AWIPS II EDEX Training Resource Material
Writing a MicroEngine Script

Prepared in Support of the AWIPS Software
Continuous Technology Refresh Re-Architecture,
Task Order T1

Document No. AWP.TRG.SWCTR/TOT1-01.00
28 February 2008

Prepared Under

Contract DG133W-05-CQ-1067
Advanced Weather Interactive Processing System (AWIPS)
Operations and Maintenance

Prepared by:

Raytheon Technical Services Company LLC
8401 Colesville Road, Suite 800
Silver Spring, MD 20910

This document includes data that shall not be duplicated, used, or disclosed – in whole or in part – outside the Government for any purpose other than to the extent provided in contract DG133W-05-CQ-1067. However, the Government shall have the right to duplicate, use, or disclose the data to the extent provided in the contract. This restriction does not limit the Government's right to use information contained in this data if it is obtained from another source without restriction. The data subject to this restriction are contained in all sheets.

Table of Contents

Page

1. Objective 1
2. Description 1
3. The µEngine and JavaScript 2
4. JavaScript Scripting Basics 2
5. µEngine Script Structure 4
6. The TermQuery Task: 5
7. A µEngine scripting Example 6
   7.a. Script Analysis 6
   7.b. The Code: 7
   7.c. Comments 7
8. JavaScript Classes 8
   8.a. Class Prototypes 9
   8.b. Function Instances 9
   8.c. Class Constructors 10
   8.d. A Hello World Example 10
   8.e. The µEngine's JavaScript Library 11
9. A µEngine Script using Classes 12
10. Additional Considerations 13
11. Known Issues 13
Appendix A. References A-1
Appendix B. Acronym List B-1

List of Figures

Page

Figure 1. µEngine Data Transformation 1
Figure 2. µEngine Script Building Blocks 2
Figure 3. µEngine Information Flow 3

List of Tables

Page

Table 1. Generic Steps in a µEngine Script 4
Table 2. µEngine script Tasks for each script step 4

Use or disclosure of data contained on this sheet is subject to the restriction on the title page of this document.

1. Objective

This document outlines the basic procedure for writing a MicroEngine (µEngine) script. The µEngine script is implemented as a JavaScript class file, which may be used by client applications to process data.

2. Description

The Environmental Data Exchange (EDEX) provides the server functionality of the AWIPS Development Environment (ADE). It consists of two parts, a static runtime environment and modifiable software – the AWIPS II EDEX – which is based on the design patterns of the Service Oriented Architecture (SOA). Also included is the µEngine Web, which is a set of Apache Tomcat-based Web pages that are available for EDEX development testing and demonstration. For development purposes, the ADE supports building and installation of the EDEX software, as well as the µEngine demo, on a development computer.

Within the AWIPS II EDEX, the µEngine provides a scripting capability for transforming raw input data into various displayable products. The actual transformation is performed by one of the EDEX services, e.g., the Product Service. Figure 1 provides a basic view of this data transformation process.

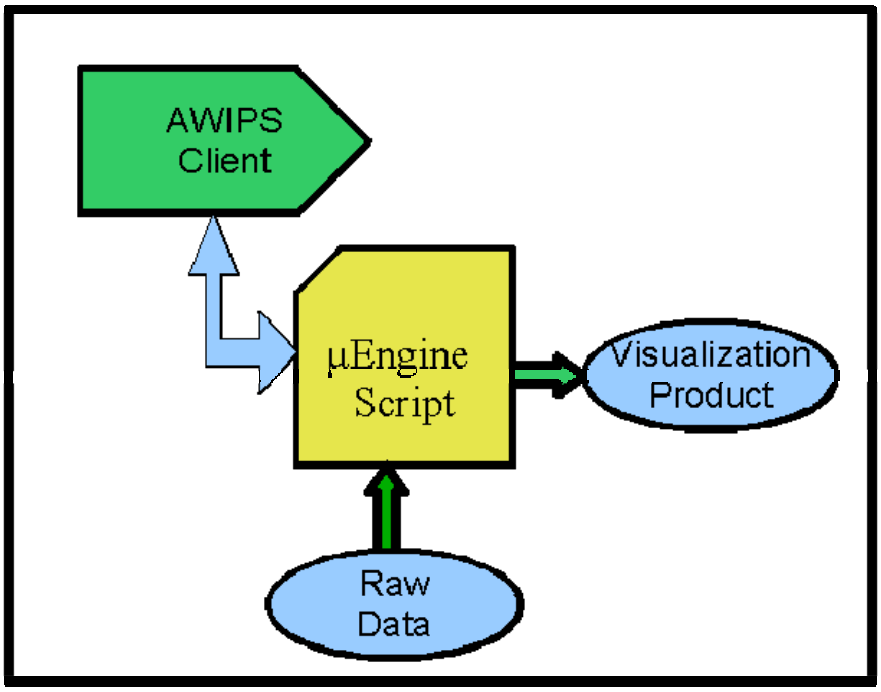

Figure 1. µEngine Data Transformation

A µEngine script is written in JavaScript and represents a series of tasks. Each task is represented as a µEngine Task object, which is written in Java. (Creation of µEngine scripts is covered in a separate document. See Appendix A, References.)

AWIPS II EDEX supports a multi-tiered approach to creating µEngine scripts as shown in

Use or disclosure of data contained on this sheet is subject to the restriction on the title page of this document.

Figure 2.

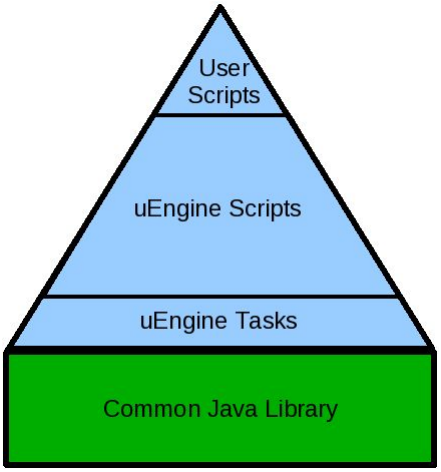

Figure 2. µEngine Script Building Blocks

User scripts are short scripts that are normally generated by a client application such as the Common AWIPS Visualization Environment (CAVE) based on user inputs. µEngine scripts are library scripts, normally defining JavaScript classes, and are referenced from within user scripts. User scripts and µEngine scripts are written in JavaScript. The AWIPS II ADE includes source code for a number of user scripts and µEngine scripts. µEngine Tasks are Java classes that form a bridge between the JavaScript µEngine scripts and the underlying Java code in the EDEX libraries. (µEngine Tasks are discussed in detail elsewhere – see the reference list in Appendix A.)

The remainder of this document is devoted to a discussion of writing JavaScript µEngine scripts. The primary focus is on writing the JavaScript classes that make up the µEngine script library.

3. The µEngine and JavaScript

JavaScript is a Java-like scripting language originally developed by Netscape for the (now defunct) Netscape browser. It was first released with Netscape version 2. Rhino is an Open Source implementation of JavaScript written in Java that does not require a Web browser to execute JavaScript scripts. Java 6 from Sun includes JSR 223, which specifies a Java scripting interface for script writers and includes a version of Rhino as a demonstration scripting engine for Java. For this reason, AWIPS II utilizes JavaScript as the scripting language for µEngine scripts. It uses the Rhino scripting engine interface rather than JSR 223, but the details of interfacing with JavaScript are hidden inside the µEngine and are beyond the scope of this paper.

4. JavaScript Scripting Basics

JavaScript is an object-based scripting language with syntax similar to Java. (For a complete discussion of JavaScript programming, see the list of resources in Appendix A.) It includes common iteration and decision structures as well as modularity via user-defined functions. Conceptually, the flow of information in the µEngine is shown in Figure 3.

Use or disclosure of data contained on this sheet is subject to the restriction on the title page of this document.

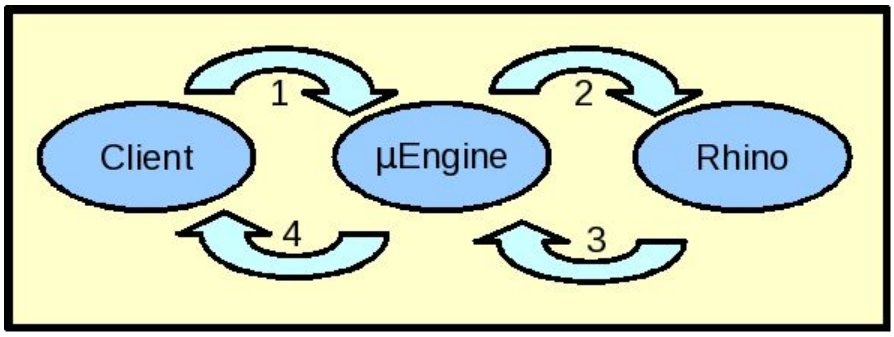

Figure 3. µEngine Information Flow

As illustrated in Figure3, there are 4 steps in the µEngine information flow

1. The client creates a JavaScript µEngine script and passes it to the µEngine. The µEngine performs certain preprocessing on the script.
2. The µEngine creates a Rhino script runner and passes the script to the script runner, which executes the script.
3. Following script execution, the Rhino script runner returns the result of the execution to the µEngine.
4. The µEngine packages the results into an XML format and returns the result to the client.

A few characteristics of JavaScript, as implemented by Rhino and the µEngine, must be considered when writing µEngine scripts.

1. Rhino returns the last value of the last executed global statement of the script. The µEngine captures this return value and returns it to the user.
2. The µEngine preprocesses any JavaScript it receives. This preprocessing involves two separate elements. First, the µEngine resolves any include statements, merging the included code into the script. This allows for modularization of scripts. In addition, the µEngine performs minor modifications to certain scripts so that they will work with the EDEX subscription service.
3. Like most scripting languages, JavaScript is loosely typed. In particular, JavaScript does not distinguish between the various numeric types available in Java. The main implication here is that you cannot call overloaded Java methods from JavaScript if the only difference in the method footprint is the type of the arguments.
4. JavaScript does not support arrays of primitives such as a float array (float[]). JavaScript does have an Array type, but it generally acts like a combination of a java.util.List and a java.util.Map. Because of this, any Java method that returns an array structure that will be processed in JavaScript must return the array as a java.util.List based structure.
5. JavaScript treats all Java Objects as objects, which allows a µEngine script to pass an object from a Java class (µEngine Task) through JavaScript to another Java class (µEngine Task). Because Java treats arrays of primitives as Objects, we can pass arrays between µEngine Tasks as well.
6. JavaScript allows the programmer to create functions to modularize code. Rhino does not consider a function to be executable when determining the statements to run in a script.

Use or disclosure of data contained on this sheet is subject to the restriction on the title page of this document.

5. µEngine Script Structure

A µEngine script is a JavaScript script that transforms data into a format that is usable to the client. The µEngine script normally uses one or more µEngine Tasks. The µEngine Tasks allow the script to interact with the Java underlying the EDEX server to more efficiently perform the computations needed for the transformation. The exact format of the displayable product depends on the client. It could be a single image or a list of available data files. In general, the µEngine Tasks fall into a number of functional groups or segments. These segments are summarized in Table 1.

Table 1. Generic Steps in a µEngine Script

Segment	Description
Data Query	The data query Tasks are used to determine what data are available for processing. In order to perform a query, some knowledge of the metadata for each data type is required. Generally, the data type plug-in name is also required.
Data Decoding	The data decoding Tasks read the data from the EDEX data store and perform any required decoding on the raw data.
Data Processing	The data processing Tasks are used to transform the data.
Data Imaging	The data imaging Tasks are used to transform the processed data into a displayable format.
Data Output	The data output Tasks are used to convert the displayable data into a form that can be returned to the client.
Data Response	The data response Tasks are used to create a message to the client. The message includes status and/or product data.
System Interface	System interface tasks allow the µEngine to interact with other services in the EDEX server.

Note that not all scripts will include all these segments.

Each of these µEngine script segments is supported by one or more µEngine Tasks. In some cases, the Tasks are data type dependent. Table 2[1] lists the tasks that are currently available for each of the µEngine script segments.

Table 2. µEngine script Tasks for each script step

Segment	Available µEngine Tasks
Data Query	Catalog, ColormapQuery, LatestTimeQuery, ShapeFileQuery, SpatialQuery, TableQuery, TableUpdate, TermQuery
Data Decoding	DecodeRadarImage, GribExtractData, FileIn
Data Processing	ConvertWindsData, ObjectiveAnalysis, MapAsciiData, GribMap, GribSlicer, ColorMapImage, ReprojectImage, StopLightImage
Data Imaging	DataToXml, ImageOut, GribContourLine, GribImpacts
Data Output	FileOut, ShapeFile, SystemLog
Data Response	MakeResponseAscii, MakeResponseInline, MakeResponseNull, MakeResponseUri, MakeResponseXml
System Interface	ArvhiveSrvControl, ExecuteCommand, ReplayArchive, VtecObjectQuery, VtecQuery, VtecUpdateEvent

[1] Deprecated/obsolete tasks are not included in Table 2.

Use or disclosure of data contained on this sheet is subject to the restriction on the title page of this document.

Details on the µEngine Tasks, including Task descriptions and coding examples, are provided in a separate document. See Appendix A, References, for more information.

6. The TermQuery Task:

Most µEngine scripts that perform data transformations start with the TermQuery task. This task defines the query used to obtain the data for processing. The general pattern for using the TermQuery task is shown in the following JavaScript snippet.

```
var query = new TermQuery("{plug-in}");
query.setCount({max record count});
query.addParameter("{field name}","{field value}");
query.addParameter("{field name}","{field
value}","{operation}");
var results = query.execute();
```

The following examples provide additional information on the options for the TermQuery task. However, the following general comments are in order

1. In the TermQuery Task's constructor, the value of the plug-in argument is the name of the data-type plug-in used to decode the requested data.
2. The addParameter(…) method is used to add conditions to limit the query. When multiple addParameter(…) method calls are specified, all the conditions must all be met to identify the data.
3. In the addParameter(…) method, the (optional) third argument defines the comparison used in the query. When omitted, the operation is equality.
4. When TermQuery's execute() method is executed, what is returned is usually a java.lang.List derived object containing the metadata objects matching the query. It is possible, however, for the return to be empty (no results available) or null (the query failed). For this reason, the µEngine script must check the return value and handle the invalid returns as special cases. The usual way to handle the error is to use the MakeResponseNull Task to return a message indicating there were no results available.

As an example, consider the following JavaScript snippet using the TermQuery task:

```
var query = new TermQuery("obs");
query.setCount(3);
query.setSortBy("datatime");
query.addParameter("reporttype","METAR");
query.addParameter("stationid","KOMH,KOFF,KOMA","in");
var results = query.execute();
if (results == null || results.size() == 0) {
   var response = new MakeResponseNull("Query returned 0
results.",
                                        query);
   return response.execute();
}
// process the results.
```


Use or disclosure of data contained on this sheet is subject to the restriction on the title page of this document.

This query is equivalent to the following SQL select statement:

```
select *

from awips.obs

where reporttype='METAR'
   and stationed in ('KOMH', 'KOFF', 'KOMA')
order by datatime
limit 3
```

The result is all metadata records selected from the awips.obs metadata where the stationed is KOMH, KOFF, or KOMA. Metadata records are limited to METAR reports. The metadata are sorted by its datatime value and at most, three records are returned.

The "if" block following the call of TermQuery's execute() method is used to immediately return a message if the query returned no results. Note the dual test for both failure conditions. In this case, no attempt is made to distinguish between "no data" and "query failure."

Once the termQuery task completes and the results have been validated, the µEngine script has access to the metadata records that match the query.

The following examples explain how the script accesses the meta-data.

7. A µEngine scripting Example

In this example, we wish to obtain the latest METAR report for a reporting station with ICAO "KBOX." The result will be returned as an XML document.

7.a. Script Analysis

We will break the script into a couple of functions – one to perform the query and one to process the metadata results returned by the query. We will also provide mainline code to call the functions and validate the return from the query function.

Use or disclosure of data contained on this sheet is subject to the restriction on the title page of this document.

7.b. The Code:

```
function metarQuery(count,icao) {
   var query = new TermQuery("obs");
   query.addParameter("reporttype","METAR");
   query.addParameter("stationid",icao);
   query.setSortBy("datatime")
   query.setCount(count);
   return query.execute();
}
function processResponse(query) {
   var responses = new Array();
   for (i = 0; i < query.size(); i++) {
      var converter = new DataToXml(query.get(i));
      var converted = converter.execute();
      var response = new
MakeResponseAscii(query.get(i),converted);
      responses[i] = response.execute();
   }
   return responses;
}
var response = metarQuery(1,"KBOX");
if (response == null || response.size() == 0) {
   var reply = new MakeResponseNull("Query returned 0
results.",
                                    query);
   return reply.execute();
}
return processResponse(response);
```

Script 1. Sample METAR Retrieval

7.c. Comments

In this script, the following actions take place:

1. Rhino does not execute the functions until they are called, so the script actually starts with line that begins with "`var response`". This line calls the metarQuery function, passing in the values "1" and "KBOX."
2. Inside the metarQuery function:
   a. A TermQuery task is created to obtain the metadata. The METAR data are handled by the "obs" plug-in.
   b. addParameter(…) is called twice, once to specify a report type of "METAR." The second call specifies the ICAO. The value is the passed in value of icao, which was set to "KBOX" when the function is called.
   c. The setSortBy(…) method is used to specify the sort column, "datatime" in this case. This ensures that we retrieve the most recent available data.
   d. Finally, we call the execute method of the TermQuery task instance and return the result.
3. The "if" block is used to check the query return. If the query failed or there were no data, the "return" statement exits the script while returning a message as a null response object.

Use or disclosure of data contained on this sheet is subject to the restriction on the title page of this document.

4. In the final statement, we return the result of calling the processResponse(…) method, passing it the query results. In the processResponse(…) method:
   a. We create a JavaScript array to collect the processed data. Once created, the JavaScript array is used like a primitive array in Java. Unlike a Java array, however, JavaScript will automatically resize the Array to match subscripting, so it is not necessary to dimension[2] the array on creation.
   b. The "for" loop allows us to process the query results one record at a time. A note on the syntax involved: Rhino allows JavaScript to interact directly with Java objects, both instantiating the objects (using "new") and calling methods on the objects. Because TermQuery returned a java.util.List containing the records from the database query, we use the methods provided by java.util.List to access the records. Keep in mind that the "query" is a Java List object while response is a JavaScript Array object.
   c. Inside the "for" loop, we create a DataToXml Task to convert each query result to an XML string. DataToXml returns the XML in a Java String object. Note that we create the task, then call its execute() method to perform the work.
   d. Once the record has been converted to XML, we create a MakeResponseAscii Task to wrap the XML in a return object to return to the client. The return object, which is a Java object, is saved in the (JavaScript) responses Array to be returned later. As noted, the responses Array will resize as needed – in this case each time the "for" loop is executes.
   e. Once the "for" loop terminates, we exit the function, returning the responses Array.

When executed from within the µEngine, it is the responses Array that the µEngine gets back after the script executes. Within the µEngine, the JavaScript Array is converted into a Java Object array (i.e., the µEngine returns an Object[]). This is transparent to the script writer, however.

8. JavaScript Classes

Although JavaScript is normally used to write procedural function to enable client side processing in Web pages, the language itself is object based. In JavaScript, nearly everything is implemented as an object. Even functions are objects, which is the basis of user-defined classes. As in Java, the object has properties and methods: Combined, properties and methods are referred to as attributes. A script can add an attribute to a JavaScript object simply by assigning a value to the desired attribute. For example, the JavaScript Object class is the generic base class for all JavaScript classes. Thus,

```
var thing = new Object();
thing.thingy = "This is a thingy";
```

creates an Object called "thing" and assigns a value to its "thingy" property. Because the Object class does not include a "thingy" property, one is created[3].

[2] Redimensioning an Array can be an expensive operation if the Array is large. Because of that, it is good practice to set the dimension when the Array is created.
[3] But only for the specific instance of Object called "thing"!

Use or disclosure of data contained on this sheet is subject to the restriction on the title page of this document.

8.a. Class Prototypes

As in Java, a class represents a "potential" object. The class has two affiliated objects; the "this" object and the "prototype" object. The "this" object represents the actual class instance (or object); the "prototype" object represents the class definition. Attributes assigned to the "this" object are contained within the class instance; attributes assigned to the "prototype" object are shared by all instances of the class. When an attribute is created on a class instance, it is assigned to the "this" object for that instance. Thus, for example, if MyClass is a JavaScript class,

```
var myClass = new MyClass();
myClass.thing = "This is a thing";
```

assigns a value to the "thing" attribute of the myClass instance of MyClass. If the attribute does not exist, it is created. On the other hand,

MyClass.prototype.count = 3;

assigns a value of 3 to the "prototype" object for the MyClass class. As a result, every instance of MyClass will have an attribute called "count" with a value of "3".

In normal circumstances, classes will be constructed in such a manner that properties are associated with the class' "this" object and methods are associated with the "prototype" object.

8.b. Function Instances

In JavaScript, class definitions are built around functions. Because a function is an object, attributes may be assigned to the function's "this" and "prototype" objects. Properties assigned to the function's name are automatically assigned to the "prototype" object. Properties assigned to an instance of a function are assigned to the instances "this" object. To assign a method to a function's prototype, the prototype object must be explicitly named. But how do we get an instance of a function?

In JavaScript, the "new" operator is used to create an instance of a function. Consider the following code snippet:

```
function myFunction() {
}
function method() {
}
myFunction.prototype.method = _method;

var thing = new myFunction();

thing.thingy = "This is a thing";
```

In this snippet, we first define a function called "myFunction", which actually does nothing. The function "method" is then defined and assigned to myFunction's prototype. (This allows us to invoke "method" on the myFunction instance "thing", i.e. "thing.method()" runs the "method" function on the "thing" instance of myFunction.) Then, we create an instance (i.e., a myFunction object) and assign it to "thing". At this point, "thing" contains an instance of myFunction.

Use or disclosure of data contained on this sheet is subject to the restriction on the title page of this document.

When the "new" operator creates the function instance, it also executes the function itself. This allows us to add code to the function to perform specific class initializations. For example, the following code snippet creates a myFunction object and assigns a value to the "thingy" property as the object is created.

```
function myFunction(value) {
   this.thingy = value;
}

var thing = new myFunction("This is a thing");
```

In this case, the function is set up so that the value for "thingy" is passed to the function when it is constructed.

The function body may perform assignments to both the "this" object and the "prototype" object. Assignments to the prototype are normally coded outside the function definition so that they are not executed each time an object is created.

8.c. Class Constructors

This brings us to the standard pattern for creating a JavaScript class.

1. Define the function that defines how to create a class object. Use the constructor to define instance attributes, referring to them using the "this" prefix. This is very similar to a class constructor in Java.
2. Define the methods that are to be associated with the class. The convention here is to prefix the method definitions with an underscore (_) to indicate they are private. Any reference to instance attributes must use the "this" prefix.
3. Assign the class methods to the function's prototype.

8.d. A Hello World Example

This example uses the µEngine's SystemLog and MakeResponseNull tasks to create a µEngine that logs a message to the EDEX system log. The message is also returned to the client. First the class definition:

Use or disclosure of data contained on this sheet is subject to the restriction on the title page of this document.

```
/* class constructor */
function HelloWorld() {
   this.message = "";
   this.level = "info";
}
/* class methods */
function _setMessage(text) {
   this.message = text;
}
function _setLevel(level) {
   this.level = level;
}
function _execute() {
   var logger = new SystemLog();
   logger.log(this.level,this.message);
   var response = new MakeResponseNull(this.message,null);
   return response.execute();
}
/* attach methods to class prototype – use aliases */
HelloWorld.prototype.execute = _execute;
HelloWorld.prototype.setMessage = _setMessage;
HelloWorld.prototype.setLevel = _setLevel;
```

Script 2. HelloWorld Class for µEngine

The script to use this class is rather short.

```
var runner = new HelloWorld();
runner.setLevel("info");
runner.setMessage("Hello World");
runner.execute();
```

Script 3. Using the HelloWorld Class

8.e. The µEngine's JavaScript Library

The µEngine supports the creation of a JavaScript library. The ADE includes a number of basic µEngine JavaScript class files. When EDEX is installed, the scripts are located in the "edex/opt/esb/js" directory of the EDEX installation. A valid JavaScript class definition is saved in a file having a ".js" extension. µEngine can automatically load the script file using an include directive at the start of the µEngine script. Assuming the script above (Script 2) is saved in a file called "HelloWorld.js" in the µEngine script library; the following snippet will log "Hello World" to the EDEX system log.

```
include("HelloWorld.js");
var runner = new HelloWorld();
runner.setLevel("info");
runner.setMessage("Hello World");
runner.execute();
```


Use or disclosure of data contained on this sheet is subject to the restriction on the title page of this document.

9. A µEngine Script using Classes

As a final example, we will convert the first script example into a script using a JavaScript class. The code for the class is shown in Script 4. Script 5 shows a user script to execute the class.

```
function MetarQuery() {
   this.count = 0;
   this.sortBy = "";
   this.query = new TermQuery("obs");
}
function _execute() {
   var results = this.performQuery();
   if (results == null || results.size() == 0) {
      var reply = new MakeResponseNull
                     ("Query returned 0 results.",
                      this.query);
      return reply.execute();
   }
   return this.processQuery(results);
}
function _addParameter(name,value) {
   this.query.addParameter(name,value);
}
function _setCount(count) {
   this.count = count;
}
function _setSortBy(name) {
   this.sortBy = name;
}
function _performQuery() {
   this.query.setSortBy(this.sortBy);
   this.query.setCount(this.count);
   return this.query.execute();
}
function _processQuery(query) {
   var responses = new Array();
   for (i = 0; i < query.size(); i++) {
      var converter = new DataToXml(query.get(i));
      var converted = converter.execute();
      var response = new MakeResponseAscii
                 (query.get(i),converted);
      responses[i] = response.execute()
   }
   return responses;
}
/* add the functions to the prototype */
MetarQuery.prototype.execute = _execute;
MetarQuery.prototype.addParameter = _addParameter;
MetarQuery.prototype.setCount = _setCount;
MetarQuery.prototype.setSortBy = _setSortBy;
MetarQuery.prototype.performQuery = _performQuery;
MetarQuery.prototype.processQuery = _processQuery;
```

Script 5. METAR Retrieval as a Class

Use or disclosure of data contained on this sheet is subject to the restriction on the title page of this document.

The client script to use this class (assuming the class was saved in “MetarQuery.js”) follows.

```
include("MetarQuery.js");
var runner = new MetarQuery();
runner.setCount(1);
runner.setSortBy("datatime");
runner.addParameter("reporttype","METAR");
runner.addParameter("stationid",icao);
runner.execute();
```

10. Additional Considerations

Both JavaScript and the µEngine support additional options. The µEngine supports a simple form of script running via a subscription service. Subscribable scripts, which must be written as classes, require additional specific coding. There are examples of JavaScript class in the ADE JavaScript library. The best way to get a feel for what is available is to examine some of the scripts in the ADE JavaScript class library.

11. Known Issues

There are a number of problems that have been identified with using JavaScript based scripts in the EDEX subscription service. Since CAVE does not depend on subscriptions, this is not an issue at this point.

Use or disclosure of data contained on this sheet is subject to the restriction on the title page of this document.

Appendix A. References

JavaScript Scripting:

1. Information on JavaScript scripting is available on the web. A good place to start is at the Rhino web site (http://www.mozilla.org/rhino/).
2. There are a number of good books about JavaScript scripting. One such is *JavaScript The Definitive Guide,* which is available from O'Reilly (http://www.oreilly.com).

The AWIPS EDEX:

1. The AWIPS EDEX code baseline is located on the Installer's media. The code is normally installed on a development computer. See the Release Notes (ReleaseNotes.txt) for more information.
2. AWIPS EDEX documentation is located in the docs directory on the ADE install media.
3. JavaDoc is embedded in the baseline code. The web version is generated using the ANT build script. See the "AWIPS ADE EDEX Build Procedure" (AWIPS_ADE_EDEX_Build_Procedure.pdf), located on the ADE install media, for information on generating the browser viewable JavaDoc.

Use or disclosure of data contained on this sheet is subject to the restriction on the title page of this document.

Appendix B. Acronym/Abbreviation List

ANT	(not an acronym) Another Nifty Tool, a Java oriented “build” tool
AWIPS	Advanced Weather Information Processing System
ADE	AWIPS Development Environment
EDEX	Environmental Data Exchange
IDE	Integrated Development Environment
JAR	Java Archive
JDK	Java Development Kit
JRE	Java Runtime Environment
METAR	Meteorological Aerodrome Report
PC	Personal Computer
SOA	Service Oriented Architecture
TAF	Terminal Aerodrome Forecast
µEngine	MicroEngine
URI	Universal Resource Indicator
XML	eXtensible Markup Language

Use or disclosure of data contained on this sheet is subject to the restriction on the title page of this document.