Background

2.1 JavaScript, its Behaviors and Challenges

JavaScript the “the World's Most Misunderstood Programming Language” is a cross-platform, object-oriented and one of the most popular dynamic scripting language as of today. JavaScript is a small, lightweight language; it is not useful as a standalone language, but is designed for easy embedding in other products and applications, such as web browsers. Inside a host environment, JavaScript can be connected to the objects of its environment to provide programmatic control over them. It accommodates a variety of programming paradigms allowing to mixing C-like procedural, object-oriented and functional programming styles.

JavaScript was developed by Brendan Eich in 1995 at Netscape in reaction to Microsoft’s inclusion of Visual Basic into the Internet Explorer for client-side scripting support. It was first released with Netscape Navigator 2.0 in 1995. While it was initially named Live script, the name was changed to JavaScript for marketing reasons [MA98]. In June 1998 Netscape and Sun announced together Netscape 2.0 with JavaScript as a "complement" to Java [net95], for which support was introduced in this version as well. There is no technical relation between those two languages (also there exists a JavaScript API to interact with Java applets) other than the marketing reasons, despite the name suggesting otherwise. In November 1996 Netscape submitted JavaScript to ECMA International for standardization, which resulted in the ECMAScript-262 standard [6] in 1997. The resulting ECMA standard was in turn submitted to the ISO/IEC and became the ISO 16262 standard by April 1998.

While the 2nd edition of ECMA-262 only ported the changes from the ISO standardization back into ECMA-262, the 3rd Edition, published in December 1999, added new features to the language. The planned 4th edition would have included a lot of new features and changes, but was never published due to differences of opinion. Some of the ideas will be discussed by the ECMA Script Harmony project. The current 5th edition was published in December 2009, clarifying some points of the 3rd edition and including a strict mode as well as adding the JSON data exchange format and allowing for more introspection. With the standardization by ECMA, JavaScript has become a dialect of ECMA Script, which
is maintained by the Mozilla Foundation and tailored towards a practical use in web-browsers to support dynamic web-pages.

With the spread of AJAX around 2005 and the increased usage of JavaScript to create rich web-site experiences arose the need for faster JavaScript execution in the browser. This started a competition between browser manufactures for a faster and more feature-rich JavaScript engine often utilizing some form of Just-In-Time compilation.

Spider Monkey is based on the original JavaScript engine by Brendan Eich, written in C/C++ and is the current JavaScript engine of Mozilla’s Firefox web-browser. While it is based on the original design, it has been improved over time and assimilated several JIT compilers which started of as separate projects. The first JIT was called Trace Monkey (included in Firefox 3.5), a tracing JIT which recorded the operations during interpretation produce optimized machine code for later execution. It was superseded in Firefox 4 by JägerMonkey, a method JIT that compiles the internal bytecode representation generated by SpiderMonkey. While JägerMonkey replaced TraceMonkey, the tracing capabilities are still available and used for optimizing loops.

The V8 JavaScript engine does not interpret the code, but rather compiles it into platform specific machine code. The dynamic types in JavaScript make it necessary to patch the generated machine code since accessing JavaScript data structures heavily relies on inline-caching accessor code that was tailored for a specific object-layout.

Over the time JavaScript has been included in and adapted to many different projects (eg. node.js [9], Qt Script for applications [10], Macromedia’s Action Script[11]), and it is not exclusively used for web-pages anymore. Most of them are interpreting JavaScript, but some are already compiling JavaScript directly to JVM byte code (eg. Mozilla’s Rhino [12] and Caucho Resin [13]). JavaScript is not easy to compile though. Several of its properties make it challenging to generate efficient code:

**Dynamic Typing:** As in most scripting languages, types are associated with values, not variables. For example, a variable x could be bound to a number as “var x=10”, and then later rebound to a string as “x=comviva”. JavaScript supports various ways to test the type of an object, including duck typing. It is illegal in statically-typed language.
In static type every variable name is bound, both to a type (at compile time) and to an object, where binding to an object is optional. In a dynamically typed language every variable name is (unless it is null) bound only to an object [14] (Fig: 1.1).

Objects as associative arrays: JavaScript is almost entirely object-based. Object property names are associative array keys: obj.x = 10 and obj["x"] = 10 are equivalent, the dot notation being merely syntactic sugar. Properties and their values can be added, changed, or deleted at run-time. The properties of an object can also be enumerated via a ‘for...in loop’.

**Run-time Evaluation:** JavaScript, like many dynamic languages before it, makes it strikingly easy to turn text into executable code at runtime. The language provides the eval function for this purpose. While eval and other dynamic features are the strength of JavaScript, as attested to by their widespread use, their presence is a hindrance to anyone intent on providing static guarantees about the behavior of JavaScript code [15]. The statement “var x=10;” compiled at compilation time where as eval (“var x=10”) compiled at run-time.

**First-class functions:** Functions are first-class objects; they are objects themselves. As such, they have properties and can be passed around and interacted with like any other object [16, 17]. Using “instance of” operator we can instantiate the function object. Properties can be added like other objects by DOT “.” operator. Function can be used as a variable. It can be assigned to other variables, passing as a parameter to other function and return from a function.
**Functions as object constructors:** Functions as object constructors along with their typical role. An object constructor is merely a regular JavaScript function, so it's just as robust (i.e.: define parameters, call other functions etc). The difference between the two is that a constructor function is called via the `new` operator [18]. Prefixing a function call with “new()” creates a new object and calls that function with its local this keyword bound to that object for that invocation. The function's prototype property determines the new object's prototype.

**Prototype-based:** In a class-based object-oriented language, in general, state is carried by instances, methods are carried by classes, and inheritance is only of structure and behaviour. In JavaScript, the state and methods are carried by objects, and structure, behaviour, and state are all inherited. All objects that do not directly contain a particular property that their prototype contains share that property and its value. The following diagram (Fig: 1.2) illustrates this:

![Diagram](image)

*Figure: 2.2 Prototype based Inheritance*

CF is a constructor (and also an object). Five objects have been created by using new expressions: cf1, cf2, cf3, cf4, and cf5. Each of these objects contains properties named q1 and q2. The dashed lines represent the implicit prototype relationship; so, for example, cf3’s prototype is CFp. The constructor, CF, has two properties itself, named P1 and P2, which are not visible to CFp, cf1, cf2, cf3, cf4, or cf5. The property named CFp1 in CFp is shared by cf1, cf2, cf3, cf4, and cf5 (but not by CF), as are any properties found in CFp’s implicit prototype chain that are not named q1, q2, or CFp1. Notice that there is no implicit prototype link between CF and CFp. Unlike class-based object languages, properties can be added to objects dynamically by assigning values to them. That is, constructors are not required to name or assign values to all or any of the constructed object’s properties. In the above...
diagram, one could add a new shared property for cf1, cf2, cf3, cf4, and cf5 by assigning a new value to the property in CFp.

Prototype-based programming is a style of object-oriented programming in which classes are not present, and behavior reuse (known as inheritance in class-based languages) is performed via a process of cloning existing objects that serve as prototypes. This model can also be known as classless, prototype-oriented or instance-based programming. Delegation is the language feature that supports prototype-based programming [19].

**Inner functions and closures:** Inner functions (functions defined within other functions) are created each time the outer function is invoked, and variables of the outer functions for that invocation continue to exist as long as the inner functions still exist, even after that invocation is finished (e.g. if the inner function was returned, it still has access to the outer function's variables) this is the mechanism behind closures within JavaScript [20]. The more of this is explained in chapter 5.

**Run-time environment:** JavaScript typically relies on a run-time environment (e.g. in a web browser) through AJAX scripts to interact with "the outside world".

**Variadic functions:** An indefinite number of parameters can be passed to a function [21,22]. The function can both access them through formal parameters and the local arguments object.

**Array and object literals:** There is very simple way to declare Array and Object though literals. An object literal is a comma separated list of name value pairs wrapped in curly braces like “var myObject = {sProp: 'some string value', numProp: 2, bProp: false};;”. Object literals are used as a means of encapsulating data, enclosing it in a tidy package to minimize the use of global variables [23].

Arrays in JavaScript, as most other things in the language, are objects. They can be created with the built-in constructor function Array(), but they also have a literal notation and, just like the object literal “var a = ["itsy", "bitsy", "spider"]”, the array literal notation is simpler and preferred [24].
Regular expressions: JavaScript also supports regular expressions in a manner similar to Perl, which provide a concise and powerful syntax for text manipulation that is more sophisticated than the built-in string functions. Regular expressions are patterns used to match character combinations in strings. In JavaScript, regular expressions are also objects. These patterns are used with the `exec` and `test` methods of regular expression, and with the `match`, `replace`, `search`, and `split` methods of string [25,26]. The pattern “/d(b+)d/” to search one or more “b” within “d” from a string ” cdbbdbsbz “ can be represented as ”var myRe = /d(b+)d/g; var myArray = myRe.exec("cdbbdbsbz");”

AJAX: Asynchronous JavaScript and XML, or AJAX, is a group of interrelated web development techniques used for creating interactive web applications or rich Internet applications [27,28]. With Ajax, web applications can retrieve data from the server asynchronously in the background without interfering with the display and behavior of the existing page.

Frame: Most of the real world applications involve multiple frames or multiple windows. Frames within a window are represented by window objects. Interestingly in these applications, there are JavaScript codes that run independently in each of such windows or frames [29]. These independent JavaScript codes of one window can interact and co-operate with each of the other windows.

Event Handling: Most JavaScript-applications perform actions as a response to events [30, 31, 32]. Event handler is an asynchronous callback subroutine that handles inputs received in a program. Each event is a piece of application-level information from the underlying framework, typically the GUI toolkit. GUI events include key presses, mouse movement, action selections, and timers expiring. Events are the beating heart of any JavaScript application.

Set Timeout/ Set Interval: With JavaScript, it is possible to execute some code NOT immediately after a function is called, but after a specified time interval [33, 34]. This is called timing events. It’s very easy to time events in JavaScript. The key methods that are used are “setTimeout()” to executes a code some time in the future, “clearTimeout()” to cancels the setTimeout() and “setInterval()” to execute a function in periodically
**Garbage Collection:** Since the Objects created are not explicitly destroyed by the programmer, as in other Object oriented languages, an effective garbage collection algorithm plays a vital role in destroying the objects when they are not referred by other variables. There are two algorithms used generally Mark-and-Sweep Garbage Collection and Garbage Collection by Reference Counting [35, 36, 37].

**Mark-and-Sweep Garbage Collection:** A mark-and-sweep garbage collector periodically traverses the list of all variables in the JavaScript environment and marks any values referred to by these variables. If any referenced values are objects or arrays, it recursively marks the object properties and array elements. By recursively traversing this tree or graph of values, the garbage collector is able to find (and mark) every single value that is still reachable. It follows, then, that any unmarked values are unreachable and are therefore garbage [38, 39].

Once a mark-and-sweep garbage collector has finished marking all reachable values, it begins its sweep phase. During this phase, it looks through the list of all values in the environment and de-allocates any that are not marked. Classic mark-and-sweep garbage collectors do a complete mark and a complete sweep all at once, which causes a noticeable slowdown in the system during garbage collection. More sophisticated variations on the algorithm make the process relatively efficient and perform collection in the background, without disrupting system performance.

**Collection by Reference Counting:** When an object is created and a reference to it is stored in a variable, the object's reference count is one [40]. When the reference to the object is copied and stored in another variable, the reference count is incremented to two. When one of the two variables that hold these references is overwritten with some new value, the object's reference count is decremented back to one. If the reference count reaches zero, there are no more references to the object. Since there are no references to copy, there can never again be a reference to the object in the program. Therefore, JavaScript knows that it is safe to destroy the object and garbage collect the memory associated with it.

**Terminating Script:** The script could be terminated by the consumer at any instant of time depending on the requirement. The currently executing script would be terminated and the control would be given back to the consumer.
2.2 Compilation and Optimization

Earlier we made mention of what is called a compiler, and in particular a compiler for an embedded device. In this section we develop these terms into a description of what a compiler is and does, and what we mean by optimizing.

What is a Compiler? In the sense of a compiler being a person who compiles, then the term compiler has been known since the 1300’s. Our more usual notion of a compiler—a software tool that translates a program from one form to another form—has existed for little over half a century. For a definition of what a compiler is, we refer to Aho et al [41]:

A compiler is a program that reads a program written in one language—the source language—and translates it into an equivalent program in another language—the target language.

Early compilers were simple machines, that did little more than macro expansion or direct translation; these exist today as assemblers, translating assembly language (e.g., “add r3,r1,r2”) into machine code (“0xE0813002” in ARM code).

Over time, the capabilities of compilers have grown to match the size of programs being written. However, Proebsting [42] suggests that while processors may be getting faster at the rate originally proposed by Moore [43], compilers are not keeping pace with them, and in-deed seem to be an order of magnitude behind. When we say “not keeping pace” we mean that, where processors have been doubling in capability every eighteen months or so, the same doubling of capability in compilers seems to take around eighteen years!

Which then leads to the question of what we mean by the capability of a compiler. Specially, it is a measure of the power of the compiler to analyse the source program, and translate it into a target program that has the same meaning (does the same thing) but does it in fewer processor clock cycles (is faster) or in fewer target instructions (is smaller) than a native compiler. Improving the power of an optimizing compiler has many attractions:

Increase performance without changing the system Ideally, we would like to see an improvement in the performance of a system just by changing the compiler for a better one, without upgrading the processor or adding more memory, both of which incur some cost
either in the hardware itself, or indirectly through, for example, higher power consumption.

**More features at zero cost.** We would like to add more features (i.e., software) to an embedded program. But this extra software will require more memory to store it. If we can reduce the target code size by upgrading our compiler, we can squeeze more functionality into the same space as was used before.

**Good programmers know their worth.** The continual drive for more software, sooner, drives the need for more programmers to design and implement the software. But the number of good programmers who are able to produce fast or compact code is limited, leading technology companies to employ average-grade programmers and rely on compilers to bridge (or at the very least, reduce) this ability gap.

**Same code, smaller/faster code.** One mainstay of software engineering is code reuse, for two good reasons. Firstly, it takes time to develop and test code, so re-using existing components that have proven reliable reduces the time necessary for modular testing. Secondly, the time-to-market pressures mean there just is not the time to start from scratch on every project, so reusing software components can help to reduce the development time, and also reduce the development risk. The problem with this approach is that the reused code may not achieve the desired time or space requirements of the project. So it becomes the compiler's task to transform the code into a form that meets the requirements.

**CASE in point.** Much of today's embedded software is automatically generated by computer-aided software engineering (CASE) tools, widely used in the automotive and aerospace industries, and becoming more popular in commercial software companies. They are able to abstract low-level details away from the programmers, allowing them to concentrate on the product functionality rather than the coding loops, state machines, message passing, and so on. In order to make these tools as generic as possible, they typically emit C or C++ code as their output. Since these tools are primarily concerned with simplifying the development process rather than producing fast or small code, their output can be large, slow, and look nothing like any software that a programmer might produce.

In some senses the name optimizing compiler is misleading, in that the optimal solution is rarely achieved on a global scale simply due to the complexity of analysis. A simplified model of optimization is:
Optimization = Analysis + Transformation:
Analysis identifies opportunities for changes to be made (to instructions, to variables, to the structure of the program, etc); transformation then changes the program as directed by the results of the analysis.

**Intermediate Codes Representation**: The usual steps (Fig. 1.3) for compilation are:

a) Parse source code into a parse tree
b) Transform parse tree into an Abstract Syntax Tree
c) Transform AST into a Control Flow Graph
d) Emit bytecode based on the Control Flow Graph

---

**Modular Compilers**

![Diagram of Modular Compiler](image)

Figure : 2.3 Intermediate Representation of Modular Compiler

Lexer (Lex), Parser (Yacc) and Symbol Table: The first phase in a compiler reads the input source and converts strings in the source to tokens. Lex generates C code for a lexical analyzer, or scanner. It uses patterns that match strings in the input and converts the strings to tokens. Tokens are numerical representations of strings, and simplify processing. This is illustrated in Figure 1.3. Each pattern in lex has an associated action. Typically an action returns a token, representing the matched string, for subsequent use by the parser.

As lex finds identifiers in the input stream, it enters them in a symbol table. The symbol table may also contain other information such as data type (integer or real) and
location of the variable in memory. All subsequent references to identifiers refer to the appropriate symbol table index.

Yacc generates code for a syntax analyzer, or parser. Yacc uses grammar rules that allow it to analyze tokens from lex and create a syntax tree (AST). A syntax tree imposes a hierarchical structure on tokens [44].

![Compilation Sequence Diagram](image)

Figure: 2.4 Compilation Sequence

Abstract Syntax Trees (AST): The abstract syntax tree (AST) is a high-level representation of the program structure without the necessity of containing the source code; it can be thought of as an abstract representation of the source code. The specification of the AST nodes is specified using the Zephyr Abstract Syntax Definition Language (ASDL)[45].

Each AST node (representing statements, expressions, and several specialized types, like list comprehensions and exception handlers) is defined by the ASDL. Most definitions
in the AST correspond to a particular source construct, such as an ‘if” statement or an attribute lookup. The definition is independent of its realization in any particular programming language.

**Abstract Syntax Description Language**

![Diagram](image)

**FIGURE: 2.5 Abstract Syntax Description Language**

**AST to CFG to Bytecode**: With the AST created, the next step is to create the Control Flow Graph. The first step is to convert the AST to bytecode without having jump targets resolved to specific offsets (this is calculated when the CFG goes to final bytecode). Essentially, this transforms the AST into bytecode with control flow represented by the edges of the CFG.

Conversion is done in two passes. The first creates the namespace (variables can be classified as local, free/cell for closures, or global). With that done, the second pass essentially flattens the CFG into a list and calculates jump offsets for final output of bytecode.

It has been observed that in JavaScript many of the scripts are compiled but not executed at runtime. It is an overhead to the compiler and will be consumes a lot memory and power of the embedded devices. Looking to this in our JavaScript engine we make an AST based interpreter rather than popular bytecode base. Appendix-II shows the comparisons of bytecode against the AST node.

**Intermediate Codes Optimization**: Optimizations applied at the intermediate code level are appealing for three reasons:
1. Intermediate code statements are semantically simpler than source program statements, thus simplifying analysis.
2. Intermediate code has a normalizing effect on programs: different source code produces the same, or similar, intermediate code.
3. Intermediate code tends to be uniform across a number of target architectures, so the same optimization algorithm can be applied to a number of targets.

This thesis presents the optimizations for compacting embedded systems target code. It optimizes the AST nodes of various JavaScript statements if-then-else, function-call and the dot operators. It also optimized vastly used identifier nodes.

2.3 **Overview of JavaScript Engine**

The Script Engine provides the execution environment for web scripts. The various operations performed by Script Engine includes buffering script source chunks, analyzing and parsing the script source, interpreting the parsed data and providing the results of the execution. Different components in the Script Engine are responsible for performing the mentioned operations.

The following figure (1.6) shows the various components of the script Engine and their relationship.

**Figure: 2.6 Script Engine**

- **A** - Script Source code.
- **B** - Parsed Abstract Syntax Tree and Symbol Table.
- **C** - Results obtained after evaluating the Abstract Syntax Tree (AST).
- **D** - Request for common objects operations.
- **E** - Results of common objects operations.
This section gives a brief description of the components involved, which shall assist in easier understanding of the subsequent topics.

2.3.1 Script Compiler [SCR]:

The Script Compiler takes the source script and compiles the source for any syntax errors and reports for the same to the consumer if any. On successful compilation of the source, Script Compiler generates AST and Symbol table. The consumer uses the AST and the Symbol table to perform execution of the AST. During the process, the SCR interacts with the consumer when the consumer requires interruption of the compilation process. The SCR contains four subcomponents namely, Lexical analyzer, Syntactic Analyzer, AST and the Symbol table manager.

The following figure (1.7) provides a pictorial view of SCR.

Lexical Analyzer [LA]: The Script Compiler contains a unit called the Lexical Analyzer that interacts with the Syntax Analyzer or parser and provides the tokens when requested. The LA generates tokens for input buffer passed provided by the parser.
When the input is incomplete, LA waits for additional input after tokenizing the buffer obtained till then. Whenever an error occurs during the process, the lexical analyzer reports appropriate error to the syntax analyzer.

**Syntax Analyzer:** The Script Compiler contains a unit called the Syntax analyzer and this component is responsible for parsing the script source and validating against the grammar rules of the language. It interacts with LA when it requires tokens for an input. The validation against the grammar rules of the language leads to actions that include creating an Abstract Syntax tree for the source script and validating the tokens in the Symbol table. Creation of the Symbol Table is done at this point.

**Abstract Syntax Tree:** The Script Compiler converts the script code into an equivalent representation called Abstract Syntax Tree (AST). The AST is a structured tree format of the source script containing different nodes that represent different constructs of the language and the references to the symbol table entries. The process of creation of AST involves interactions with the Symbol Table Manager for creating symbol table and updating information in it. AST and Symbol table are used by consumer to perform evaluation.

**Symbol Table Manager:** The Script Compiler creates a symbol table using Symbol Table Manager. The Symbol table manager provides an interface to the consumer in order to store the data pertaining to a symbol and whenever a lookup is required an appropriate reference to the symbol table entry is returned back to the caller. The Symbol Table Manager is responsible for managing the storage of the symbols encountered during the process of compilation and interpretation of the scripts.

**String Table:** The String table contains the names of the identifiers that are used in the source script. The reference to a string is maintained in a hash table and the reference is used by other modules to access identifier names. The identifier names can belong to host object names, Built in object names and user defined names in the scripts.

The names are stored separately depending on the scope of usage ie. User defined names (or session/temporary names) are referred from the time they are declared but the built in objects and host object names (or permanent names) can be referred anytime. Hence the temporary names are stored separately than the permanent names by using different structures.
2.3.2 Script Common Objects [SCO]

This component provides all the necessary services to perform the operations related to the common objects like number, string, math, Boolean etc (Figure 1.7). The services could range for simple type conversion facilities to manipulating the objects.

![Diagram of Script Common Objects (SCO)](image)

Figure 2.8 : Script Common Object (SCO)

2.3.3 Script Interpreter [SIP]

This component evaluates the AST with the help of symbol table information and performs the appropriate actions described in the script source.

The main functions of SIP is as follows:

- Interprets the AST (Abstract Syntax Tree) generated by the Script Compiler, by following operations
  - Non recursively Traverses the IST (Interpretive Syntax Tree (AST+ST)) in appropriate order
  - Evaluate the AST Nodes/Sub trees using a stack in sync with the Symbol table and Scope information
  - Fires execution commands for the Consumer
- Handles event from the Consumer (Figure 1.8)
The Script Interpreter works with the Interpretive Syntax tree i.e. we can say annotated Abstract Syntax Tree (AST) with Symbol Table (ST) information. The IST is optimized for efficient traversal while interpretation and is perfectly semantically checked.

Choice of Interpretation method: While designing an interpreter for a language like Java Script, we have two choices for script interpretation approach.

a) Byte Code based Interpretation (popular)
b) AST based Interpretation

a) **Byte Code based Interpretation**: This requires the either a VM (Virtual Machine) support from the OS/Browser or the Interpreter is designed to execute byte codes in sequence. The Script Compiler generates the Byte Code sequence for the target Interpreter from the AST and the interpreter executes that in the sequence.

b) **AST based Interpretation**: The AST node will generate after lexical analysis followed by syntax analysis of the source. Along with the Symbol Table, the AST depicts the complete structural, runtime view of the AST. The Interpreter thus traverses the AST and tries to evaluate/execute the different nodes/sub-trees of AST.

**Choice of Interpretation**: The Script Interpreter for Script Engine will be an AST based. Following are the factors that stimulate in arriving at this decision (APPENDIX-II). It is easy
to implement, partly because AST semantics are the same as the source language script engine do not run on a platform integrated with a VM. AST representation can be customized to suit the Interpreter for better performance.

**Stack based interpretation:** The choice of stack-based interpretation comes not out of choice but out of compulsion. For a typical platform like feature phone where stack size and memory available are low, features like recursion are proscribed. Since AST based Interpretation is chosen, due to vary recursive nature of the AST, its traversal is going to be inherently recursive. But since use of recursion is out of scope, therefore the ultimate decision would be to emulate the recursive behavior using a set of stacks (APPENDIX-I).

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 is refers to 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.

Again, traversing the AST will be a typical post-order traversal, which also must be implemented without recursion, for that we may use a stack, which we will call as Instruction Stack. Also we need to save the Environment or say Execution Context in typical compiler language, which gives the current state of the interpretation and other details. As we move from one execution context to other we may require to push them one after the other in a stack called Execution Stack, so that we can come back to the previous execution context with a mere pop.

2.3.4 **Script Engine Controller [SEC]**  
This component regulates all other components in the Script Engine. It co-ordinates and controls the data flow between the Script Engine components. It acts as the main interface to the consumer.