a Runtime Stack in the Data Segment. The Runtime Stack consists of Stack Frames where each stack frame refers to a function call. Similar behavior has to be emulated in the form of a stack using linked-list; we can use the same name Runtime Stack for this.

On earlier use of the executor’s runtime stack, it was limited to keeping track of intermediate values while evaluating the expression. In a complete interpreter, the runtime stack plays a much greater role. At any point during the execution of a program, the stack represents a “snapshot” of the execution state. It contains the information about the procedures and functions that have been called, their return values and addresses, the current values of all their formal parameters and local variables, and which of these values are accessible according to the scope rules [144].

In JavaScript, for each function call, the execution context stack top is incremented and runtime stack is created. Runtime stack is created based on number of arguments and local variables ( e.g. var a ; inside function). All these local variables declared and defined inside function have the function scope. The runtime stack is created with its entries being type with the values of arguments and local variables updated. Whenever any symbol local to function is looked up, the offset value of the symbol is looked up in the runtime stack. If the
symbol is found, then the value would be returned or updated into the runtime stack. The execution stack is destroyed at the end of function call when the function returns the function value.

Closure is one of the most powerful features of JavaScript that allows inner functions, i.e. function definition inside the function bodies of another function. A closure is made when one such inner function is made accessible outside the function in which it was contained, so that it can be executed even after the function has returned. At that point, it still has access to the local variables, parameters and function declaration of its outer function. Those local variables, parameters and function declarations have the same value that they had when the outer function returned and may be interacted with by the inner function. In this chapter, we address the challenges of closure property in the JavaScript and design an algorithm to resolve this using runtime stack and execution stack without any memory garbage.

7.2 Problem Definition

In JavaScript, a function can be defined inside a function as inner function property. A function object can be assigned to a variable and it holds the function body and its local variables. If a non local variable of the function scope is assigned to the inner function, as per the closure behavior discussed earlier, the variable can be accessed out of the scope of the function as follows:

<script>
var X;
function foo( . . . ){  
    var A = 5;
    function subfoo(args){
        document.write(args+A);
    }
    subfoo(“inside the foo scope”)  
    X = subfoo;
}
foo();
X(“outside the foo scope”);
</script>
During the execution of “foo”, the variable “X” is assigned to the “subfoo”. The subfoo will be executed twice, i.e. once inside the function and once outside the function scope through the variable X, whereas in both the cases, it will print the value of “A” as “5”. During execution of the function, the runtime stack will keep the local variable of the function and will be deleted after the execution of the function. If it so, the value of “A” cannot be accessed as function X is invoked after the execution of function “foo”.

```javascript
var X, Y;
function foo(. . .){
    var A = 5;
    function subfoo(args){
        A++;
        document.write(args+A);
    }
    subfoo(“inside the foo scope”);  X = subfoo;  Y = subfoo;
}
foo();
X(“out side the foo scope”);
Y(“out side the foo scope”);
<script>
```

In the above code fragment, the output of value “A” will be 6, 7 and 8, i.e. after every execution of “subfoo”, the value of A will be increased.

In JavaScript, besides functions, this closure property appears under different features like “with object” and “eval”. Table–1 describes the list of typical closure behaviors test cases of JavaScript. We have tested this with different browsers like “Chrome”, “Mozilla” and “IE”.

<table>
<thead>
<tr>
<th>Sl.</th>
<th>Scripts</th>
<th>Results</th>
</tr>
</thead>
</table>
| 1   | function say(){
    var num = 6;
    var sayAlert = function(){
        if (num == 7){
        }  
    }
}                                                      | Success  |
| 6   | var x = new Object ;
    x.a=100;
    x.c=10;
    function add(){
}                                          | 10       |
<table>
<thead>
<tr>
<th>Sl.</th>
<th>Scripts</th>
<th>Results</th>
<th>Sl.</th>
<th>Scripts</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>document.write(&quot;Success&quot;) )</td>
<td></td>
<td></td>
<td>var a = 10;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>}</td>
<td></td>
<td></td>
<td>function</td>
<td></td>
</tr>
<tr>
<td></td>
<td>num++;</td>
<td></td>
<td></td>
<td>add1() { return a;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>return sayAlert;</td>
<td></td>
<td></td>
<td>with (x) {</td>
<td></td>
</tr>
<tr>
<td></td>
<td>}</td>
<td></td>
<td></td>
<td>c = add1;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>say();</td>
<td></td>
<td></td>
<td>} add(); document.write(x.c() ) ;</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>var cl1 = undefined;</td>
<td></td>
<td>5</td>
<td>var a = 56 ;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>function test1() {</td>
<td></td>
<td></td>
<td>function add() {</td>
<td></td>
</tr>
<tr>
<td></td>
<td>var a = 45 ;</td>
<td></td>
<td></td>
<td>var a = 19 ;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>with (this) { function x() { return a } ;</td>
<td></td>
<td></td>
<td>function x() { return a } ;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>cl1 = x ;</td>
<td></td>
<td></td>
<td>this . y = x ;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>eval(&quot;var a = 5&quot;);</td>
<td></td>
<td></td>
<td>eval(&quot;var a = 5&quot;);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>}</td>
<td></td>
<td></td>
<td>} add(); document.write( y() );</td>
<td></td>
</tr>
<tr>
<td></td>
<td>test1();</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>document.write(cl1() ) ;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>var cl2 = undefined ;</td>
<td></td>
<td>5</td>
<td>var obj = { a: 2, obj: { a: 3 } } ;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>function test2 () {</td>
<td></td>
<td></td>
<td>with (obj) {</td>
<td></td>
</tr>
<tr>
<td></td>
<td>var a = 45 ;</td>
<td></td>
<td></td>
<td>var f = function () {</td>
<td></td>
</tr>
<tr>
<td></td>
<td>function x() { return a } ;</td>
<td></td>
<td></td>
<td>{ return a ; }</td>
<td></td>
</tr>
<tr>
<td></td>
<td>with (this) { cl2 = x ;</td>
<td></td>
<td></td>
<td>}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>eval(&quot;var a = 5&quot;);</td>
<td></td>
<td></td>
<td>} add(); document.write( f() ) ;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>test2();</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>document.write(cl2()) ;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>var x = new Object ;</td>
<td></td>
<td>200</td>
<td>function add() {</td>
<td></td>
</tr>
<tr>
<td></td>
<td>x.a=100;</td>
<td></td>
<td></td>
<td>var a=20;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>20, 10(Mozilla)</td>
</tr>
<tr>
<td>Sl.</td>
<td>Scripts</td>
<td>Results</td>
<td>Sl.</td>
<td>Scripts</td>
<td>Results</td>
</tr>
<tr>
<td>-----</td>
<td>---------</td>
<td>---------</td>
<td>-----</td>
<td>---------</td>
<td>---------</td>
</tr>
<tr>
<td>88</td>
<td>x.c=10; function add(){ var a = 10; with (x){ eval(&quot;var a = 200; c=add1{return a;}&quot;); } } add(); document.write( x.c() );</td>
<td>var obj = {a:10, obj:{a:3}}; with (obj){ function add1(){ f = function () {return a;}; } y=add1; } add(); y(); actual=f(); document.write( actual );</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>var x = new Object ; x.a=100; x.c=10; function add(){ var a = 10; function add1(){return a}; with (x){ eval(&quot;c=add1&quot;); } } add(); document.write( x.c() );</td>
<td>var x; function add(){ function add1(){ eval(&quot;(function () { x= function(){ document.write(' add called'); mul(); });})();&quot;); } with(this){ add1(); } add(); x(); add = 5; function mul(){ document.write(&quot; mul called&quot;); }</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| 7.3 | **Structures**

**Function Object:** A function object upon deciding to retain the runtime stacks, which contains its local variables, parameters and other function declarations, of its outer functions being under execution, it should keep the runtime stacks in its own object even after the outer functions terminate. Later, while invoking this function as a closure function, the lookup for variables should take place along this list of runtime stacks.
**Function Structure:** A typical structure of the function object contains: a) Function Body, the AST nodes, b) Embedded function Object List.(EFO List). Fig: 7.1 demonstrate Function Object structure.

<table>
<thead>
<tr>
<th>Function Body (AST)</th>
<th>List of Embedded function Object</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Function Object</strong></td>
</tr>
</tbody>
</table>

**FIGURE: 7.1 Function Object**

**Embedded function Object List (EFO):**

When a function is going to terminate, it contains the list of all embedded objects inside the function. It contains: a) Runtime Data, b) Number of references to this EFO list, c) Function Block (Scope of the function of whose EFO list is made), d) Point to the next EFO list. Fig:7.2 depicts the EFO list structure.

<table>
<thead>
<tr>
<th>Runtime Data</th>
<th>Number of Reference to this EFO List</th>
<th>Function Block</th>
<th>Pointer to Next EFO List</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**FIGURE: 7.2 Embedded Function Object List**

**Runtime Data:** As shown in Table:7.1, at the time of execution, the function object can be assigned to a variable inside a “function”, “with object” or in “eval property”. Thus, the runtime data contain: a) With object/Function, b) Eval Stack List. Fig:7.3 represents the Runtime Data structure.

<table>
<thead>
<tr>
<th>With Object</th>
<th>Function Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>EVAL STACK LIST</strong></td>
</tr>
</tbody>
</table>

**FIGURE: 7.3 Runtime Data**
**Eval Stack List:** In JavaScript, the statements in the “eval” are evaluated at the runtime. When interpreter encounters the eval statement, it sends that string to the compiler, gets its symbol tables and the AST body of the “eval” statements. It contains: a) eval stack (offsets of variables), where the size of eval stack is set by the interpreter depending on the size of the operation stack used by an average function and the acceptable call depth; b) Eval symbol table; and c) Pointer to the next eval list (If the function has more eval statements). Fig:4 demonstrates EVAL list structure.

![EVAL Stack List Diagram](image)

**Function Data or With Object:** Depending on the closure behavior, it may be inside a function or inside the “with” object. For “with”, it is the object for function that contains a) Runtime Stack (function local variable with offset), where size of runtime stack is set by the interpreter depending on the size of the operation stack used by an average function and the acceptable call depth, b) Number of formal arguments, c) Number of actual arguments. Fig:7.5 represents function data.

![Function Data Diagram](image)
**Execution Stack:** For each function call, the execution context stack top (Fig.7.6) will be incremented and runtime data (Fig.7.3) will be created. Runtime data create function data (Fig.7.5) and it creates runtime stack based on number of arguments and local variables (e.g. Var a ; inside function). All these local variables declared and defined inside function will have the function scope. The runtime stack is a 4-byte signed integer type sequence. Whenever any symbol local to function is looked up, the value of the symbol will be looked up in the runtime stack. If the symbol is found, then the value will be returned or updated, else a new entry will be created into the global scope. The execution stack (Fig.7.6) will be destroyed at the end of function call when the function returns the function value.

![Execution Stack Diagram](image)

**FIGURE: 7.6 Execution Stack**

7.4 **Algorithm**

A number of algorithms with their functional behaviors are detailed below.

7.4.1 **Check for Embedded function Object:**

**Algorithm: 1 Update value from Right to Left Hand Side Variable**

```
PROCEDURE UPDATE_VALUE_FROM_RIGHT_TO_LEFT
ARG1: LHS, ARG2: RHS, ARG3: EXEC_STACK
{
  if (NOTE_EQUAL ( LHS, RHS )){
  
  }
```
FREE_VALUE ( LHS ) ;
COPY_VALUE ( RHS, LHS ) ;
if(IS_NOT_A_LOCAL_FUNCTION_VARIABLE ( LHS ) AND
   IS_A_FUNCTION_OBJECT ( LHS ) )
{
    CHECK_FOR_EFO (LHS, EXEC_STACK ) ;
}
}

Algorithm: 2 Check for Embedded function Object
PROCEDURE CHECK_FOR_EFO
ARG1: FN_OBJECT,
ARG2: EXEC_STACK
{
if ( EFO_LIST exist in FN_OBJECT )
{
    EFO_LIST->>RefEFOCount ++ ;
}
else
{
    CREATE_EFO_LIST ( FN_OBJECT , EXEC_STACK ) ;
}
}

While updating the variable from right hand side (RHS) to left hand side (LHS), the functional aim of this algorithm is to find out whether a function should behave as a closure function or not. The condition is to choose a function to behave as a closure must be implementation dependent. Upon deciding a function to behave as a closure, the algorithm will create Embedded Function Object list (Discussed in Section B) to append the necessary runtime stack information of its outer parent functions on to it. Upon successful addition of information, the function object will contain a valid member in its structure called “Embedded function object list”. This is the structure that contains all the required information pertaining to one execution level, its runtime stack, eval stack etc. So there could be a list of more than one such structures, listed depending upon the number of
functions (of parent child relation) being executed at that point of time. In case the function might be already carrying such a list on it, a counter keeping the count of such reference, ‘Number of Reference ‗, present in the Embedded function object list (Fig:7.2) structure, is incremented, without creating a new list. The reference value shows the number of embedded function objects assigned during the function execution. The reference counter will be reduced, whenever an object is deleted. The function object will be destroyed when the reference counter becomes 0. It will help to handle the memory garbage.

7.4.2 Create Embedded function Object List

Algorithm: CREATE EMBEDDED FUNCTION OBJECT

PROCEDURE CREATE_EFO_LIST ( FN_OBJECT , EXEC_STACK )
{
    EMBED_FN_SCOPE := SYM_GET_SYMBLOCK_PARENT( FN_OBJECT ) ;
    /*Embedding function's scope*/
    STACK_NO := GET_EXECSTACK_TOP( EXEC_STACK ) ;
    while( -- STACK_NO >= 0 && EMBED_FN_SCOPE )
    {
        /*Step #1: Find and index the private function's parent's scope in the function scope*/
        RUNTIME_DATA := GET_EXECSTACK_FROM_TOP( EXEC_STACK ) ;
        if ( IS_EXECSTACK_TOP_HAS_FUN_SCOPE( EXEC_STACK, STACK_TOP ) )
        {
            EVAL_LIST := GET_EVAL_LIST_FROM_RUNTIME_DATA
            (RUNTIME_DATA ) ;
            /*1.1 :If Eval scope present inside the function scope,Search across eval chain*/
            if ( EVAL_LIST != NULL )
            {
                EMBED_EVAL_SCOPE := GET_MATCH_EVAL_SCOPE
                ( EVAL_LIST, EMBED_FN_SCOPE ) ;
            }
            /*1.2 :If scope is not indexed yet, get the Stack No-level function object for match*/
            if ( EMBED_FN_SCOPE != EMBED_EVAL_SCOPE )
            {
                EMBED_EVAL_SCOPE := SIP_GET_FN_SYMBLOCK
                ( EXEC_STACK, STACK_TOP ) ;
            }
            /*1.3 :If indexed scope matches with the embedding scope, create EFO entry */
        }
    }
}
if ( EMBED_FN_SCOPE == EMBED_EVAL_SCOPE )
{
/*1.3.a: If matching function object already has EFO entry, inherit, else create new */
  FN_EFO_LIST := ALLOCATE_MEMORY_FOR_EFO_LIST ;
  if ( FN_EFO_LIST ) {
    if ( EFO_LIST ) {
      EFO_LIST->NEXT := FN_EFO_LIST ;
      EFO_LIST := EFO_LIST->NEXT ;
    }
    else {
      EFO_LIST := FN_EFO_LIST ;
    }
  }
  REFERENCE_NUMBER := GET_REFERENCE_NUMBER ( EFO_LIST ) ;
  REFERENCE_NUMBER := 0 ;
  EFO_LIST->RuntimeData := RUNTIME_DATA ;
  EFO_LIST->FnBlock := EMBED_FN_SCOPE ;
  EMBED_FN_SCOPE :=
    SYM_GET_SYMBLOCK_PARENT ( EMBED_FN_SCOPE ) ;
}
*/
/*Step #2: Find and index the private function referred(assigned)inside with object*/
else
{
  WITH_EFO_LIST = GET_NEW_EFO_LIST_FOR_WITH
    ( EXEC_STACK, RUNTIME_DATA ) ;
  if ( WITH_EFO_LIST )
  {
    if ( EFO_LIST )
    {
      EFO_LIST->NEXT := WITH_EFO_LIST ;
      EFO_LIST := EFO_LIST->NEXT ;
    }
  }
}
 else
  {
    EFO_LIST := WITH_EFO_LIST ;
  }
/*If WITH contains one or more eval, check if ‘vEmbedFnScope’, i.e parent function scope*/
EVAL_LIST := GET_EVAL_LIST_FROM_RUNTIME_DATA
           ( RUNTIME_DATA ) ;
if (EVAL_LIST != NULL )
  {
 if ( GET_MATCH_EVAL_SCOPE
           ( EVAL_LIST, EMBED_FN_SCOPE ) )
   {
     EMBED_FN_SCOPE :=
     SYM_GET_SYMBLOCK_PARENT
           ( EMBED_FN_SCOPE ) ;
   }
  }
SIP_SET_FN_EFOLIST( FN_OBJECT , EFO_LIST ) ;
}

This algorithm creates an entry for every execution layer (if any) in the execution stack, be
it for a function call or a ‘with’ object, if the outer function is parent of the inner function.
Hence, for every execution layer from top to empty, the child and parent relationship is
maintained by creating linked list. If the outer execution layer refers to a ‘with’ object, then
node entry is created with a slightly different contents with the help of algorithm ‘create with
object’ (Discussed in section C). Since an execution layer may refer to both a function call
and a ‘with’ object, the variable ‘Runtime Data’ to this structure, which in turn may
accommodate both function’s runtime stack in case of function layer, or the ‘with’ object if it
belongs to ‘with’ object layer. Structure” Runtime Data” also contains any eval stack present
inside that function or ‘with’ layer. Every time a function becomes closure, it keeps its
surrounding outer function runtime stacks. Hence, the same runtime stack information of
outer functions is shared if more than one inner function becomes closure.
7.4.3 Match Eval Scope
Algorithm: MATCH EVAL SCOPE
PROCEDURE GET_MATCH_EVAL_SCOPE
ARG1: EVAL_LIST,
ARG2: SCOPE_TO_MATCH
{
    while( EVAL_LIST )
    {
        EVAL_SCOPE := GET_EVAL_SCOPE ( EVAL_LIST ) ;
        if ( SCOPE_TO_MATCH == EVAL_SCOPE )
        {
            return EVAL_SCOPE ;
        }
        EVAL_LIST := EVAL_LIST->NEXT ;
    }
    return NULL ;
}

This algorithm is used to find out whether a scope is same as one of the eval scopes in a eval list. It will compare the input scope with the eval scopes present in the input eval handle.

7.4.4 Embedded Function Object list creates With Object
Algorithm: EFO CREATES IN WITH OBJECT
PROCEDURE GET_NEW_EFO_LIST_FOR_WITH
ARG1: RUNTIME_DATA
{
    EFO_LIST = ALLOCATE_MEMORY_FOR_EFO_LIST ;
    if ( EFO_LIST )
    {
        EFO_LIST->RuntimeData := RUNTIME_DATA ;
        EFO_LIST->bIsObject := E_TRUE ;
        EFO_LIST->RefEFOCount := 0 ;
        EFO_LIST->FnBlock := NULL ;
    }
}
This algorithm creates Closure (EFO) entry when the execution layer refers to a ‘with’ object.

7.4.5 Get Value from the Embedded Function Object list

**ALOG: GET VALUE FROM THE EMBEDDED FN OBJECT**

**PROCEDURE GET_VALUE_FROM_EFO_LIST**

ARG1: VARIABLE_NAME, ARG2: EFO_LIST

```c
while ( EFO_LIST ) {
    RUNTIME_DATA := EFO_LIST ->RunimeData;
    /**EFO INSIDE WITH OBJECT**/
    if ( E_TRUE == EFO_LIST ->bIsObject ) {
        /*1.1 Search in the Object*/
        if ( ! IS_GET_VALUE_FROM_WITH_OBJECT (RUNTIME_DATA, VARIABLE_NAME, &VARIABLE_VALUE ) ) {
            /*1.2 If not found above, check in the evalscope if present*/
            if ( ! IS_GET_VALUE_FROM_WITH_EVAL_LIST (RUNTIME_DATA, VARIABLE_NAME, &VARIABLE_VALUE ) ) {
                EFO_LIST := EFO_LIST -> NEXT;
            } else {
                return VARIABLE_VALUE;
            }
        } else {
        }
    }
}
return EFO_LIST;
return NULL;
```
This algorithm is used to look up a variable in the Embedded Function Object list of a function object. It searches in all the entries in the list and returns the value if found, unsuccessful report otherwise. This algorithm will search every Embedded Function Object entry depending upon whether it is a “function call” entry or ‘with’ entry, by handling it in

```c
return VARIABLE_VALUE;
}
}
else /**EFO INSIDE WITH FUNCTION SCOPE**/
{
/**1.1 First Check In Eval Scope/Scopes */
if ( ! IS_GET_VALUE_FROM_FUNCTION_EVAL_LIST (RUNTIME_DATA, VARIABLE_NAME, &VARIABLE_VALUE) )
{
/*1.2 Search in the Function scope present in the EFO entry */
EMBED_FN_SCOPE := EFO_LIST->FnBlock;
if ( ! IS_GET_OFFSET_VALUE_FROM_EMBEDDED_SCOPE (EMBED_FN_SCOPE, VARIABLE_NAME, &OFFSET_VALUE) )
{
EFO_LIST := EFO_LIST -> NEXT;
}
else {
    VARIABLE_VALUE := GET_VALUE_FROM_EFO (RUNTIME_DATA, OFFSET_VALUE);
    return VARIABLE_VALUE;
}
}
else {
    return VARIABLE_VALUE;
}
}
return NULL;
}
```
separate ways, and goes to next entry if not found. For a function call Embedded Function Object entry, the algorithm is discussed in section E to retrieve the value.

7.4.6 Get Value from the Embedded Function Object entry

**ALOG: GET VALUE FROM THE EMBEDED FN OBJECT**

**PROCEDURE GET_VALUE_FROM_EFO**

ARG1: RUNTIME_DATA , ARG2: OFFSET_VALUE

{
/*Get Variable value in the function runtime stack*/
RUNTIME_STACK := GET_CURRENT_RUNTIMENTSTACK ( RUNTIME_DATA ) ;
VARIABLE_VALUE := RUNTIME_STACK + OFFSET_VALUE ;
return VARIABLE_VALUE ;
}

This algorithm is used to get a variable in a function from its runtime stack present in the Embedded Function Object entry structure. It does not search, but retrieves a value from the runtime stack with the help of offset.

7.5 Conclusion

During the execution of the function, the values of the local variables are stored in to the runtime stack according to their offset values from the symbol table. This algorithm will keep the runtime stack into the function object while accounting closure property during the execution. Here the reference counter plays the crucial role to handle the memory garbage. The reference counter will increase when an embedded function object assigned to the variable and will be reduced when an object is deleted. The runtime stack will be deleted when the reference count reduced to 0. We have tested and verified this algorithm with top10 Alexa web-sites in different mobile devices. It executes all the scripts of the web-sites without blocking any mobile operation. We have ported, tested and verified our script engine with low end devices such are Moto RAZR v3 (brew 3.15), Qtopia (Linux OS), Samsung (Windows) and Nokia Series (Symbian OS ). For looking up a symbol, it is required to lookup first into the local symbol table and then eval scope or with object and global symbol table. In future, this can be optimized further and execution time can be reduced significantly.